LẬP TRÌNH ĐA LUỒNG

NỘI DUNG TRONG CHƯƠNG

Chương 8 đề cập đến các vấn đề về lập trình đa luồng, các cách tạo luồng và quan trọng là việc xử lý các vấn đề liên quan đến đồng bộ,

- *Cách tạo tiểu trình (Thread) bằng Thread, Runable*
- *Các xử lý đồng bộ bằng khóa*
- *Sử dụng Fork-Join*
- *Tạo và quản lý tiến trình*

MỘT SỐ KHÁI NIỆM CƠ BẢN

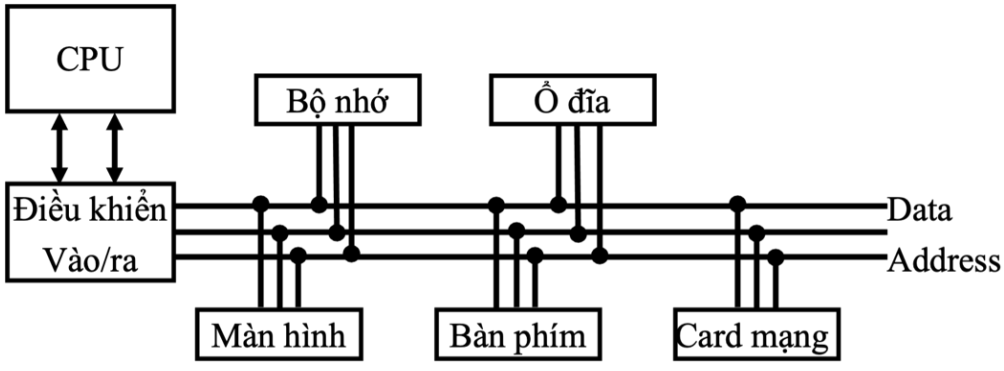
Vòng lặp thăm dò và ngắt

CPU sử dụng toàn bộ thời gian của nó để tìm nạp các lệnh từ bộ nhớ và thực thi chúng từ bộ nhớ chính (RAM).

CPU và bộ nhớ chính chỉ là hai trong số nhiều thành phần trong một hệ thống máy tính thực. Một hệ thống máy tính hoàn chỉnh bao gồm các thiết bị khác như: ổ đĩa cứng, bàn phím, chuột, màn hình, thiết bị kết nối mạng v.v... Danh sách các thiết bị hoàn toàn mở và hệ thống máy tính được xây dựng để chúng có thể dễ dàng mở rộng bằng cách thêm các thiết bị mới. Bằng cách nào đó, CPU phải giao tiếp và điều khiển tất cả các thiết bị này. Cách thức hoạt động là đối với mỗi thiết bị trong hệ thống chúng sẽ có một trình điều khiển thiết bị (Driver). Trình điều khiển này sẽ đóng vai trò trung gian giúp CPU có thể giao tiếp với các loại thiết bị. Như vậy khi cài đặt thiết bị mới trên hệ thống thường có hai bước: cắm thiết bị vật lý vào máy tính và cài đặt phần mềm trình điều khiển thiết bị. Nếu không có trình điều khiển thiết bị, thiết bị vật lý thực tế sẽ vô dụng, vì CPU sẽ không thể giao tiếp với nó.

Một hệ thống máy tính bao gồm nhiều thiết bị thường được tổ chức bằng cách kết nối các thiết bị đó với một hoặc nhiều đường Bus. Bus là một tập hợp các dây dẫn chứa nhiều loại thông tin giữa các thiết bị được kết nối với các dây đó. Các dây này sẽ truyền dữ liệu, địa chỉ và tín hiệu điều khiển. Địa chỉ hướng dữ liệu đến một thiết bị cụ thể hoặc một thành phần của thiết bị đó.

Một hệ thống máy tính khá đơn giản có thể được tổ chức như sau:



Hình 8-1 Tổ chức của một hệ thống máy tính đơn giản

Các thiết bị như bàn phím, chuột và giao diện mạng làm việc với CPU thông qua cơ chế ngắt.

Làm thế nào để CPU biết rằng dữ liệu ở đó? Một ý tưởng đơn giản nhưng không khả quan lắm, đó là để CPU liên tục kiểm tra dữ liệu đến. Bất cứ khi nào nó tìm thấy dữ liệu, nó sẽ xử lý dữ liệu đó. Phương pháp này được gọi là thăm dò (polling), vì CPU liên tục thăm dò các thiết bị đầu vào để xem liệu chúng có bất kỳ dữ liệu đầu vào nào để báo cáo hay không. Mặc dù thăm dò rất đơn giản, nhưng nó cũng rất kém hiệu quả. CPU sẽ lãng phí rất nhiều thời gian chỉ để chờ đầu vào.

Để tránh sự kém hiệu quả này, ngắt thường được sử dụng thay vì thăm dò. Ngắt là một tín hiệu được gửi bởi một thiết bị khác đến CPU. CPU đáp ứng tín hiệu ngắt bằng cách tạm dừng bất cứ điều gì nó đang làm để đáp ứng ngắt. Khi nó đã xử lý ngắt, nó sẽ trở lại những gì nó đang làm trước khi xảy ra ngắt. Ví dụ: khi nhấn một phím trên bàn phím máy tính, một ngắt bàn phím sẽ được gửi đến CPU. CPU phản hồi lại tín hiệu này bằng cách ngắt những gì nó đang làm, đọc phím đã nhấn, xử lý nó và sau đó quay lại tác vụ mà nó đã thực hiện trước khi bạn nhấn phím. Ngắt giúp CPU có thể làm việc hiệu quả với các xử lý "không đồng bộ", tức là vào những thời điểm không thể đoán trước.

