ĐA LUỒNG & ĐỒNG BỘ HÓA

1 Giới thiệu

Trong cách thức lập trình truyền thống, sau khi viết xong mã nguồn chương trình sẽ được biên dịch thành một dạng mã mà máy tính có thể hiểu được (machine code). Mã này được khối xử lý trung tâm (CPU) xử lý, chương trình được xử lý một cách tuần tự. Thời gian thực thi các câu lệnh có thể khác nhau, nhưng nếu một câu lệnh chưa được thực hiện xong thì các câu lệnh khác sẽ không được chạy, đó gọi là xử lý đơn luồng (**Single-Threaded**). Ưu điểm của lập trình đơn luồng (**Single-Thread programming**) là đơn giản, nếu một câu lệnh không được hoàn thành thì câu lệnh khác sẽ không được thực thi. Điều này giúp cho các bạn dễ dàng tìm kiếm và biết được các lỗi phát sinh ở đâu.

Tuy nhiên, trong thực tế người sử dụng máy tính luôn có nhu cầu hệ thống của họ có thể làm được nhiều việc cùng lúc. Họ có thể vừa làm việc với phần mềm xử lý văn bản, trong khi các phần mềm khác tải file, quản lý hàng đợi in ấn và nghe nhạc. Ngay cả một ứng dụng đơn cũng có thể làm được nhiều việc một lúc. Từ đó, dẫn đến khái niệm Lập trình xử lý đa luồng công việc (**Multi-Threaded**). Với lập trình đa luồng (**Mutil-Thread programming**), các lập trình viên phải nhìn nhận một cách khác về phần mềm. Thay vì thực hiện một loạt các câu lệnh một cách tuần tự, thì chúng ta có thể thực hiện nhiều câu lệnh, nhiều nhiệm vụ đồng thời. Các câu lệnh được thực hiện cùng một lúc, chứ không phải câu lệnh này thực hiện xong câu lệnh kia mới thực hiện. Một ứng dụng đa luồng có thể thực hiện được nhiều nhiệm vụ trong cùng một thời điểm, cùng một không gian bộ nhớ, và các luồng có thể cho phép chia sẻ các biến dữ liệu để cùng xử lý. Có nhiều cách giải quyết vấn đề này, trong bài này sẽ giới thiệu một giải pháp là sử dụng thread, mỗi thread sẽ thực thi một công việc và thực thi song song với các công việc còn lại.

2 Lập trình đa luồng

Java cũng giống như nhiều ngôn ngữ lập trình cao cấp khác, hỗ trợ tốt lập trình đa luồng. Trong Java, chúng ta có thể tạo ra các luồng bằng 2 cách:

- Tạo ra các thread bằng cách sử dụng lớp **java.lang.Thread**
- Tạo ra các thread thông qua **interface Runable**

2.1 Tạo một ứng dụng Multi-Thread với lớp Thread

Lớp **Java.lang.Thread** cung cấp các phương thức:

- Start
- Suspend
- Resmune
- Stop

Cách đơn giản nhất là kế thừa lớp **Java.lang.Thread** và ghi đè lên phương thức **run()**

Thread sẽ không được tự động kích hoạt, muốn kích hoạt một thread ta phải sử dụng phương thức **start()**

```
import java.io.*;

//This class extends Thread

class WriteThread extends Thread

{

    // This method is called when the thread runs

    // Overrided method

    public void run()

    {

          PrintWriter pw = new PrintWriter(new

                              OutputStreamWriter(System.out));

          pw.println("hello!");

          pw.close();

    }

}
```

```
public class JThread
{
     public static void main(String[] args)
     {
          // Create and start the thread
          Thread thread = new WriteThread();
          thread.start();
      }
}
```

Sau khi phương thức **start()** của một thread được gọi, thì thread này sẽ gửi một yêu cầu để tạo ra thread riêng biệt, sau đó phương thức run được xử lý. Nếu trong chương trình ta tạo ra nhiều thread, có thể sử dụng dòng lệnh **Thread.sleep(k)** để khai báo thời gian chờ để các thread được xử lý, thời gian chờ càng lớn thì việc xử lý kết quả càng lâu.

