Thread

1. Process là gì ?

**Process** (tiến trình) là một chương trình máy tính đang được thực thi. Khi bạn mở một ứng dụng như trình duyệt web, mỗi phiên bản của ứng dụng đó là một process. Một process bao gồm mã chương trình, dữ liệu, tài nguyên (như tệp tin đang mở), và trạng thái thực thi hiện tại.

Mỗi process hoạt động độc lập và có không gian bộ nhớ riêng. Điều này có nghĩa là một process không thể truy cập trực tiếp vào bộ nhớ của process khác, giúp đảm bảo sự ổn định và an toàn của hệ thống.

2. Thread là gì ?

**Thread** (luồng) là một đơn vị thực thi nhỏ nhất trong một process. Một process có thể có một hoặc nhiều thread. Tất cả các thread trong cùng một process chia sẻ chung không gian bộ nhớ và tài nguyên của process đó.

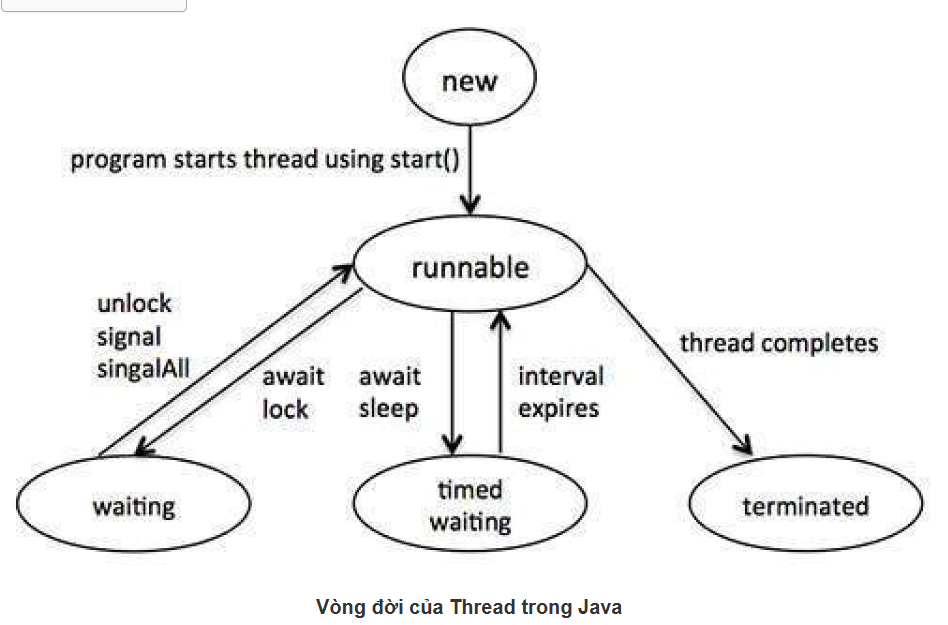
Vì các thread chia sẻ chung tài nguyên, chúng có thể giao tiếp với nhau một cách hiệu quả. *Tuy nhiên, điều này cũng làm nảy sinh vấn đề về đồng bộ hóa, nơi nhiều thread có thể cố gắng truy cập và sửa đổi cùng một dữ liệu cùng một lúc, dẫn đến các lỗi không nhất quán.*

Mối quan hệ giữa process và thread:

* **Process** là một đơn vị độc lập. Nó có tài nguyên riêng (như xưởng sản xuất có máy móc và nguyên liệu riêng).
* **Thread** là các đơn vị thực thi bên trong process. Các công nhân trong xưởng (thread) có thể làm việc cùng nhau và chia sẻ tài nguyên chung (như máy móc và nguyên liệu của xưởng).

|  |  |  |
| --- | --- | --- |
| Đặc điểm | Process | Thread |
| Độc lập | Có không gian bộ nhớ riêng | Chia sẻ không gian bộ nhớ của process |
| Khởi tạo | Tốn nhiều thời gian và tài nguyên | Nhanh chóng và hiệu quả |
| Giao tiếp | Cần cơ chế khá phức tạp IPC (Inter-Process Communication) | Nhanh chóng, đơn giản |
| An toàn | An toàn hơn vì hoạt động độc lập | Ít an toàn hơn, dễ xảy ra vấn đề đồng bộ hóa |

Vòng đời của thread trong java:



3. Có bao nhiêu cách để tạo 1 thread trong java ? Khác biệt giữa việc sử dụng các cách đó gì ?

**Có hai cách chính để tạo một Thread:**

* Tạo Thread bằng cách kế thừa lớp java.lang.Thread
* Tạo Thread bằng cách triển khai giao diện java.lang.Runnable

**Tạo Thread bằng cách kế thừa lớp java.lang.Thread**

Để tạo một Thread bằng cách kế thừa lớp Thread, cần tạo một lớp con của lớp Thread và ghi đè phương thức run(). Phương thức run() chứa mã mà Thread sẽ thực thi khi nó bắt đầu chạy.