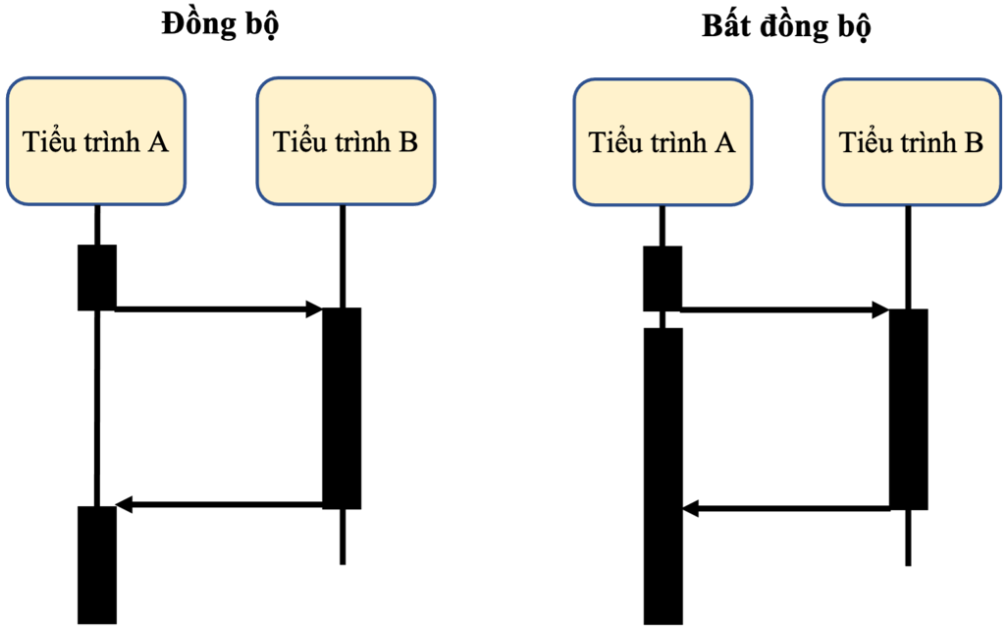
Một ví dụ khác về cách sử dụng ngắt, hãy xem xét điều gì sẽ xảy ra khi CPU cần truy cập dữ liệu được lưu trữ trên đĩa cứng. CPU chỉ có thể truy cập dữ liệu trực tiếp nếu nó nằm trong bộ nhớ chính. Dữ liệu trên đĩa phải được sao chép vào bộ nhớ trước khi nó có thể được truy cập. Tuy nhiên, ổ đĩa cực kỳ chậm so với bộ nhớ chính. Khi CPU cần dữ liệu từ đĩa, nó sẽ gửi tín hiệu đến ổ đĩa để bảo nó định vị dữ liệu và chuẩn bị sẵn sàng (tín hiệu này được gửi đồng bộ, dưới sự điều khiển của một chương trình thông thường). Sau đó, thay vì chỉ đợi khoảng thời gian dài và không thể đoán trước mà ổ đĩa sẽ mất để thực hiện việc này, CPU sẽ tiếp tục với một số tác vụ khác. Khi ổ đĩa có sẵn dữ liệu, nó sẽ gửi tín hiệu ngắt đến CPU. Bộ xử lý ngắt sau đó mới có thể đọc dữ liệu được yêu cầu.

Tất cả các máy tính hiện đại đều sử dụng đa nhiệm để thực hiện nhiều tác vụ cùng một lúc. Một số máy tính có thể được sử dụng bởi nhiều người cùng một lúc. CPU rất nhanh vậy nên nó có thể nhanh chóng chuyển việc phục vụ từ người dùng này sang người dùng khác, dành một phần nhỏ giây cho mỗi người dùng. Ứng dụng đa nhiệm này được gọi là chia sẻ thời gian. Nhưng một máy tính cá nhân hiện đại chỉ với một người dùng duy nhất cũng sử dụng đa nhiệm.

Xử lý đồng bộ và bất đồng bộ

Mỗi tác vụ riêng lẻ mà CPU đang làm việc được gọi là một luồng. Các máy tính hiện nay với cơ chế cho phép thực thi nhiều luồng cùng hoạt động tại một thời điểm. Điều này nghĩa là nhiều công việc có thể được giải quyết đồng thời. Các ngôn ngữ lập trình có thể triển khai điều này bằng cơ chế hoạt động đa luồng, xử lý đồng bộ và bất đồng bộ:

- **Xử lý đồng bộ** là code sẽ được chạy tuần tự theo trình tự đã viết sẵn – tức là đoạn code ở dưới phải đợi cho tới khi đoạn code ở trên trả ra kết quả.
- **Xử lý bất đồng bộ** là những xử lí mà đoạn code ở dưới có thể tiếp tục chạy mặc dù đoạn code ở trên chưa được xử lý hết và trả về kết quả.



Hình 8-2 Xử lý đồng bộ và bất đồng bộ

Nhiều CPU có thể thực thi nhiều hơn một công việc cùng lúc, những CPU như vậy chứa nhiều lõi (core), mỗi lõi trong số đó có thể chạy một hoặc nhiều luồng. Vì thường xuyên có nhiều luồng được thực thi cùng lúc nên máy tính bắt buộc phải có cơ chế để có thể phân bổ thời gian từ luồng này sang luồng khác, giống như máy tính chia sẻ thời gian chuyển sự phục vụ của nó từ người dùng này sang người dùng khác. Nói chung, một chuỗi lệnh đang được thực thi sẽ tiếp tục chạy cho đến khi một trong số những điều sau xảy ra:

- Luồng có thể tự động nhường quyền kiểm soát, để cho các luồng khác có cơ hội chạy.
- Luồng có thể phải đợi một số sự kiện không đồng bộ xảy ra. Ví dụ, chuỗi lệnh có thể yêu cầu một số dữ liệu từ ổ đĩa hoặc có thể đợi người dùng nhấn một phím. Trong khi chờ đợi, luồng được cho là bị chặn (block) và các luồng khác (nếu có) sẽ có cơ hội chạy. Khi sự kiện xảy ra, một ngắt sẽ "đánh thức" luồng để nó có thể tiếp tục chạy.

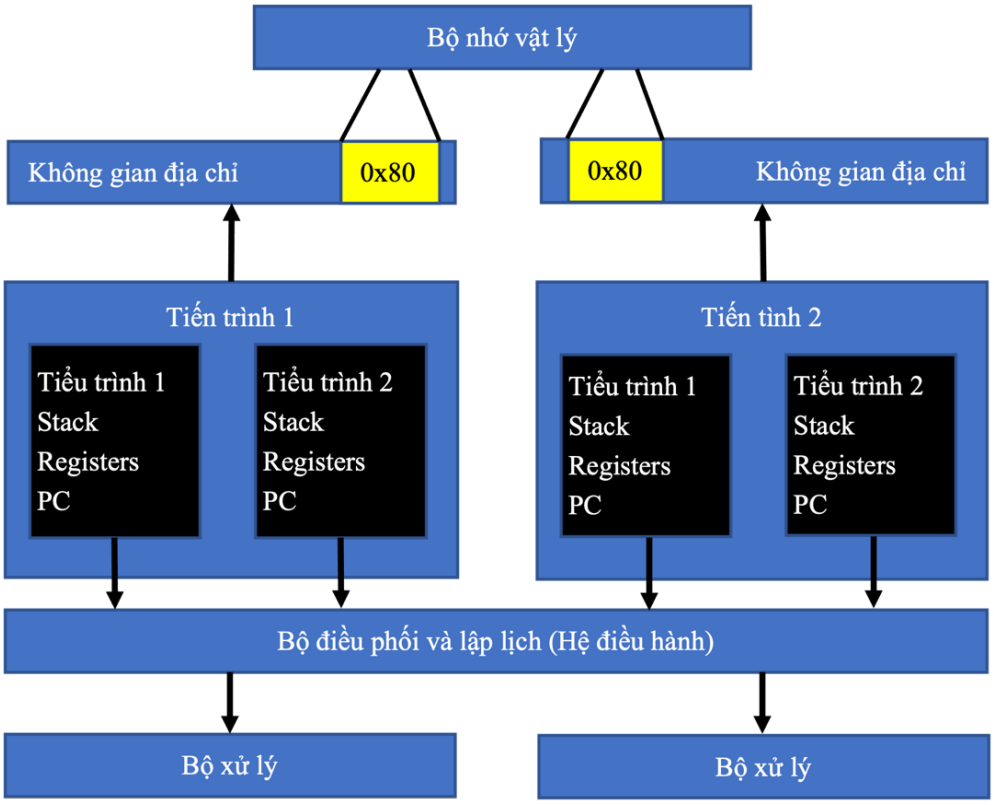
Luồng có thể sử dụng hết phần thời gian được phân bổ của nó và bị tạm ngưng để cho phép các luồng khác chạy. Hầu hết các máy tính có thể "cưỡng bức" đình chỉ một luồng theo cách này.