2.2 Tạo một ứng dụng Multi-Thread sử dụng Interface Runable

Sử dụng lớp **Java.lang.Thread** là một cách đơn giản để tạo ra một ứng dụng đa luồng, nhưng đó *chưa phải là cách tốt nhất*. Ngôn ngữ *Java chỉ hỗ trợ đơn kế thừa* (ngôn ngữ C++ hỗ trợ đa kế thừa). Điều đó cũng có nghĩa: nếu một lớp đã được kế thừa từ một lớp khác thì nó không thể nào kế thừa từ lớp **Java.lang.Thread** được nữa.

Ví dụ: Có một ứng dụng xử lý đa luồng cho quản lý Nhân viên. Khi lớp Nhân viên kế thừa từ lớp Người thì nó sẽ không được phép tiếp tục kế thừa từ lớp Java.lang.Thread.

Một cách khác để tạo ra một ứng dụng đa luồng đó là **implement** từ **Interface** java.lang.Runnable**. Interface Runnable** cung cấp một phương thức duy nhất **run()**. Khi gọi phương thức **start()** của thread thì phương thức **run()** của Inteface Runnable cũng được thực thi.

```
import java.io.*;
class BasicThread2 implements Runnable
{
   // This method is called when the thread runs
   // Overrided method
   public void run()
   {
         PrintWriter pw = new PrintWriter(new
                                 OutputStreamWriter(System.out));
         pw.print("Nhap vao 1 chuoi: ");
         pw.flush();
         try
         {
               BufferedReader br = new BufferedReader(new
                                    InputStreamReader(System.in));
               String s;
               s = br.readLine();
               pw.println("Ban vua nhap chuoi " + s);
               pw.close();
         }
         catch (IOException e)
         {
               pw.println("Loi io!");
               pw.close();
               e.printStackTrace();
         }
   }
```

```
}
public class JThread
{
   public static void main(String[] args)
   {
         // Create the object with the run() method
         Runnable runnable = new BasicThread2();

         System.out.println("Tao thread 1: ");
         // Create the thread supplying it with the runnable object
         Thread t1 = new Thread(runnable);

         System.out.println("Tao thread 2: ");
         // Create the thread supplying it with the runnable object
         Thread t2 = new Thread(runnable);

         System.out.println("chay 2 thread");
         // Start two threads
         t1.start();
         t2.start();
   }
}
```

2.3 Các hàm điều khiển Thread

2.3.1 Ngắt một Thread

Khi sử dụng phương thức **Thread.Sleep()** thì chương trình phải bắt các ngoại lệ để xử lý. Lý do nếu một thread bị dừng lại trong một khoảng thời gian lâu, mà trong khoảng thời gian đó nó không thể tự đánh thức nó được. Tuy nhiên, nếu thread đó cần được đánh thức sớm hơn, ta có thể sử dụng phương thức **Interrupt()** để ngắt nó.

```
class InterruptThreadDemo extends Thread {
    public void run() {
       System.out.println("I slept, wake up me after 6 minutes");
    try {
       //1000 = 1 giây => 6 minutes = 1000 * 60 * 6
       Thread.sleep(1000*60*6);
       System.out.println("Sleeping is my habit");
    }
    catch (InterruptedException ex) {
       System.err.println ("Just 2 minutes! Please...");
    }
}
public static void main(String[] args) throws IOException {
    Thread t1 = new InterruptThreadDemo();
    t1.start();

    System.out.println("Press Enter key to wake up/ interrupt the thread");
    System.in.read();

    //Ngắt Thread
    t1.interrupt();
}
```

Trong ví dụ đoạn code trên, đã sử dụng chính hàm **main** (hàm **main** cũng là một **thread**), sau khi người dùng ấn nút **enter,** hàm main sẽ gửi một thông điệp để đánh thức **thread** đang ngủ.

