### 1. Process là gì?

1.1 Định nghĩa:

* Process là một chương trình đang chạy trong một hệ điều hành. Mỗi process có không gian bộ nhớ và tài nguyên hệ thống riêng, độc lập với các process khác.

1.2 Cơ chế hoạt động

* Khi một chương trình được thực thi, hệ điều hành sẽ tạo ra một process cho chương trình đó.
* Mỗi process có các thành phần chính bao gồm:
  + Code: Phần chứa các lệnh sẽ được thực thi.
  + Data: Các biến, thông tin được chương trình sử dụng.
  + Program Counter: Chỉ định lệnh nào sẽ được thực thi tiếp theo.
  + Stack: Chứa các lệnh và địa chỉ trả về trong các lời gọi hàm.
  + Heap: Dùng để lưu trữ các biến động (dynamic memory).

1.3 Ưu điểm

* Tách biệt và bảo mật: Mỗi process có không gian bộ nhớ riêng, điều này giúp các tiến trình không ảnh hưởng lẫn nhau (giảm rủi ro về bảo mật).
* An toàn: Khi một tiến trình bị lỗi, tiến trình khác vẫn có thể hoạt động bình thường.

1.4 Nhược điểm:

* Tốn tài nguyên: Vì mỗi process có bộ nhớ và tài nguyên độc lập, khi tạo nhiều process có thể gây tiêu tốn bộ nhớ và các tài nguyên hệ thống.
* Chi phí context switching cao: Khi chuyển đổi giữa các tiến trình, hệ điều hành cần lưu trữ và phục hồi trạng thái của từng tiến trình, dẫn đến chi phí cao.

1.5 Trường hợp sử dụng :

* Các ứng dụng yêu cầu tính tách biệt và an toàn cao giữa các thành phần, ví dụ: một trình duyệt web sử dụng các tiến trình độc lập cho từng tab.
* Khi cần chạy các ứng dụng có yêu cầu cao về bảo mật và sự ổn định, nơi mà lỗi của một tiến trình không ảnh hưởng đến tiến trình khác.

2. Thread

2.1 Định nghĩa :

* Thread là đơn vị nhỏ nhất trong quá trình thực thi của một chương trình. Mỗi tiến trình có thể chứa nhiều thread, và các thread trong cùng một process chia sẻ cùng một không gian bộ nhớ và tài nguyên.

2.2 Cơ chế hoạt động :

* Một process có thể tạo ra nhiều thread, và các thread này cùng chia sẻ bộ nhớ và tài nguyên (heap, data) của tiến trình mẹ.
* Mỗi thread vẫn có stack và program counter riêng để thực hiện các nhiệm vụ độc lập.

2.3 Ưu điểm :

* Chi phí thấp: Tạo thread ít tốn kém hơn tạo process.
* Các thread trong cùng tiến trình có thể chia sẻ tài nguyên như bộ nhớ, giúp tăng cường hiệu quả làm việc khi xử lý các tác vụ song song.

2.4 Nhược điểm :

* Khó quản lý: Vì các thread chia sẻ bộ nhớ và tài nguyên, việc quản lý và đồng bộ giữa các thread có thể phức tạp, dễ xảy ra lỗi về race condition hoặc deadlock.
* Nếu một thread gặp sự cố , toàn bộ tiến trình có thể bị ảnh hưởng hoặc kết thúc.

2.5 Trường hợp sử dụng :

* Các ứng dụng yêu cầu thực thi song song nhiều tác vụ mà có thể chia sẻ tài nguyên, ví dụ: một ứng dụng web server sử dụng nhiều thread để xử lý các yêu cầu cùng lúc.
* Khi cần tăng cường hiệu suất bằng cách sử dụng multithreading .

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

3.1 Có 2 các để tạo 1 thread trong java :

3.1.1 Kế thừa từ class Thread :

* Cơ chế hoạt động :
  + Khi bạn gọi Thread.start(), một thread mới của hệ điều hành được tạo ra, thực thi song song với các thread khác trong chương trình.
  + Lớp con sẽ tựu động overridding method run() của class Thread.
* Ưu điểm
  + Triển khai nhanh và dễ hiểu .
* Nhược điểm
  + Chỉ có thể kế thùa class Thread mà ko thể kế thừa lớp khác .

3.1.2 Triển khai Runnable interface :

* Cơ chế hoạt động
  + Tạo ra dối tượng runnable và gọi contructor cuat class thread để khởi tạo luồng mới .
  + Thread được tạo ra khi triển khai interface runnable sẽ gọi đến hàm run bên trong runnable chứ không phải thread.
* Ưu điểm
  + Có thể sử dụng đa kế thừa nên linh hoạt hơn .
* Nhược điểm
  + Phức tạp hơn so với thread.