Ví dụ:

public class Main {

public static void main(String[] args) {

MyThread myThread = new MyThread();

myThread.start(); // Khởi chạy luồng

}

}

Sử dụng java.lang.Thread khi:

* Khi lớp của ta không cần kế thừa từ một lớp khác (vì Java không hỗ trợ đa kế thừa).
* Khi chỉ cần tạo một đối tượng và gọi phương thức start().

**Tạo Thread bằng cách triển khai interface java.lang.Runnable**

Để làm điều này, cần tạo một lớp triển khai interface Runnable và định nghĩa phương thức run().

Ví dụ:

class MyRunnable implements Runnable {

@Override

public void run() {

// Công việc của luồng

}

}

Sử dụng java.lang.Runnable khi:

* Khi lớp đã kế thừa từ một lớp khác, vì Java cho phép một lớp triển khai nhiều giao diện.
* Tách biệt hóa mã chạy trong Thread (thực thi trong phương thức run()) và việc tạo và quản lý Thread (sử dụng lớp Thread).
* Khi cần chia sẻ tài nguyên giữa các Thread, vì bạn có thể chia sẻ cùng một đối tượng Runnable giữa nhiều Thread.
* Khuyến khích sử dụng với Java 8 trở lên, vì có thể sử dụng biểu thức lambda để tạo một đối tượng Runnable một cách ngắn gọn.

|  |  |  |
| --- | --- | --- |
| Đặc điểm | Kế thừa Thread | Triển khai interface Runnable |
| Tính kế thừa | Không thể kế thừa thêm lớp khác | Có thể kế thừa từ lớp khác, làm cho tăng tính linh hoạt và tái sử dụng mã |
| Tính đa hình | Bị hạn chế | Cho phép sử dụng đa hình vì đối tượng vừa là runnable, vừa là loại khác |
| Tính đóng gói | Lớp và luồng được gắn chặt vào nhau | Tách biệt giữa logic Runnable và cơ chế thực thi Thread. |

4. Thế nào là multi thread ? Sử dụng multi thread mang lại ưu nhược điểm gì ?

**Đa luồng (Multithreading)** cho phép thực hiện đồng thời nhiều tác vụ trong cùng một chương trình, bao gồm hai hoặc nhiều Thread chạy đồng thời, cùng chia sẻ tài nguyên hệ thống như bộ nhớ và CPU. Các Thread có thể làm việc độc lập hoặc phối hợp với nhau để hoàn thành một công việc lớn hơn.

**Ưu và nhược điểm của Multithreading**

**Ưu điểm**

* **Tăng hiệu suất và phản hồi:** Các ứng dụng đa luồng có thể thực hiện nhiều tác vụ cùng một lúc, giúp chúng phản hồi nhanh hơn. Ví dụ, trong một trình duyệt web, một luồng có thể tải xuống một tệp tin trong khi một luồng khác vẫn xử lý các yêu cầu của người dùng, giúp giao diện không bị treo.
* **Tận dụng tài nguyên CPU:** Trên các hệ thống có nhiều lõi xử lý (multi-core), đa luồng cho phép các luồng chạy song song trên các lõi khác nhau, tối đa hóa việc sử dụng CPU và cải thiện đáng kể hiệu suất tổng thể.
* **Chia sẻ tài nguyên hiệu quả:** Các luồng trong cùng một tiến trình chia sẻ chung bộ nhớ và các tài nguyên khác, giúp việc giao tiếp giữa chúng nhanh chóng và hiệu quả hơn so với giao tiếp giữa các tiến trình độc lập.

**Nhược điểm**

* **Phức tạp trong lập trình:** Lập trình đa luồng phức tạp hơn nhiều so với lập trình đơn luồng. Việc quản lý và đồng bộ hóa các luồng để tránh các vấn đề như **race condition** (tình trạng tranh chấp) và **deadlock** (tình trạng bế tắc) đòi hỏi sự cẩn thận và kiến thức chuyên sâu.
* **Nguy cơ lỗi dữ liệu:** Vì các luồng chia sẻ chung tài nguyên, nếu không có cơ chế đồng bộ hóa phù hợp, nhiều luồng có thể cố gắng truy cập và sửa đổi cùng một dữ liệu cùng một lúc, dẫn đến dữ liệu không nhất quán và khó gỡ lỗi.
* **Chi phí quản lý:** Việc tạo và quản lý nhiều luồng cũng tốn một lượng tài nguyên hệ thống nhất định, bao gồm bộ nhớ và thời gian xử lý.

5. Làm thế nào để biết được 1 thread, multi thread đã hoàn thành hay chưa?

**Cách 1: Sử dụng phương thức isAlive()**

Phương thức isAlive() trả về true nếu luồng vẫn đang chạy và false nếu nó đã kết thúc.

**Ưu điểm:** Đơn giản, dễ sử dụng để kiểm tra trạng thái tức thời của một luồng.

**Nhược điểm:** Đây là cách "polling" (kiểm tra liên tục), có thể lãng phí tài nguyên CPU nếu bạn phải lặp đi lặp lại để kiểm tra.

**Cách 2: Sử dụng phương thức join()**

Phương thức join() của một luồng sẽ khiến luồng hiện tại (ví dụ: luồng main) **chờ đợi** cho đến khi luồng đó kết thúc. Đây là cách tiếp cận hiệu quả hơn vì nó không lãng phí CPU.

**Ưu điểm:** Hiệu quả về tài nguyên, luồng gọi không cần lặp đi lặp lại để kiểm tra.

**Nhược điểm:** Luồng hiện tại bị chặn và không thể làm việc gì khác cho đến khi luồng kia kết thúc.

**Cách 3: Sử dụng ExecutorService và Future cho Multithreading**

Đây là cách hiện đại và được khuyến khích khi quản lý một nhóm luồng. ExecutorService quản lý một nhóm các luồng, và Future đại diện cho kết quả của một tác vụ bất đồng bộ.

**Ưu điểm:** Cung cấp một framework mạnh mẽ để quản lý nhiều luồng, dễ dàng lấy kết quả và xử lý lỗi.

**Nhược điểm:** Cú pháp phức tạp hơn một chút so với hai cách trên.

ThreadPool là một tập hợp các luồng làm việc (worker threads) được tạo ra trước và sẵn sàng xử lý các tác vụ. Điều này giúp giảm thiểu chi phí liên quan đến việc tạo và hủy luồng. ThreadPool cho phép  quản lý hiệu quả số lượng luồng đang hoạt động trong ứng dụng và tự động điều chỉnh chúng để đáp ứng nhu cầu.

**Lợi ích khi dùng ExecutorService để quản lý ThreadPool**

1. **Hiệu suất:** Tái sử dụng luồng. Việc tạo và hủy luồng là một quá trình tốn kém về tài nguyên và thời gian. ExecutorService giữ một nhóm luồng có sẵn, giúp giảm đáng kể chi phí này.
2. **Đơn giản:** Bạn không cần phải quan tâm đến việc tạo luồng, vòng đời của chúng, hay việc xử lý các ngoại lệ. ExecutorService tự động hóa những công việc này.
3. **Kiểm soát:** Bạn có thể dễ dàng kiểm soát số lượng luồng tối đa, điều chỉnh hành vi của chúng và quản lý cách thức các tác vụ được xử lý. Các phương thức tiện ích trong lớp **Executors** giúp bạn tạo các nhóm luồng với các cấu hình khác nhau một cách nhanh chóng. Ví dụ:
   * **Executors.newFixedThreadPool(int nThreads)**: Tạo một nhóm luồng có kích thước cố định, hữu ích khi bạn muốn giới hạn số luồng.
   * **Executors.newCachedThreadPool()**: Tạo một nhóm luồng linh hoạt, có thể tự động tăng hoặc giảm số luồng tùy theo nhu cầu.

6. Có giới hạn việc tạo ra bao nhiêu thread trong 1 ứng dụng java hay không?

Không giới hạn cụ thể từ Java nhưng bị giới hạn bởi hệ điều hành và tài nguyên phần cứng của máy tính (bộ nhớ ram, hệ điều hành, CPU)

**Cách giải quyết**

Thay vì tạo một số lượng lớn luồng một cách tùy tiện, thì nên sử dụng **ThreadPool** (nhóm luồng), được quản lý bởi **ExecutorService**. Cách này mang lại nhiều lợi ích:

* **Tái sử dụng luồng:** Các luồng được tạo ra trước và tái sử dụng, tránh chi phí tạo và hủy luồng.
* **Kiểm soát số lượng:** Bạn có thể dễ dàng giới hạn số luồng tối đa để phù hợp với tài nguyên của hệ thống, tránh lỗi OutOfMemoryError và tình trạng quá tải.