2.3.2 Dừng một Thread

Đôi khi ta muốn dừng một thread trước khi nó hoàn thành, ta sẽ yêu cầu một thread khác gửi thông điệp tới thread bằng cách gọi phương thức **thread.Stop()**. Điều này yêu cầu thread điều khiển phải giữ một tham chiếu tới thread muốn dừng.

```
public class StopThreadDemo extends Thread {
    public void run() {
```

```
        System.out.println ("Thread dang dem : ");
        for (int i = 0; i < 10000; i++) {
            // Tăng count sau mỗi lần đếm
            System.out.print (i + " ");
            try {
                // Ngủ 1/2 giây rồi tiếp tục công việc
                Thread.sleep(500);
            }
            catch (InterruptedException ie) {}
        }
}
public static void main(String[] args) throws IOException {
    Thread t1 = new StopThreadDemo();
    t1.start();

    // Muốn dừng lại việc đếm của Thread
    System.out.println ("An Enter de dung lai.");
    System.in.read();

    // Dừng thread lại
    t1.stop();
}
```

2.3.3 Tạm dừng và phục hồi Thread

Cách đúng đắn để ngừng một thread đang chạy là thiết lập một biến mà thread này kiểm tra thường xuyên. Khi một thread phát hiện rằng biến đó đã được thiết lập, nó sẽ trở về từ phương thức *run()*.

Lưu ý*: *Thread.suspend() và *Thread.stop()* cung cấp các phương thức không đồng bộ để ngừng một thread. Tuy nhiên, những phương thức này đã không còn được hỗ trợ do sử dụng chúng rất không an toàn. Chúng thường gây nên deadlock và lỗi khi giải phóng tài nguyên. Phương thức Thread.resume() dùng để tiếp tục một thread đã bị suspend.

import java.io.*;

```
public class MyThread extends Thread
{
   Boolean stop = false;
   Integer n = new Integer(100);
   public void run()
   {
          while(!stop)
          {
                n = n + 10;
          }
   }
public class JStopThread
{
   public static void main(String[] args)
   {
          MyThread mt = new MyThread();
          mt.start();
          PrintWriter pw = new PrintWriter(new
                                OutputStreamWriter(System.out));
          pw.println("Ask for killing the thread!");
          pw.flush();
          try
          {
                Thread.sleep(100);
          }
          catch (InterruptedException e)
          {
```

```
                e.printStackTrace();
            }
            mt.stop = false;
            pw.println("n = " + mt.n);
            pw.println("The thread was killed.");
            pw.close();
    }
}
```

Ngoài ra, Java cũng cung cấp phương thức:

Giải phóng thời gian cho CPU (Yielding CPU Time) để nâng cao hiệu quả cho hệ thống (giải quyết tình huống khi một thread rơi vào trạng thái đợi một sự kiện xảy ra hoặc đi vào vùng mã lệnh). Ta có thể dùng phương pháp static yield() của thread để giải phóng thời gian CPU cho thread hiện hành và chỉ xử lý được trên thread hiện hành.

Đợi một thread kết thúc một công việc nào đó, ta dùng phương thức **IsAlive()** để xác định thread còn chạy hay không. Tuy nhiên, việc thường xuyên gọi phương thức IsAlive thì hiệu quả của CPU sẽ thấp, để tránh tình trạng này, ta có thể dùng phương thức **joint()** để đợi một thread kết công việc.

3 Đồng bộ hóa

Ở đây ta có một khái niệm mới là **miền găng** (a race condition). Miền găng nói đến sự xung đột khi đa truy cập không được quản lý hợp lý trong lúc làm việc với nhiều thread. Khi làm việc với nhiều thread, có nhiều hơn một thread muốn truy cập cùng một tài nguyên chia sẻ (một file hoặc biến) tại cùng một thời điểm sẽ xảy ra sự mất đồng bộ. Ví dụ, một thread có thể cố gắng đọc dữ liệu, trong khi thread khác cố gắng thay đổi dữ liệu. Trong trường hợp này, dữ liệu có thể bị sai lạc.

