**1. Process và Thread trong Java**

Đầu tiên khi nói về lập trình đa luồng, chúng ta cần phân biệt được 2 khái niệm Process (tiến trình) và Thread (luồng) trong Java:

|  | **Process** | **Thread** |
| --- | --- | --- |
| **Khái niệm** | Một chương trình đang chạy được gọi là một **process**. | Một chương trình chạy có thể có nhiều **thread**, Cho phép chương trình đó chạy trên nhiều luồng một cách "đồng thời". |
| **Không gian địa chỉ** | Mỗi **process** có một không gian địa chỉ riêng biệt. | Tất cả **thread** thuộc một **process** chia sẻ không gian địa chỉ với nhau, hợp chúng lại thành một tiến trình. |
| **Đa nhiệm** | Đa nhiệm dựa trên process cho phép máy tính chạy 2 hoặc nhiều hơn 2 chương trình đồng thời. | Đa nhiệm dựa trên thread cho phép một chương trình chạy trên 2 hoặc nhiều luồng đồng thời. |
| **Giao tiếp** | Giao tiếp giữa 2 tiến trình là tốn kém và bị giới hạn. | Giao tiếp giữa 2 thread ít tốn kém hơn so với tiến trình. |
| **Thành phần** | Một tiến trình có: không gian địa chỉ, biến global, xử lí tín hiệu, những tiến trình con, thông tin tính toán. | Một thread có: thanh ghi, trạng thái, stack, bộ đếm chương trình. |
| **Điều khiển** | Đa nhiệm dựa trên process không thuộc quyền kiểm soát của Java. | Đa nhiệm dựa trên thread phụ thuộc quyền kiểm soát của Java. |
| **Ví dụ** | Khi chạy một ứng dụng Java thì đó được gọi là một tiến trình. | Một ứng dụng đếm từ trong 1000 file, có sử dụng 4 luồng để chạy đồng thời. |

**2. Concurrency và Parallelism**

Trong multi-threading có tồn tại 2 khái niệm là Concurrency (đồng thời) và Parallelism (song song). Thoạt đầu thì nghĩ chúng có vẻ giống nhau, nhưng chúng không phải là một đâu nhé.

**Concurrency** nghĩa là những tác vụ có thể bắt đầu, chạy, và hoàn thành trong những khoảng thời gian chống chéo lên nhau mà không theo thứ tự nào cả. Ví dụ: chạy trên 1 core processor. Còn **Parallelism** là nhiều tác vụ hoặc một phần của tác vụ chạy đồng thời tại cùng một thời điểm. Ví dụ: chạy trên multi-core processors.

**Concurrency** thực sự được ứng dụng khi chúng ta có ít nhất 2 tác vụ trở lên. Khi mà ứng dụng có thể thực hiện 2 task gần như cùng một thời điểm, chúng ta gọi đây là Concurent Application. Có thể chúng ta nhìn thấy nó gần như là thực hiện cùng 1 thời gian nhưng có vẻ không phải như vậy. Chúng tận dụng lợi thế của CPU time-slicing (Chia cắt thời gian) của hệ điều hành, mỗi tác vụ thực hiện nhiệm vụ của nó và chuyển sang trạng thái chờ. Khi tác vụ đầu tiên ở trạng thái chờ, CPU được gán cho tác vụ thứ 2 thực hiện nhiệm vụ của nó. Hệ điều hành dựa trên độ ưu tiên cũng từn task, sẽ phân chia CPU và tài nguyên để tính toán. Đối với người sử dụng, thấy dường như là các task được chạy một cách song song.

**Parallelism** không yêu cầu 2 task trở lên để tồn tại. Nó chạy một phần của task hoặc multi-task tại cùng 1 thời gian, dựa vào cấu trúc mult-core của CPU, bằng cách phân chia mỗi core của CPU thực hiện 1 task hoặc 1 sub-task. **Parallelism** yêu cầu phần cứng phải có nhiều đơn vị xử lí. Đối với CPU có 1 core, bạn có thể thực hiện **Concurrency** nhưng không thể là **Parallelism**.

**3. Code ví dụ**

**a. Sử dụng Thread và Runnable**

Tạo một implement của Runnable:

public class MyRunnable implements Runnable {

@Override

public void run() {

try {

String threadName = Thread.currentThread().getName();

System.out.println("Running in " + threadName);

Thread.sleep(5000);

System.out.println("Finish in " + threadName);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Sử dụng Thread để khởi chạy:

private static void runThread() {

MyRunnable runnable1 = new MyRunnable();

MyRunnable runnable2 = new MyRunnable();

Thread thread1 = new Thread(runnable1);

Thread thread2 = new Thread(runnable2);

Arrays.asList(thread1, thread2).parallelStream().forEach(Thread::start);

}

**b. Sử dụng ExecutorService**

ExecutorService là một Java api thuộc package java.util.concurrent. Đầu tiên, chúng ta tạo một implement của Callable:

public class MyCallable implements Callable<String> {

@Override

public String call() {

String threadName = Thread.currentThread().getName();

System.out.println("Running in " + threadName);

try {

Thread.sleep(5000);

return threadName;

} catch (InterruptedException e) {

e.printStackTrace();

return null;

}

}

}

Sử dụng ExecutorService để khởi chạy:

private static void runConcurrency() throws InterruptedException {

ExecutorService executor = Executors.newFixedThreadPool(4);

MyCallable callable1 = new MyCallable();

MyCallable callable2 = new MyCallable();

List<Future<String>> futures = executor.invokeAll(Arrays.asList(callable1, callable2));

futures.parallelStream().forEach(future -> {

try {

String result = future.get();

System.out.println("Finish in " + result);

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

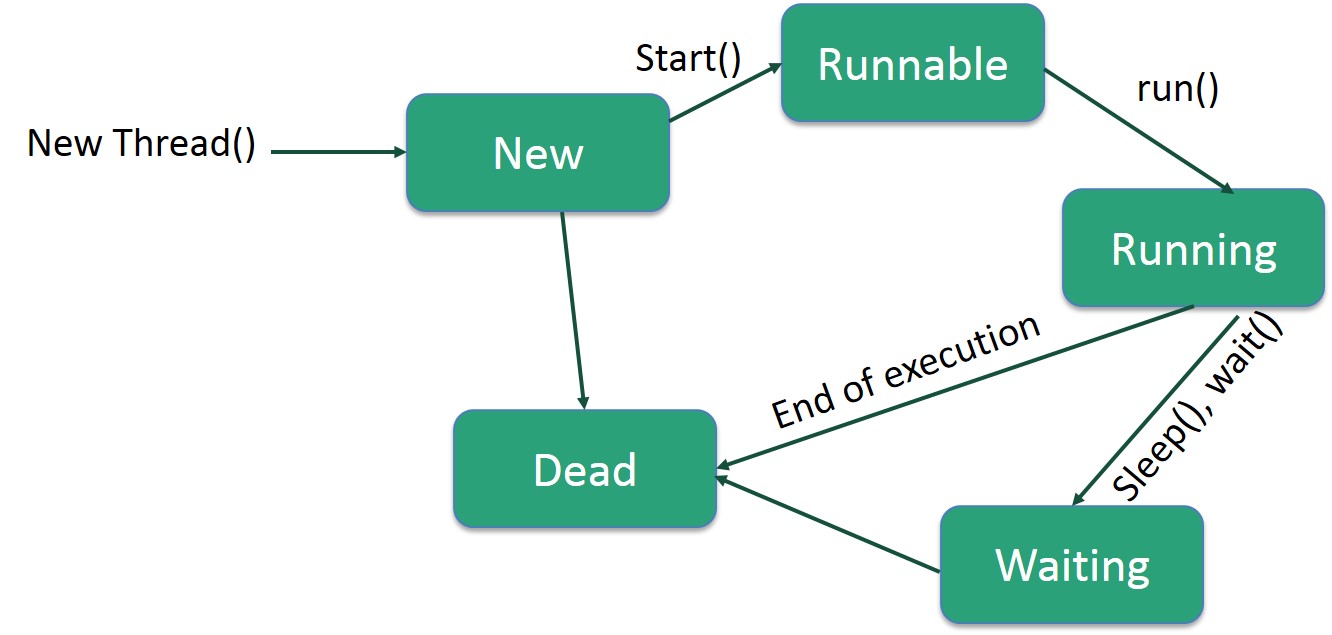
});

executor.shutdown();

}

## 1. Vòng đời của Thread

Một thread đi qua các giai đoạn khác nhau trong vòng đời của nó. Ví dụ, một thread được sinh ra, được bắt đầu, chạy và sau đó hủy. Sơ đồ sau biểu diễn một vòng đời đầy đủ của một thread.



Ta có thể giải thích vòng đời trên như sau:

1. **New:** Một thread mới bắt đầu vòng đời của nó trong trạng thái new. Nó tồn tại trong trạng thái này tới khi chương trình bắt đầu thread này. Nó cũng được xem như là một thread mới sinh.
2. **Runnable:** Sau khi một thread mới sinh ra được bắt đầu, thread trở thành runnable. Một thread trong trạng thái này được xem như đang thực hiện tác vụ của nó.
3. **Waiting:** Đôi khi, một thread quá độ qua trạng thái waiting trong khi thread đợi cho thread khác thực hiện một tác vụ. Một thread chuyển về trạng thái runnable chỉ khi thread khác ra hiệu cho thread đang đợi để tiếp tục thực thi.
4. **Timed waiting:** Một thread trong trạng thái runnable có thể đi vào trạng thái timed waiting trong một khoảng thời gian nào đó. Một thread trong trạng thái này chuyển về trạng thái runnable khi khoảng thời gian đó kết thúc hoặc khi sự kiện nó đang đợi xuất hiện.
5. **Terminated:**Một thread trong trạng thái runnable có thể đi vào trạng thái terminated khi nó hoàn thành tác vụ của nó hoặc nó chấm dứt.

## 2. Quyền ưu tiên của Thread

Mỗi thread trong Java có một quyền ưu tiên mà giúp hệ điều hành xác định thứ tự thread nào được ghi lịch trình.

Quyền ưu tiên của thread trong Java là một dãy giữa MIN\_PRIORITY (hằng số 1) và MAX\_PRIORITY (hằng số 10). Theo mặc định, mỗi thread được cung cấp một quyền ưu tiên NORM\_ PRIORITY (hằng số 5).

Các thread với quyền ưu tiên cao hơn là quan trọng hơn với một chương trình và nên được cấp phát thời gian bộ vi xử lý trước các thread có quyền ưu tiên thấp hơn. Tuy nhiên, các quyền ưu tiên của thread bảo đảm thứ tự trong đó các thread thực thi và phụ thuộc rất nhiều vào platform.