3.1.3 So sánh 2 cách :

* Tính linh hoạt :
  + Class thread không hỗ trợ đa kế thừa.
  + Interface runnable có hỗ trọ đa kế thừa .
* Cơ chế hoạt động :
  + Thread lớp kế thừa tự dộng override phương thức run().
  + Runnable truyền contructor của thread , phương thức void cần phải được triển khai thủ công.
* Mục đích sử dụng
  + Thread: Được sử dụng khi bạn muốn mở rộng lớp Thread và không cần kế thừa thêm các lớp khác.
  + Runnable sử dụng thường xuyên hơn trong thực tế vì dễ dàng khi triển khai đa luồng hơn .

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

4.1 Định nghĩa :

* Multi-threading là một kỹ thuật trong lập trình cho phép một chương trình thực hiện nhiều công việc cùng lúc bằng cách chia nhỏ chương trình thành các thread riêng lẻ.
* Mỗi thread là một quá trình con, có thể thực thi đồng thời với các thread khác trong cùng một chương trình. Các luồng này có thể chia sẻ cùng bộ nhớ và tài nguyên của process chính, nhưng mỗi thread có program counter riêng của mình.

4.2 Ưu điểm :

* Tận dụng tài nguyên CPU tốt hơn:
  + Multi-threading giúp tận dụng hiệu quả hơn các CPU . Các thread có thể được thực thi song song trên các CPU khác nhau, giúp tăng tốc độ xử lý của chương trình.
* Cải thiện hiệu năng:
  + Multi-threading có thể cải thiện hiệu năng, đặc biệt trong các ứng dụng yêu cầu thực hiện nhiều tác vụ đồng thời.
  + Ví dụ: các ứng dụng đa người dùng hoặc các tác vụ nặng về I/O như tải dữ liệu từ mạng.
* Trải nghiệm người dùng tốt hơn:
  + Đối với các ứng dụng giao diện người dùng (UI), multi-threading giúp giao diện không bị "treo" khi chương trình xử lý các tác vụ nặng, như tải file hoặc kết nối mạng.

4.3 Nhược điểm :

* Thao tác khai báo và sử dụng multi-threading rất phức tạp .
  + Việc đồng bộ hóa (synchronization) giữa các luồng để tránh tranh chấp dữ liệu (data races) và các lỗi như deadlock (khoá chết) có thể rất khó kiểm soát.
* Vấn đề đồng bộ hoá :
  + Các luồng chia sẻ bộ nhớ và tài nguyên với nhau, điều này có thể dẫn đến lỗi đồng bộ , khi nhiều luồng cùng truy cập và thay đổi cùng một dữ liệu.
  + Điều này có thể gây ra lỗi như race condition, dẫn đến dữ liệu không nhất quán hoặc thậm chí crash chương trình .
* Khó khăn trong debug
  + Việc theo dõi và gỡ lỗi một chương trình multi-threading có thể phức tạp hơn rất nhiều so với lập trình đơn luồng, vì các lỗi có thể xảy ra không dự đoán trước và khó tái hiện.

4.4 Trường hợp sử dụng :

* Các ứng dụng có nhiều tác vụ I/O có thể hưởng lợi từ multi-threading, vì các luồng khác có thể tiếp tục hoạt động trong khi một luồng chờ kết quả từ tác vụ I/O.
* Multi-threading giúp giữ giao diện người dùng phản hồi nhanh, ngay cả khi các tác vụ nặng chạy ngầm.

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

5.1 Phương thức join()

* Cơ chế hoạt động :
  + Khi bạn gọi join() trên một thread, main thread sẽ bị chặn lại và đợi cho đến khi child thread đó hoàn thành công việc
* Ưu điểm:
  + Dễ sử dụng và đơn giản.
  + Giúp đảm bảo luồng chính không kết thúc trước khi các luồng con hoàn tất.
  + Cách tiếp cận trực tiếp để đợi một thread hoàn thành.
* Nhược điểm:
  + Chặn luồng chính, khiến chương trình phải đợi thread con xong trước khi tiếp tục, điều này có thể không hiệu quả khi bạn muốn thực hiện đa nhiệm.
  + Không linh hoạt nếu có nhiều thread cần quản lý (đặc biệt là trong ứng dụng có nhiều luồng chạy song song).
* Trường hợp sử dụng:
  + Khi bạn có số lượng ít thread và muốn đảm bảo chương trình chính không tiếp tục cho đến khi thread hoàn thành.
  + Phù hợp cho các chương trình đơn giản hoặc ít luồng, nơi quản lý luồng không phức tạp.

5.2 Thread pool()

* Thread Pool sử dụng lại một nhóm nhỏ các thread có sẵn để xử lý nhiều công việc hơn thay vì tạo mới Thread liên tục, giúp tiết kiệm tài nguyên và tăng hiệu suất.
* Cơ chế hoạt động :
  + Khởi tạo Thread Pool:
    - Khi tạo Thread Pool, hệ thống sẽ tạo ra một số lượng cố định hoặc linh hoạt các luồng dựa trên cấu hình của bạn
  + Thực thi nhiệm vụ:
    - Khi một tác vụ được gửi đến Executor thông qua các phương thức như submit() hoặc execute(), Executor sẽ tìm một thread nhàn rỗi trong pool để xử lý nhiệm vụ đó.
    - Nếu tất cả ác thread đều bận , nhiệm vụ sẽ được xếp vào Task Queue để chờ.
  + Quản lý Task Queue:
    - Khi một thread hoàn thành nhiệm vụ hiện tại, nó sẽ kiểm tra Task Queue để xem có nhiệm vụ nào đang chờ không. Nếu có, nó sẽ lấy nhiệm vụ từ hàng đợi và tiếp tục thực hiện.
    - Nếu Task Queue trống, thread sẽ chuyển sang trạng thái chờ, sẵn sàng cho nhiệm vụ mới.
  + Giới hạn số lượng luồng:
    - Thread Pool có thể được cấu hình với một số lượng tối đa các luồng (ví dụ: 5 hoặc 10). Nếu tất cả các luồng đều bận và Task Queue cũng đã đầy, Executor có thể từ chối các nhiệm vụ mới hoặc đưa chúng vào trạng thái chờ.
    - Tùy thuộc vào cấu hình, Thread Pool có thể linh hoạt tạo thêm luồng nếu cần thiết hoặc ngừng nhận nhiệm vụ
  + Shutdown và Cleanup:
    - Khi không còn nhiệm vụ nào cần xử lý và chương trình cần kết thúc, bạn có thể gọi phương thức shutdown() để thông báo cho Thread Pool rằng không cần nhận thêm nhiệm vụ mới.
    - Các thread sẽ tiếp tục thực hiện các nhiệm vụ hiện tại, và sau khi hoàn tất, chúng sẽ tự động được dừng lại.
* Ưu điểm :
  + Tái sử dụng luồng: Thay vì tạo và hủy luồng liên tục, Thread Pool cho phép tái sử dụng các luồng có sẵn. Điều này giúp giảm overhead (chi phí tài nguyên) liên quan đến việc tạo và hủy luồng.
  + Giới hạn số lượng luồng: Cho phép kiểm soát và giới hạn số lượng luồng chạy đồng thời, giúp tránh tình trạng quá tải CPU hoặc bộ nhớ
  + Hiệu suất tốt hơn:Việc tạo mới luồng tốn thời gian và tài nguyên, trong khi việc sử dụng lại các luồng trong Thread Pool nhanh hơn và hiệu quả hơn
  + Quản lý dễ dàng:Thay vì phải trực tiếp quản lý từng thread, người lập trình chỉ cần nộp nhiệm vụ cho Thread Pool và để hệ thống tự điều phối.
  + Dễ mở rộng và tái sử dụng.
* Nhược điểm :
  + Cấu hình phức tạp:Thread Pool cần được cấu hình cẩn thận (ví dụ: số lượng luồng tối thiểu, tối đa, kích thước hàng đợi) để đạt hiệu suất tối ưu. Nếu cấu hình không đúng, có thể gây ra tình trạng thiếu hoặc thừa luồng.
  + Deadlock (Khoá chết):Nếu các nhiệm vụ trong Thread Pool phụ thuộc lẫn nhau hoặc yêu cầu tài nguyên mà chưa được giải phóng, điều này có thể gây ra tình trạng deadlock, khi đó không có thread nào có thể tiến hành.
  + Giới hạn tài nguyên:Nếu tất cả các thread trong pool đều đang bận và hàng đợi cũng đầy, các tác vụ mới có thể bị từ chối hoặc gây ra chậm trễ trong xử lý
* Trường hợp sử dụng :
  + Khi có nhiều nhiệm vụ nhỏ cần thực hiện song song.
  + Khi cần quản lý một số lượng lớn Thread nhưng muốn giới hạn số lượng Threaad chạy đồng thời để tiết kiệm tài nguyên.
  + Thích hợp cho các ứng dụng yêu cầu hiệu suất cao hoặc cần thực hiện nhiều tác vụ song song (ví dụ như web server, các hệ thống xử lý dữ liệu lớn.

5.3 Flag

* Định nghĩa :Flag là một biến (thường là kiểu boolean) được chia sẻ giữa các luồng và có thể được sử dụng để báo hiệu trạng thái hoặc giao tiếp giữa các luồng.
* Cơ chế hoạt động :
* Một luồng có thể set (đặt) hoặc clear (xóa) giá trị của flag.
* Các luồng khác sẽ kiểm tra flag này trong quá trình chạy và thực hiện hành động tương ứng khi flag có một giá trị nhất định.
* Ưu điểm :
* Đơn giản và dễ hiểu.
* Không chặn luồng, tức là luồng vẫn có thể thực hiện công việc khác trong khi kiểm tra flag.
* Nhược điểm :
* Yêu cầu kiểm tra liên tục (polling), gây tốn tài nguyên nếu luồng liên tục kiểm tra trạng thái của flag.
* Cần được bảo vệ (bằng các cơ chế khóa như mutex) nếu nhiều luồng có thể truy cập và thay đổi flag cùng lúc, để tránh các vấn đề như race condition (điều kiện tranh chấp).
* Trường hợp sử dụng :
* Thích hợp cho các trường hợp mà một luồng chỉ cần biết trạng thái của luồng khác mà không cần phải chờ đợi. Ví dụ, để báo hiệu trạng thái của một tiến trình con, hoặc theo dõi tình trạng của tài nguyên trong hệ thống.

5.4 event()

* Định nghĩa : Event là một cơ chế đồng bộ hóa mạnh mẽ hơn. Nó không chỉ đơn thuần là một giá trị boolean mà còn cung cấp các phương thức để một luồng chờ (wait) cho đến khi một sự kiện xảy ra (tức là event được set), hoặc để báo hiệu cho các luồng khác khi sự kiện đã xảy ra.
* Cơ chế hoạt động :
* Một event bắt đầu ở trạng thái clear (tức là chưa xảy ra).
* Một luồng có thể gọi wait() để chờ event này được set.
* Khi một event được set, tất cả các luồng đang chờ nó sẽ được báo hiệu và tiếp tục công việc.

Ưu điểm

* Không cần kiểm tra liên tục như flag. Luồng chỉ cần chờ sự kiện và sẽ được đánh thức khi sự kiện xảy ra.
* Dễ dàng quản lý đồng bộ nhiều luồng.
* Cơ chế chờ của wait() không tốn CPU.

Nhược điểm

* Khó hơn flag trong việc sử dụng vì cần có hiểu biết về cơ chế đồng bộ hóa.
* Dễ gây ra deadlock (khoá chết) nếu không xử lý cẩn thận, đặc biệt là trong các hệ thống có nhiều event và luồng phức tạp.

Trường hợp sử dụng :

* Thích hợp khi bạn muốn một hoặc nhiều luồng chờ một cách hiệu quả cho đến khi một sự kiện cụ thể xảy ra (ví dụ, luồng tải dữ liệu hoàn tất hoặc một tài nguyên trở nên sẵn sàng).
* Event rất hữu ích trong việc đồng bộ hóa nhiều luồng mà không cần phải sử dụng các cơ chế khóa phức tạp.

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

* Trong Java, không có một giới hạn cố định về số lượng thread mà một ứng dụng có thể tạo ra, tuy nhiên, số lượng thread tối đa mà một ứng dụng Java có thể tạo ra phụ thuộc vào nhiều yếu tố khác nhau, bao gồm:
  + Bộ nhớ khả dụng (RAM): Mỗi thread tiêu thụ một lượng bộ nhớ nhất định cho stack của nó. Nếu tạo ra quá nhiều thread mà không đủ bộ nhớ, ứng dụng sẽ ném ra lỗi OutOfMemoryError.
  + Hệ điều hành: Một số hệ điều hành có giới hạn cứng về số lượng thread mà một tiến trình có thể tạo ra.
  + Tài nguyên hệ thống: Ngoài bộ nhớ, CPU và các tài nguyên khác (ví dụ: I/O) cũng có thể trở thành nút thắt .
  + JVM (Java Virtual Machine): Mặc dù JVM không áp đặt giới hạn cụ thể về số thread, hiệu suất sẽ giảm sút nếu có quá nhiều thread vì cơ chế lập lịch thời gian của CPU sẽ trở nên kém hiệu quả.
* Trong thực tế, số lượng thread nên được kiểm soát cẩn thận và thường xuyên sử dụng các mô hình đa luồng hiệu quả hơn như Thread Pool (Sử dụng các lớp như ExecutorService) để quản lý số lượng thread tối ưu.