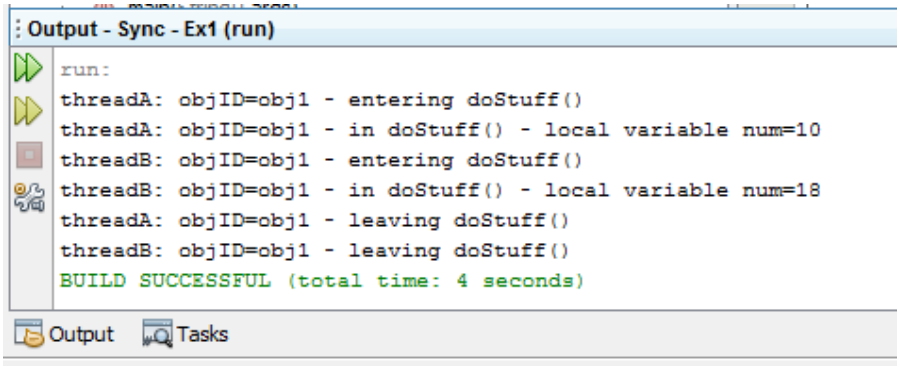
Ví dụ dưới đây mô phỏng 2 thread cùng truy cập vào một phương thức:

```
1   package syncex1;
2   public class BothInMethod extends Object {
3       private String objID;
4
5       public BothInMethod(String objID){
6           this.objID = objID;
7       }
8
9       public synchronized void doStuff(int val) {
10          print("entering doStuff()");
11          int num = val*2 + objID.length();
12          print("in doStuff() - local variable num=" + num);
13
14          // slow things down to make observations
15          try { Thread.sleep(2000); }
16          catch ( InterruptedException x ) { }
17
18          print("leaving doStuff()");
19      }
20
21      public void print(String msg) {
22          threadPrint("objID=" + objID + " - " + msg);
23      }
24
25      public static void threadPrint(String msg) {
26          String threadName = Thread.currentThread().getName();
27          System.out.println(threadName + ": " + msg);
28      }
29
30      public static void main(String[] args) {
31          final BothInMethod bim = new BothInMethod("obj1");
32
33          Runnable runA = new Runnable() {
35              public void run() {
36                  bim.doStuff(3);
37              }
38          };
39
40          Thread threadA = new Thread(runA, "threadA");
41          threadA.start();
42
43          try { Thread.sleep(200); }catch ( InterruptedException x ) { }
44
46          Runnable runB = new Runnable() {
47              public void run() {
48                  bim.doStuff(7);
49              }
50          };
51
52          Thread threadB = new Thread(runB, "threadB");
53          threadB.start();
      }
  }
```

Trong phương thức main(), đối tượng BothMethod được khởi tạo với identifier là obj1 (dòng 31). Tiếp theo 2 threadA và threadB được tạo ra để truy cập đồng thời vào phương thức doStuff(). Sauk hi threadA bắt đầu, nó gọi doStuff() và truyền vào giá trị 3, khoảng 200mili giây sau thread được bắt đầu, nó cũng gọi phương thức doStuff() và truyền vào giá trị 7.

Cả threadA và thread đều cùng ở trong doStuff() (từ dòng 9->19) tại cùng một thời điểm. Bên trong doStuff(), biến cục bộ num được tính toán thông qua tham số val và biến thành viên objID (dòng 11). Bởi vì, threadA và threadB đều gán một biến val khác nhau, nên giá trị biến num sẽ khác nhau cho mỗi thread. Phương thức sleep() được sử dụng trong doStuff() nhằm làm chậm lại để đảm bảo rằng cả hai thread đều ở trong cùng một phương thức của cùng một đối tượng một cách đồng thời.