TIẾN TRÌNH VÀ TIỂU TRÌNH

Tiến trình và tiểu trình là hai khái niệm quan trọng trong lập trình đa luồng. Dưới đây là một số khái niệm cơ bản về chúng:

- Một chương trình **đang được thực thi** thường được gọi là tiến trình.
- Một **tiến trình bao gồm nhiều tiểu trình**. Một tiểu trình là một phần nhỏ nhất của tiến trình có thể thực thi đồng thời
- Một tiến trình có **không gian địa chỉ riêng**. Một tiểu trình **sử dụng không gian địa chỉ của tiến trình** và chia sẻ nó với các tiến trình khác
- Một tiến trình chỉ có thể giao tiếp với tiến trình khác bằng cách sử dụng **IPC**. Một tiểu trình có thể giao tiếp **trực tiếp** với tiểu trình khác.

- Tiểu trình mới được tạo **dễ dàng**. Tuy nhiên, việc tạo ra các tiến trình mới **tốn tài nguyên và thời gian** hơn rất nhiều.
- Tiểu trình có thể **kiểm soát** các tiểu trình khác. Một tiến trình không thể kiểm soát tiến trình anh em, mà chỉ có thể kiểm soát được các tiến trình con.



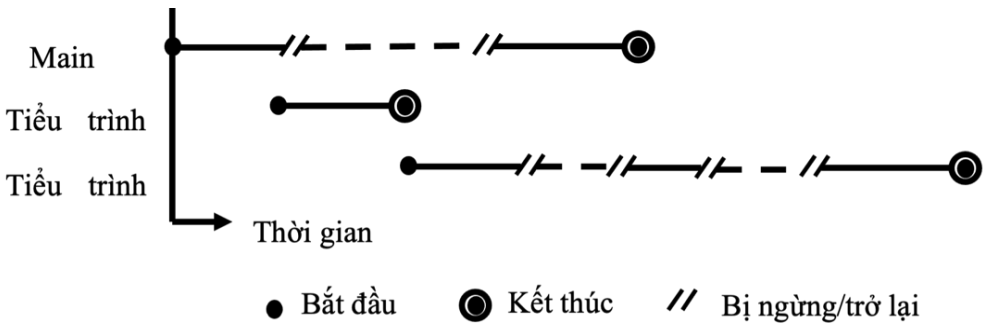
Hình 8-3 Cách thức làm việc của tiến trình và tiểu trình

TIỂU TRÌNH TRONG JAVA

Cơ chế chia sẻ thời gian

Java có hỗ trợ tích hợp cho lập trình đồng thời bằng cách chạy nhiều luồng (threads) đồng thời trong một chương trình duy nhất. Một "thread", còn được gọi là một quy trình nhỏ, là một luồng hoạt động lập trình tuần tự đơn lẻ, với phần đầu và phần cuối xác định. Trong suốt thời gian tồn tại của luồng, chỉ có một điểm thực thi duy nhất. Bản thân một luồng không phải là một chương trình vì nó không thể tự chạy. Thay vào đó, nó chạy trong một chương trình.

Hình sau cho thấy một chương trình có ba luồng chạy dưới một CPU:



Hình 8-4 Cơ chế chia sẻ thời gian

Cách để tạo tiểu trình trong Java

Có hai bước chính để tạo ra một tiểu trình:

- **Bước 1**: Tạo đối tượng Runnable (Bằng Thread Class hoặc Runnable Interface)
- **Bước 2**: Tạo Thread và khởi động (start) Thread đó

Ví dụ: Tạo lớp "RunThread" là kế thừa từ lớp "Thread".

```
1  public class RunThread extends Thread {
2     private String toSay;
3     private int Sleep;
4     public RunThread(String st, int sl) {
5         toSay = st;
6         Sleep = sl;
7     }
8     public void run() {
9         try {
10            for (; ; ) {
11                System.out.println(toSay);
12                Thread.sleep(Sleep);
13            }
14        } catch (InterruptedException e) {
15        }
16    }
17 }
```

Viết phương thức main() trong một class nào đó để hiển thị kết quả:

```
1  public static void main(String[] args) {
2     RunThread thr1 = new RunThread("Thread A", 1200);
3     RunThread thr2 = new RunThread("Thread B", 2000);
4     thr1.start();
5     thr2.start();
6  }
```

Chạy thử và kiểm tra kết quả hiển thị:

```
1  Thread A
2  Thread B
3  Thread A
4  Thread B
5  Thread A
6  Thread A
7  Thread B
8  Thread A
9  ...
```

GIAO DIỆN RUNNABLE

Trong Java, giao diện Runnable được sử dụng để thực hiện đa luồng (multithreading). Đây là một cách để thực hiện mã trong một luồng riêng biệt, đặc biệt là khi bạn muốn thực hiện các hoạt động đồng thời

hoặc nền. Sử dụng Runable cũng khá giống dùng class Thread, tuy nhiên, nó giúp cho lớp vẫn vẫn có thể kế thừa từ class khác nếu muốn.
Sửa lại lớp RunThread triển khai từ giao diện Runnable. Mã lệnh bên trong RunThread vẫn tương tự:

```
1 public class RunThread implements Runnable {
2 ...
3 }
```

Sửa lại cách viết phương thức main như sau

```
1 public static void main(String[] args) {
2    RunThread out1 = new RunThread("Thread A", 2000);
3    RunThread out2 = new RunThread("Thread B", 1200);
4    Thread thr1 = new Thread(out1);
5    Thread thr2 = new Thread(out2);
6    thr1.start();
7    thr2.start();
8 }
```

Chạy thử và kiểm tra kết quả hiển thị:

```
1 Thread A
2 Thread B
3 Thread A
4 Thread B
5 Thread A
6 Thread A
7 Thread B
8 Thread A
9 ...
```

TIỂU TRÌNH VÔ DANH

Trong Java, bạn có thể tạo các **tiểu trình vô danh** bằng cách sử dụng lớp Thread và truyền một đối tượng Runnable vào hàm khởi tạo của nó một cách trực tiếp, thay vì phải tạo một lớp mới để triển khai giao diện Runnable. Điều này giúp rút ngắn mã và làm cho mã của bạn trở nên gọn gàng hơn. Dưới đây là một ví dụ về cách tạo tiểu trình vô danh:

```
1 public static void main(String[] args) {
2    new Thread(() -> {
3        System.out.println("Bắt đầu");
4        System.out.println("Thực hiện công veiec");
5        System.out.println("Kết thúc");
6    }).start();
7 }
```

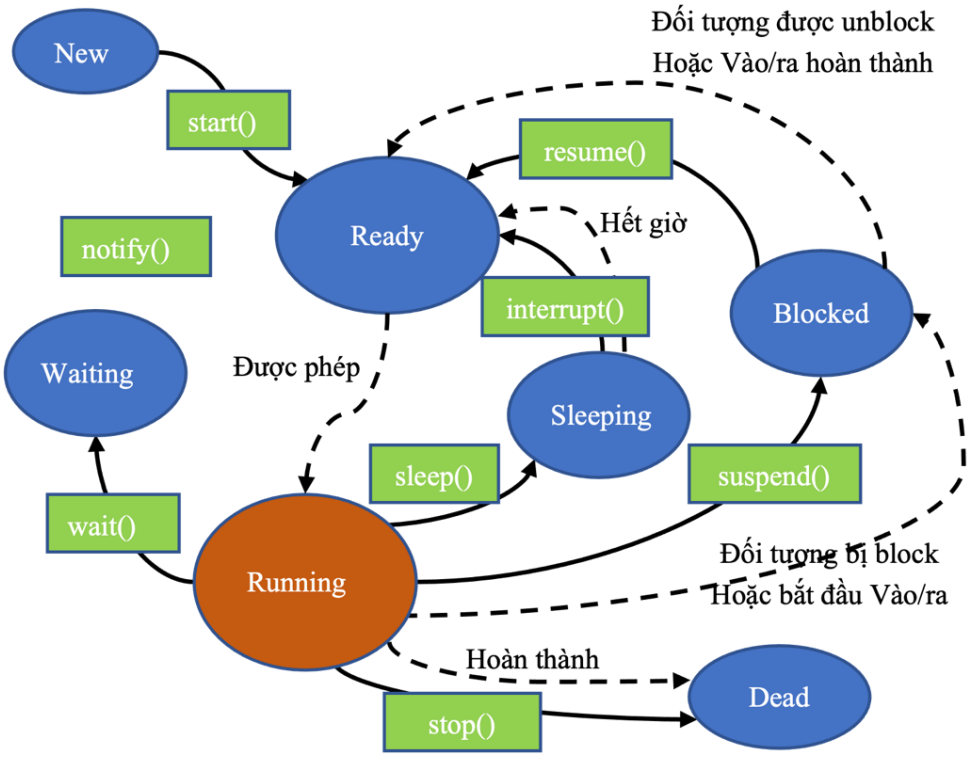
TRẠNG THÁI CỦA TIỂU TRÌNH

Bảy trạng thái của tiểu trìnhMột tiểu trình được tao ra có thể tồn tại ở một trong bảy trạng thái sau:

- **new**: vừa được tạo nhưng chưa bắt đầu

- **ready**: được tạo, bắt đầu và có thể chạy
- **running**: đang được thực thi (bởi CPU)
- **waiting**: chờ đợi sự kiện nào đó xảy ra
- **sleeping**: không làm gì trong một thời gian cụ thể
- **blocked**: bị chặn hoạt động bởi khóa hoặc chờ một hoạt động IO
- **dead**: chuỗi đã kết thúc hoặc bị dừng

Vòng đời của một tiểu trình thể hiện trong sơ đồ sau:



Hình 8-5 Vòng đời của một tiểu trình

XỬ LÝ ĐỒNG BỘ

Xử lý đồng bộ luồng trong Java là quá trình đảm bảo rằng các tiểu trình không xâm nhập vào nhau khi truy cập vào tài nguyên chia sẻ. Điều này là cần thiết để tránh tình trạng đọc/ghi không đồng bộ có thể dẫn đến lỗi và dữ liệu không đúng đắn. Trong Java có một số cách để xử lý đồng bộ

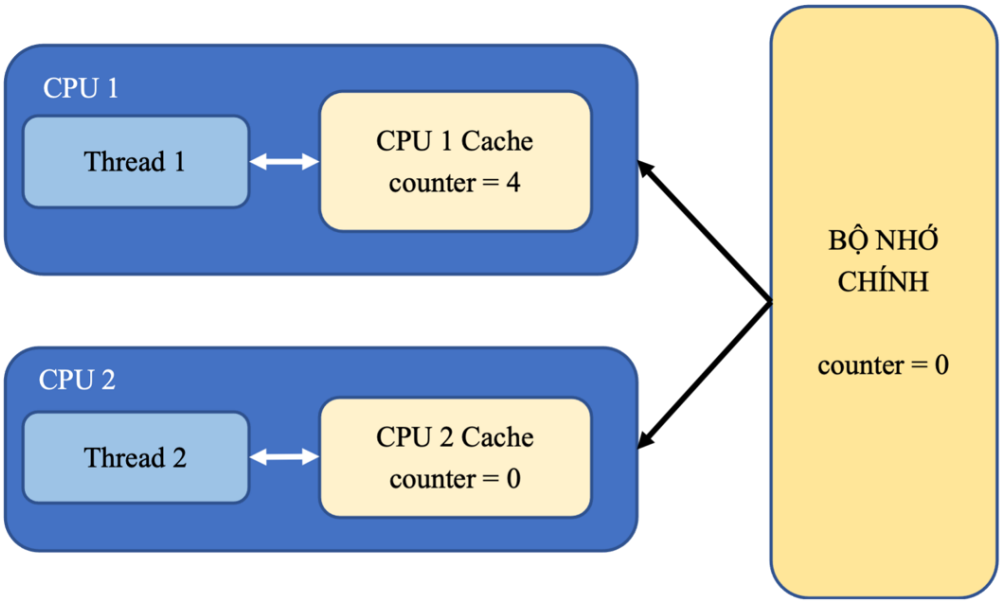
- Sử dụng từ khoá **synchronized**
- Sử dụng từ khoá **volatile**
- Sử dụng **lock**

Trong Java, từ khóa **synchronized** được sử dụng để đồng bộ hóa các phương thức hoặc khối mã để đảm bảo rằng chỉ một luồng có thể truy cập vào một phần của mã tại một thời điểm. Điều này giúp tránh các vấn đề liên quan đến đồng thời và cung cấp tính nhất quán cho ứng dụng đa luồng.

Ví dụ:

```
public class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
    public synchronized void decrement() {
        count--;
    }
}
```

Từ khóa **volatile** cũng nên được sử dụng để đảm bảo dữ liệu của biến được tải từ bộ nhớ chứ không phải từ bộ nhớ đệm cache của CPU.



Hình 8-6 Dữ liệu từ các biến có thể được tải từ bộ nhớ đệm của các CPU

Ví dụ:

```
public class SharedObject {
    private volatile int value;
}
```

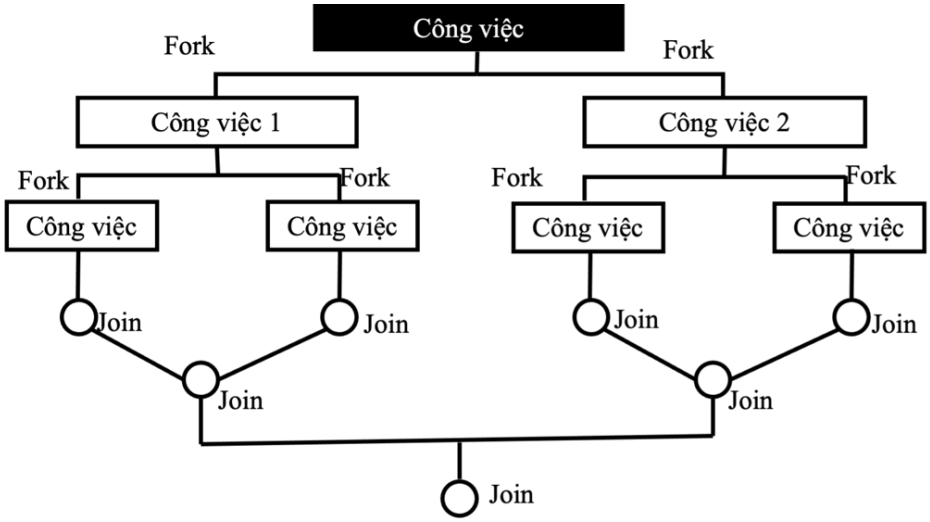
Ngoài ra, Java cung cấp gói java.util.concurrent.locks để quản lý đồng bộ hóa bằng cách sử dụng lock.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class Counter {
    private int count = 0;
    private Lock lock = new ReentrantLock();
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
    public void decrement() {
```

```
15          lock.lock();
16          try {
17              count--;
18          } finally {
19              lock.unlock();
20          }
21      }
22  }
```

KHUNG LÀM VIỆC FORK/JOIN

Khung làm việc **fork/join** cung cấp các công cụ để giúp tăng tốc xử lý song song bằng cách cố gắng sử dụng tối đa khả năng hoạt động của CPU, cách thức làm việc dựa trên cơ chế chia để trị:



Hình 8-7 Khung làm việc fork/join

Một ví dụ sử dụng khung làm việc folk/join:

```
 1  public class ForkJoinTest {
 2      public static void main(String[] args) {
 3          final int SIZE = 100000000;  //100M
 4          var numbers = new double[SIZE];
 5          for (int i = 0; i < SIZE; i++)
 6              numbers[i] = Math.random();
 7          var counter = new Counter(numbers, 0, numbers.length);
 8          var pool = new ForkJoinPool();
 9          pool.invoke(counter);
10          System.out.println(counter.join());
11      }
12  }
13  class Counter extends RecursiveTask<Integer> {
14      private double[] values;
15      private int from;
16      private int to;
17      public Counter(double[] values, int from, int to) {
18          this.values = values;
19          this.from = from;
20          this.to = to;
21      }
22      protected Integer compute() {
```

```
        if (to - from < 1000) {
            int count = 0;
            for (int i = from; i < to; i++)
                if (values[i]>0.5) count++;
            return count;
        } else {
            int mid = (from + to) / 2;
            var first = new Counter(values, from, mid);
            var second = new Counter(values, mid, to);
            invokeAll(first, second);
            return first.join() + second.join();
        }
    }
}
```

TIẾN TRÌNH

Java có thể tạo tiến trình mới bằng cách sử dụng các lớp Process và ProcessBuilder. Việc tạo tiến trình có thể khởi động một ứng dụng bất kỳ.

Ví dụ đoạn mã sau tạo tiến trình và chạy lệnh duyệt thư mục. /bin/ls."

```
public static void main(String[] args) throws IOException, InterruptedException {
    Process p = new ProcessBuilder("/bin/ls", "-l")
            .directory(Paths.get("/tmp").toFile())
            .start();
    try (var in = new Scanner(p.getInputStream())) {
        while (in.hasNextLine())
            System.out.println(in.nextLine());
    }
}
```

THỰC HÀNH

Join tiểu trình

Tạo lớp “MyThread” triển khai từ giao diện Runnable:

```
class MyThread implements Runnable {
    String name; // name of thread
    Thread t;
    MyThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
```

```
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
```

Viết phương thức main() trong một class nào đó để tạo và khởi động ba thread:

```
public static void main(String[] args) {
   MyThread ob1 = new MyThread("One");
   MyThread ob2 = new MyThread("Two");
   MyThread ob3 = new MyThread("Three");
   System.out.println("Thread One sống: " + ob1.t.isAlive());
   System.out.println("Thread Two sống: " + ob2.t.isAlive());
   System.out.println("Thread Three sống: " + ob3.t.isAlive());
   System.out.println("Main thread kết thúc");
}
```

Chạy thử và kiểm tra kết quả

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One sống: true
Thread Two sống: true
Thread Three sống: true
Thread One sống: true
Thread Two sống: true
Thread Three sống: true
Main thread kết thúc
Three: 5
One: 5
Two: 5
Three: 4
Two: 4
One: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
Two: 1
One: 1
Three: 1
One exiting.
Two exiting.
Three exiting.
```

Sửa lại phương thức main như sau, bổ sung thêm join()

```
public static void main(String[] args) {
   MyThread ob1 = new MyThread("One");
   MyThread ob2 = new MyThread("Two");
   MyThread ob3 = new MyThread("Three");
   System.out.println("Thread One is alive: " + ob1.t.isAlive());
   System.out.println("Thread Two is alive: " + ob2.t.isAlive());
   System.out.println("Thread Three is alive: " + ob3.t.isAlive());
   try {
       System.out.println("Waiting for threads to finish.");
       ob1.t.join();
       ob2.t.join();
       ob3.t.join();
   } catch (InterruptedException e) {
       System.out.println("Main thread Interrupted");
   }
   System.out.println("Thread One is alive: " + ob1.t.isAlive());
   System.out.println("Thread Two is alive: " + ob2.t.isAlive());
   System.out.println("Thread Three is alive: " + ob3.t.isAlive());
   System.out.println("Main thread exiting.");
}
```

Chạy thử và kiểm tra kết quả. Giải thích kết quả in ra.

Bài toán số nguyên tố

Chương trình trong bài tập này sẽ viết chương trình sử dụng nhiều thread để đếm số nguyên tố từ 0 đến 1000000. Số lượng luồng cần thiết để tìm kiếm các số nguyên tố sẽ được quyết định bởi người sử dụng.

Tạo lớp “CountPrimesThread” là mở rộng của lớp “Thread”. Khai báo các biến:

```
//Variable to store the number of primes in a specified range of
static int count = 0;
// Variable to store the range of minimum and maximum integer value
int min, max;
```

Viết các phương thức tạo:

```
public CountPrimesThread(int min, int max) {
   this.min = min;
   this.max = max;
}
```

Phương thức isPrime () được gọi để kiểm tra x có phải là số nguyên tố hay không.

```
private static boolean isPrime(int x) {
   int top = (int) Math.sqrt(x);
   for (int i = 2; i <= top; i++)
       if (x % i == 0)
           return false;
   return true;
}
```

Phương thức countPrimes() được gọi để đếm các số nguyên tố giữa min và max:

```
private static void countPrimes(int min, int max) {
    for (int i = min; i <= max; i++)
        if (isPrime(i))
            count++;
}
```

Phương thức Run() được gọi để xuất ra một thông báo về số lượng các số nguyên tố mà nó đã tìm thấy

```
public void run() {
    countPrimes(min, max);
    System.out.println("There are " + count + " primes between " + "" + min + " and " + max);
}
```

Viết phương thức CountPrimesWithThreads() để tạo luồng đếm số nguyên tố

```
public static void countPrimesWithThreads(int numberOfThreads) {
    int start = 0;
    int to = 1000000;
    int increment = to / numberOfThreads;
    System.out.println("\nCounting primes between " + (start + 1) + " and " + to + " using " + numberOfThreads + " threads. " + "Please wait...\n");
    long startTime = System.currentTimeMillis();
    CountPrimesThread[] worker = new CountPrimesThread[numberOfThreads];
    for (int i = 0; i < numberOfThreads; i++) worker[i] = new CountPrimesThread(start + i * increment + 1, start + (i + 1) * increment);
    for (int i = 0; i < numberOfThreads; i++) worker[i].start();
    for (int i = 0; i < numberOfThreads; i++) {
        while (worker[i].isAlive()) {
            try {
                worker[i].join();
            } catch (InterruptedException e) {
            }
        }
    }
    long elapsedTime = System.currentTimeMillis() - startTime;
    System.out.println("\nTotal elapsed time: " + (elapsedTime / 1000.0) + "seconds.\n");
}
```

Viết chương trình cho phương thức main():

```
public static void main(String[] args) {
    int numberOfThreads = 0;
    do {
        System.out.println("How many threads do you want to use (from 1 " + "to 10)?");
        Scanner input = new Scanner(System.in);
        numberOfThreads = input.nextInt();
        if (numberOfThreads < 1 || numberOfThreads > 10)
```

```
            System.out.println("Please enter 1  .. 10!");
    }
    while (numberOfThreads < 1 || numberOfThreads > 10);
    countPrimesWithThreads(numberOfThreads);
}
```

Thực hiện chương trình và kiểm tra với số lượng thread khác nhau thì tốc độ tính toán thay đổi thế nào. Hãy cho biết khi nào thời gian được cải thiện khi tăng số lượng thread và khi nào thì thời gian tính toán không được cải thiện?

Bài toán ngân hàng

Tạo class Bank

```
import java.util.Arrays;
public class Bank {
    public final double[] accounts;
    public Bank(int n, double initialBalance) {
        accounts = new double[n];
        Arrays.fill(accounts, initialBalance);
    }
    public void transfer(int from, int to, double amount) {
        System.out.print(Thread.currentThread().getName());
        accounts[from] -= amount;
        System.out.printf(" %10.2f from %d to %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
    }
    public double getTotalBalance() {
        double sum = 0;
        for (double a : accounts)
            sum += a;
        return sum;
    }
}
```

Tạo class Main

```
public class Main {
    public static final int NACCOUNTS = 3;
    public static final double INITIAL_BALANCE = 1000;
    public static final double MAX_AMOUNT = 1000;
    public static void main(String[] args) throws InterruptedException {
        Bank bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
        while (true) {
            int fromAccount = (int) (bank.accounts.length * Math.random());
            int toAccount = (int) (bank.accounts.length * Math.random());
            double amount = MAX_AMOUNT * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep(10);
        }
```

```
14      }
15  }
```

Chạy thử và xem kết quả, giả sử muốn nhận input từ bàn phím thì không thể.

```
1  main     407.68 from 0 to 0 Total Balance:    3000.00
2  main     507.47 from 0 to 0 Total Balance:    3000.00
3  main      78.25 from 0 to 1 Total Balance:    3000.00
4  main     165.24 from 0 to 1 Total Balance:    3000.00
5  main     356.55 from 1 to 0 Total Balance:    3000.00
6  main      59.27 from 2 to 2 Total Balance:    3000.00
7  main     626.90 from 0 to 0 Total Balance:    3000.00
8  main     758.75 from 0 to 0 Total Balance:    3000.00
```

Bây giờ sẽ chỉnh sửa lại bài toán trên trong đó có sử dụng Thread. Đầu tiên, tạo ra một class có tên ThreadTest và đưa phần giao dịch vào một thread:

```
1  public class ThreadTest {
2      public static final int NACCOUNTS = 3;
3      public static final double INITIAL_BALANCE = 1000;
4      public static final double MAX_AMOUNT = 1000;
5      public static void main(String[] args) {
6          Bank bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
7          Runnable runnable = () -> {
8              while (true) {
9                  int fromAccount = (int) (bank.accounts.length * Math.random());
10                 int toAccount = (int) (bank.accounts.length * Math.random());
11                 double amount = MAX_AMOUNT * Math.random();
12                 bank.transfer(fromAccount, toAccount, amount);
13             }
14         };
15         new Thread(runnable).start();
16     }
17 }
```

Chạy thử và xem kết quả

```
1  Thread-0     251.53 from 1 to 0 Total Balance:    3000.00
2  Thread-0     642.04 from 0 to 1 Total Balance:    3000.00
3  Thread-0     308.59 from 1 to 1 Total Balance:    3000.00
4  Thread-0     484.27 from 2 to 1 Total Balance:    3000.00
5  Thread-0     462.66 from 2 to 0 Total Balance:    3000.00
6  Thread-0     263.15 from 1 to 0 Total Balance:    3000.00
7  Thread-0     904.53 from 1 to 1 Total Balance:    3000.00
8  Thread-0      70.65 from 2 to 0 Total Balance:    3000.00
```

Cho tất cả các Thread chạy

```
1  public static void main(String[] args) {
2      Bank bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
3      for (int i = 0; i < NACCOUNTS; i++) {
4          int fromAccount = i;
```

```
        Runnable runnable = () -> {
            while (true) {
                int toAccount = (int) (bank.accounts.length * Math.random());
                double amount = MAX_AMOUNT * Math.random();
                bank.transfer(fromAccount, toAccount, amount);
            }
        };
        new Thread(runnable).start();
    }
}
```

Chạy thử và xem kết quả:

```
Thread-0     263.14 from 1 to 2 Total Balance:    2356.72
Thread-0     640.00 from 2 to 1 Total Balance:    2543.76
Thread-2     454.85 from 2 to 2 Total Balance:    2543.76
Thread-2     452.96 from 1 to 0 Total Balance:    2963.40
Thread-0     443.90 from 1 to 1 Total Balance:    2963.40
Thread-0     411.98 from 1 to 2 Total Balance:    2963.40
Thread-0     272.90 from 0 to 0 Total Balance:    2963.40
Thread-0       3.28 from 1 to 1 Total Balance:    2752.24
Thread-1     606.72 from 0 to 0 Total Balance:    2752.24
Thread-1     129.45 from 0 to 0 Total Balance:    2752.24
Thread-1     173.33 from 1 to 2 Total Balance:    2752.24
Thread-1     214.44 from 0 to 2 Total Balance:    2668.70
Thread-0     726.02 from 2 to 0 Total Balance:    2668.70
Thread-0     701.75 from 0 to 1 Total Balance:    2668.70
```

Tổng amount bị sai do trừ, in ra nhưng chưa cộng cho account dưới, khắc phục bằng cách **sử dụng SyncThread**

Sửa lại Bank class bổ sung thuộc tính lock:

```
private Lock lock;
```

Trong phương thức tạo của Bank khởi tạo 1 ReentrantLock

```
public Bank(int n, double initialBalance) {
    accounts = new double[n];
    Arrays.fill(accounts, initialBalance);
    lock = new ReentrantLock();
}
```

Khi transfer thì khóa và mở khóa sau khi transfer xong

```
public void transfer(int from, int to, double amount) {
    lock.lock();
    System.out.print(Thread.currentThread().getName());
    accounts[from] -= amount;
    System.out.printf(" %10.2f from %d to %d", amount, from, to);
    accounts[to] += amount;
    System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
```

```
    lock.unlock();
}
```

Sử dụng try finally để đảm bảo khóa được mở

```
public void transfer(int from, int to, double amount) {
    lock.lock();
    try {
        System.out.print(Thread.currentThread().getName());
        accounts[from] -= amount;
        System.out.printf(" %10.2f from %d to %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
    } finally {
        lock.unlock();
    }
}
```

Nhằm giải quyết vấn đề về xung đột, cần sửa lại để khiến chương trình hoạt động đồng bộ. Đầu tiên, trong class bank bổ sung thêm đối tượng điều kiện:

```
private Condition condition;
```

Khởi tạo đối tượng điều kiện trong phương thức tạo Bank

```
public Bank(int n, double initialBalance) {
    accounts = new double[n];
    Arrays.fill(accounts, initialBalance);
    lock = new ReentrantLock();
    condition = lock.newCondition(); //Bổ sung
}
```

Trong giao dịch (hàm transfer) thay việc kiểm tra:

```
if (accounts[from] < amount) return;
```

Bằng việc kiểm tra như sau:

```
while (accounts[from] < amount) {
    try {
        System.out.println(Thread.currentThread().getName() + "  " + from + " is not
enough");
        condition.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Sau khi giao dịch xong thì gọi thêm phương thức signalAll()

```
condition.signalAll();
```

Khi đó phương thức transfer sẽ như sau:

```
public void transfer(int from, int to, double amount) {
    lock.lock();
    try {
        while (accounts[from] < amount) {
            try {
                System.out.println(Thread.currentThread().getName() + "  " + from + " is not enough");
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.print(Thread.currentThread().getName());
        accounts[from] -= amount;
        System.out.printf(" %10.2f from %d to %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
        condition.signalAll();
    } finally {
        lock.unlock();
    }
}
```

Cách thứ 2 để giải quyết vấn đề đồng bộ là sử dụng **synchronized** cùng với **wait** và **notifyAll.**

Xóa các lock và điều kiện, sử dụng synchronized wait notifyAll ở phương thức transfer

```
public synchronized void transfer(int from, int to, double amount) {
    while (accounts[from] < amount) {
        try {
            System.out.println(Thread.currentThread().getName() + "  " + from + " is not enough");
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.print(Thread.currentThread().getName());
    accounts[from] -= amount;
    System.out.printf(" %10.2f from %d to %d", amount, from, to);
    accounts[to] += amount;
    System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
    notifyAll();
}
```

Sửa lại MAX_AMOUNT = 1500

Chạy lại chương trình sẽ thấy deadlock xảy ra:

```
 1  Thread-0  0 is not enough
 2  Thread-2  2 is not enough
 3  Thread-1     907.57 from 1 to 2 Total Balance:    3000.00
 4  Thread-0  0 is not enough
 5  Thread-1  1 is not enough
 6  Thread-2    1078.15 from 2 to 1 Total Balance:    3000.00
 7  Thread-2  2 is not enough
 8  Thread-0  0 is not enough
 9  Thread-1     142.17 from 1 to 0 Total Balance:    3000.00
10  Thread-1     120.96 from 1 to 2 Total Balance:    3000.00
11  Thread-1  1 is not enough
12  Thread-2  2 is not enough
13  Thread-0  0 is not enough
```

Hãy thử suy nghĩ để khử Deadlock xảy ra trong trường hợp trên.

Sử dụng Thread Pool

Tạo một Thread để giả lập việc tải file:

```
 1  public class DownloadFile implements Runnable {
 2     @Override
 3     public void run() {
 4         int timetoFinish = (int) (5000 * Math.random());
 5         System.out.println(Thread.currentThread().getName() + " Bắt đầu tải một file
mới. Cần" + timetoFinish + "ms để hoàn thành");
 6         try {
 7             Thread.sleep(timetoFinish);
 8         } catch (InterruptedException e) {
 9             e.printStackTrace();
10         }
11         System.out.println(Thread.currentThread().getName() + " Download Hoàn thàn
h");
12     }
13  }
```

Tạo Thread Pool bằng cách viết trong phương thức main như sau:

```
 1  public static void main(String[] args) {
 2     ExecutorService executor = Executors.newFixedThreadPool(5);
 3     for (int i = 0; i < 10; i++) {
 4         Runnable worker = new DownloadFile();
 5         executor.execute(worker);
 6     }
 7  }
```

Chạy thử, và giải thích cách thức làm việc của chương trình

CÂU HỎI ÔN TẬP LÝ THUYẾT

1. Tiểu trình khác gì tiến trình:

- A. Tiểu trình dùng riêng một không gian bộ nhớ
- B. Tiểu trình dùng chung không gian bộ nhớ
- C. Tiểu trình chứa nhiều tiến trình bên trong
- D. Tiểu trình chạy chậm hơn tiến trình

2. Tiểu trình được tạo ra bằng cách:

- A. Khai báo lớp triển khai từ giao diện Runable
- B. Khai báo lớp kế thừa từ lớp Thread
- C. Sử dụng cách tạo tiểu trình vô danh
- D. Các đáp án trên đều đúng

3. Tiểu trình ở trạng thái running khi:

- A. Nó đang chạy và phục vụ bởi CPU
- B. Nó đang chờ đợi một tiểu trình khác
- C. Nó đang bị khóa hoặc đang đợi một hoạt động vào ra
- D. Các đáp án trên đều dúng

4. Phương thức join được dùng để:

- A. Thông báo các tiểu trình dừng chờ đợi tiếp và tục làm việc
- B. Kiểm tra xem một tiến trình còn sống (alive) hay không
- C. Chờ một tiến trình khác giải phóng (release) tài nguyên
- D. Chờ một tiến trình khác kế thúc

5. Từ khóa synchronized được dùng để:

- A. Giải quyết vấn đề về bất đồng bộ
- B. Đồng bộ hóa việc truy cập dữ liệu
- C. Đảm bảo một luồng duy nhất thực hiện một khối lệnh
- D .Giải quyết vấn đề về deadlock

6. Để khởi động một tiểu trình dùng phương thức:

- A. run()
- B. action()
- C. ready()
- D. start()

7. Một trong những mục đích của lập trình đa luồng là:

- A. Tăng tốc độ tính toán
- B. Tận dụng được tối đa tài nguyên của máy tính
- C. Thực hiện nhiều công việc một lúc
- D. Tất cả các đáp án đều đúng

8. Nhược điểm của đa luồng là:

- A.Việc xử lý phức tạp
- B. Tốc độ bị giảm
- C. Các công việc bị gián đoạn vì sử dụng chia sẻ thời gian
- D. Tất cả các đáp án đều sai

9. Để tạo ra một luồng, khi nào thì triển khai từ giao diện Runable thay vì kế thừa từ lớp Thread:

- A. Khi muốn viết mã ngắn hơn
- B. Khi muốn tạo ra nhiều luồng hơn
- C. Khi lớp chạy vẫn muốn kế thừa từ lớp khác
- D. Khi muốn mã lệnh được rõ ràng, tường minh

10. Tiểu trình sẽ chuyển từ trạng thái new sang trạng thái running khi:

- A. Hệ điều hành cho phép tiểu trình hoạt động
- B. Gọi hàm start()
- C. Gọi hàm run()
- D. Ngay sau khi tiểu trình được tạo ra

11. Hãy cho biết số luồng sẽ thay đổi và in giá trị của biến i:

```
1  class CountAndPrint implements Runnable {
2      private final String name;
3
4      CountAndPrint(String name) {
5          this.name = name;
6      }
7
8      static int i = 0;
9
10     @Override
11     public void run() {
12         while (i < 10) System.out.println(this.name + ": " + i++);
13     }
14
15     public static void main(String[] args) {
16         for (int t = 1; t <= 4; t++)
17             new Thread(new CountAndPrint("Instance " + t)).start();
18         while (i < 10) System.out.println("Main: " + i++);
19     }
20 }
```

- A. 4
- B. 5
- C. 9
- D. 10

BÀI TẬP TỰ THỰC HÀNH

Tính tổng các phần tử của một mảng

Viết một chương trình Java sử dụng đa luồng để tính tổng các phần tử của một mảng số nguyên. Mảng số nguyên có kích thước lớn hơn 1000 phần tử.

Tính tổng các phần tử của một mảng

Viết một chương trình Java tạo ra 2 luồng, mỗi luồng in ra chẵn và một luồng. Luồng in số chẵn phải bắt đầu trước luồng in số lẻ.

Tính tổng sử dụng Fork/Join

Viết một chương trình tính tổng của một mảng số nguyên bằng cách sử dụng Fork/Join. Sử dụng thuật toán chia để trị (divide and conquer) để tách mảng thành các phần nhỏ và tính tổng của chúng. Thử nghiệm chương trình với một mảng số nguyên có 100 triệu phần tử.

Tính tổng mảng bằng ThreadPoolExecutor

Tạo một ThreadPoolExecutor để tính tổng các phần tử trong một mảng sử dụng nhiều luồng.

TÀI LIỆU THAM KHẢO

[1] Core Java: Fundamentals (2021) , Cay Horstmann (Oracle Press Java)

[2] Head First Java: A Brain-Friendly Guide (2022), Kathy Sierra, O'Reilly Media

[3] Java OOP Done Right: Create object oriented code you can be proud of with modern Java Paperback (2019), Mr Alan Mellor, Mellor Books

[4] Murach's Java Programming (5th Edition) (2017), Joe Murach, Mike Murach & Associates

[5]. Java for Absolute Beginners Learn to Program the Fundamentals the Java 9+ Way

[6]. Modern Java Recipes: Simple Solutions to Difficult Problems in Java 8 and 9 (2017), by Ken Kousen, O'Reilly Media

[7] Effective Java (2018), Joshua Bloch, Addison-Wesley Professional