Kết quả chạy ví dụ trên như sau:



Như vậy việc 2 thread cùng truy cập vào một phương thức cùng một thời điểm sẽ dẫn đến tranh chấp trong việc truy cập các tài nguyên cục bộ. Trong trường hợp này, ta cần cho phép một thread hoàn thành trọn vẹn nhiệm vụ của nó (thay đổi giá trị của biến), rồi các thread kế tiếp mới được phép thực thi. Để giải quyết vấn đề này ta phải đồng bộ hóa các thread, việc đồng bộ hóa nhằm bảo đảm khi có nhiều hơn một thread truy cập tới một tài nguyên được chia sẽ, thì tài nguyên đó sẽ chỉ được sử dụng bởi một thread tại một thời điểm. Phương thức được đồng bộ hóa sẽ báo cho hệ thống đặt một khóa vòng một tài nguyên riêng biệt.

Các thread được đồng bộ hoá trong Java sử dụng thông qua một monitor (cũng được gọi là một semaphore). Một monitor là một đối tượng (object) cho phép một thread truy cập vào một tài nguyên. 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 trong số các luồng đang chờ monitor có thể truy cập vào tài nguyên dùng chung tương ứng với monitor đó.

Để giải quyết vấn đề miền găng ta có hai giải pháp là **Synchronized Methods** và **Synchronized Blocks (Statements)**

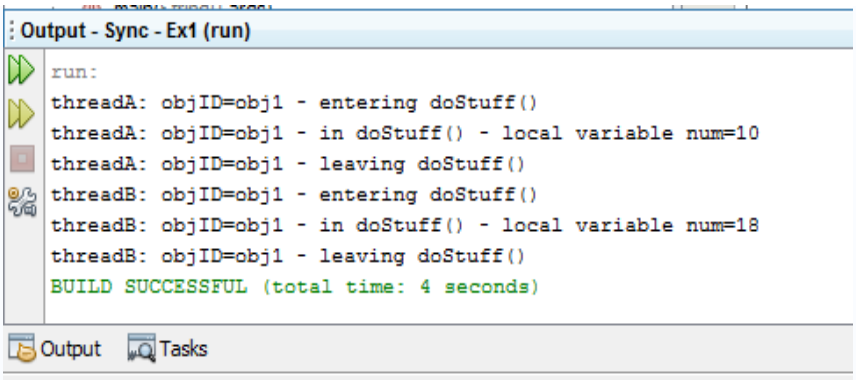
3.1 Phương thức đồng bộ hóa (Synchronized Method):

Để tạo một phương thức được đồng bộ hóa, đơn giản chỉ thêm từ khóa synchronized vào khai báo một phương thức. Việc bổ sung từ khóa synchronized nhằm đảm bảo chỉ có một thread được phép ở bên trong phương thức tại một thời điểm.

Cụ thể ở ví dụ trên ta thêm từ khóa s*ynchronized* vào trước phương thức doStuff (dòng 9) như sau:

```
public synchronized void doStuff(int val) {
```

Và kết quả chạy chương trình là:



Lúc này sự động bộ hóa xảy ra. Khi threadA vào doStuff thì threadB phải chờ ở ngoài đến khi threadA chạy xong.

3.2 Đồng bộ khối (Synchronized Statement Block)

Tạo ra các phương thức đồng bộ với từ khóa synchronized trong phạm vi các lớp là một con đường dễ dàng và có hiệu quả của việc thực hiện sự đồng bộ. Tuy nhiên, điều này không có hiệu quả trong tất cả các trường hợp. Đôi khi ta chỉ muốn đồng bộ việc truy cập vào một đối tượng (object) của một lớp.

Đồng bộ khối được sử dụng khi không cần phải đồng bộ toàn bộ phương thức hoặc khi muốn nhận lock trên một đối tượng khác. Để đồng bộ truy cập một đối tượng của lớp này, ta gọi các phương thức mà lớp này định nghĩa, được đặt trong một khối đồng bộ.Tất cả các được đặt trong một câu lệnh đồng bộ như sau:

```
synchronized(object){
        // block code
}
```

Ở đây "object" là một tham chiếu đến một đối tượng được đồng bộ.

Ví dụ:

```
class Client{
    BankAccount account;
    // ...
    public void updateTransaction(){
        synchronized(account){        // (1) Khối đồng bộ
            account.update();         // (2)
        }
    }
}
```

Ngoài ra, có thể sử dụng statement block để thay thế cho phương thức được synchronized như sau:

```
public synchronized void setPoint(int x, int y) {
    this.x = x;
    this.y = y;
}
```

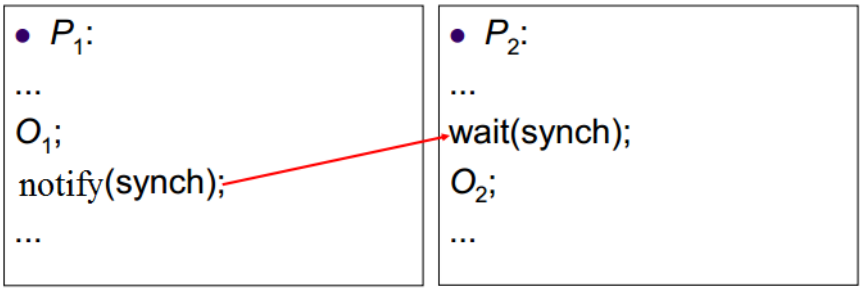
Thay thế bằng statement block như sau:

```
public void setPoint(int x, int y) {
    synchronized ( this ) {
        this.x = x;
        this.y = y;
    }
}
```

3.3 Phối hợp hoạt động

Ngoài việc kiểm soát nhiều thread cùng truy cập vào tài nguyên dùng chung tại một thời điểm, còn có một vấn đề nữa là ta phải phối hợp hoạt động giữa các thread với nhau.

Xét hai thread P1 và P2. Trong đó, P1 cần thực hiện toán tử O1, P2 cần thực hiện O2 và điều kiện là O2 chỉ được thực hiện sau khi toán tử O1 đã hoàn thành. Sử dụng semaphore là phù hợp trong trường hợp này. Chúng ta chỉ cần thêm các dòng lệnh đồng bộ hóa vào chương trình như sau:



Trong lớp semaphore, chúng ta có thể sử dụng tương ứng hàm *Semaphore.acquire()* như Wait() và Semaphore.*release()* như là Notify().

Ví dụ sau minh họa chương trình có 2 thread hoạt động đồng thời. Quy tắc là "Thread 1 phải thêm 10 vào giá trị của biến N trước khi thread 2 nhân đôi giá trị N".

```
import java.util.concurrent.Semaphore;

class comvar {
        public static int N = 111;
        public static Semaphore sem1 = new Semaphore(0);
        public static Semaphore sem2 = new Semaphore(0);
}

class Thread1 extends Thread {
        public void run()
        {
                comvar.N += 10;
                comvar.sem1.release();//báo với thread2 là thread1 đã hoàn thành
                System.out.println("sem1 release");
        }
}

class Thread2 extends Thread {
        public void run()
        {
                try {
                        System.out.println("sem1 acquire");
                        comvar.sem1.acquire(); //chờ trong khi thread1 tăng N lên 10
                } catch (InterruptedException e) {
                        System.out.println("loi");
                        e.printStackTrace();
                }
                comvar.N *= 2;
                comvar.sem2.release(); //báo với hàm main là thread đã hoàn thành
                System.out.println("sem2 release");
        }
}

public class JSemaphore extends Thread {
        public static void main(String[] args) {
                Thread1 t1 = new Thread1();
                t1.start();
                Thread2 t2 = new Thread2();
                t2.start();

                try {
                        System.out.println("sem2 acquire");
                        comvar.sem2.acquire();//chờ cho thread2 thực thi xong
                } catch (InterruptedException e) {
                        System.out.println("loi");
                        e.printStackTrace();
                }

                System.out.println("N = " + comvar.N);
        }
}
```

Kết quả chạy chương trình là: