1. [**Race Condition là gì? Làm sao để tránh gặp phải race condition?**](https://discuss.grokking.org/t/race-condition-la-gi-lam-sao-d-tranh-g-p-ph-i-race-condition/325)

Trong lĩnh vực lập trình Race condition là một tình huống xảy ra khi nhiều threads cùng truy cập và cùng lúc muốn thay đổi dữ liệu (có thể là 1 biến, 1 row trong database, 1 vùng shared data, memory , etc...).

Vì thuật toán chuyển đổi việc thực thi giữa các threads có thể xảy ra bất cứ lúc nào nên không thể biết được thứ tự của các threads truy cập và thay đổi dữ liệu đó sẽ dẫn đến giá trị của data sẽ không như mong muốn. Kết quả sẽ phụ thuộc vào thuật toán thread scheduling của hệ điều hành.

Quá trình các thread thực thi lệnh trông như 1 cuộc đua giữa các vận động viên điền kinh olympic vì vậy có thể liên tưởng đến thuật ngữ (keyword) **“Race condition”**.

**Ví dụ cụ thể**

**public** **class** **Counter** {

**protected** **long** count = 0;

**public** **void** **add**(**long** **value**) {

**this**.count = **this**.count + **value**;

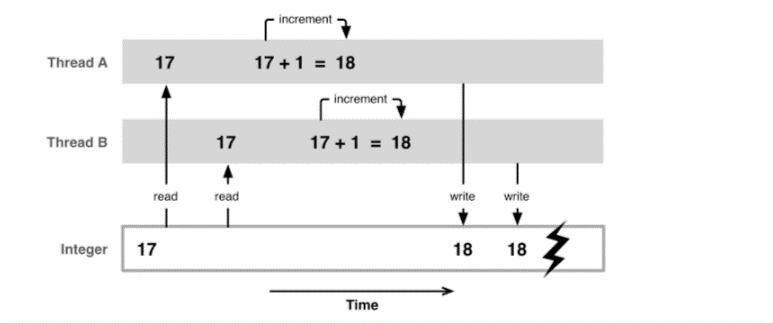
}

}

Tình huống có 2 thread A và B cùng thực thi phương thức add trên 1 thể hiện của class Counter các bước sẽ như sau:

1. Đọc giá trị count từ memory.
2. Cộng value vào count.
3. Gán lại giá trị count.

Quá trình thực thi của thread A và B có thể diễn ra như sau:



Để ngăn chặn việc đó xảy ra phải để hàm add() được thực thi tuần tự. Có thể lock hoặc synchronized method add().

**public** **class** **Counter** {

**protected** **long** count = 0;

**public** synchronized **void** **add**(**long** **value**) {

**this**.count = **this**.count + **value**;

}

}

**Kinh nghiệm trong java**

Như đã thấy race condition là vấn đề thường xuyên xảy ra trong lập trình. Với kinh nghiệm về java khi sử dụng spring boot, spring jpa và mysql đã gặp phải tình huống trên. Các giải pháp tôi biết như sau:

1. Dùng một message queue để chứa và xử lí tuần tự việc đọc, ghi dữ liệu.
2. Sử dụng cơ chế Locking trong JPA để đồng bộ dữ liệu.
3. Sử dụng câu update trực tiếp và đưa phần tính toán xuống data base. Ví dụ với bài toán chuyển tiền của các ngân hang tôi có thể làm như sau

**update** **table** t **set** t.balance = t.balance + amount;

Tôi đã dùng phương án 3, vì theo tôi:

• Phương án 1 tất cả tài khoản của user đều phải xử lý tuần tự rất tốn thời gian để làm và không có được hiệu suất tốt nhất.

• Phương án 2 thì chúng ta phải try catch exception và xử lí lại record đó nên theo tôi cũng không phải giải pháp tốt.

**1. Synchronous là gì?**

**Synchronous có nghĩa là xử lý đồng bộ**, chương trình sẽ chạy theo từng bước và chỉ khi nào bước 1 thực hiện xong thì mới nhảy sang bước 2, khi nào chương trình này chạy xong mới nhảy qua chương trình khác. Đây là nguyên tắc cơ bản trong lập trình mà bạn đã được học đó là khi biên dịch các đoạn mã thì trình biên dịch sẽ biên dịch theo thứ tự từ trên xuống dưới, từ trái qua phải và chỉ khi nào biên dịch xong dòng thứ nhât mới nhảy sang dòng thứ hai, điều này sẽ sinh ra một trạng thái ta hay gọi là **trạng thái chờ**. Ví dụ trong quy trình sản xuất dây chuyền công nghiệp được coi là một hệ thống xử lý đồng bộ.

Synchronous hai mặt là mặt xấu và mặt tốt.

**Mặt tốt của Synchronous**

Chương trình sẽ chạy theo đúng thứ tự và có nguyên tắc nên sẽ không mắc phải các lỗi về tiến trình không cần thiết. Không chỉ trong lập trình mà trong thực tế cũng vậy, một công ty đưa ra quy trình đồng bộ sẽ đảm bảo được chất lượng của sản phẩm, nếu bị lỗi thì sẽ biết ngay là lỗi tại quy trình nào và từ đó sẽ dễ dàng khắc phục.

**Mặt xấu của Synchronous**

Chương trình chạy theo thứ tự đồng bộ nên sẽ sinh ra trạng thái chờ và là không cần thiết trong một số trường hợp, lúc này bộ nhớ sẽ dễ bị tràn vì phải lưu trữ các trạng thái chờ vô duyên đó.

Khi bạn viết một chương trình quản lý và trong đó có thao tác lưu, mỗi khi lưu bạn yêu cầu người dùng có muốn lưu hay không? Nếu muốn lưu thì click Yes, ngược lại click No. Trường hợp nay gây tai họa nếu người dùng vô tình chỉ click Lưu mà không chú ý đến câu hỏi mà hệ thống đưa ra nên ngồi nhâm nhi cafe, đột nhiên cúp điện thế là cứ tưởng đã lưu rồi :) Vậy quy trình xử lý nên đưa ra chức năng lưu tự động, nghĩa là thao tác lưu sẽ bỏ qua bước hỏi đáp kia đi, không nhất thiết phải chờ nó OK mới lưu.

**2. Asynchronous là gì?**

Ngược lại với Synchronous thì Asynchronous là xử lý bất động bộ, nghĩa là chương trình có thể nhảy đi bỏ qua một bước nào đó, vì vậy Asynchronous được ví như một chương trình hoạt động không chặt chẽ và không có quy trình nên việc quản lý rất khó khăn. Nếu một hàm A phải bắt buộc chạy trước hàm B thì với Asynchronous sẽ không thể đảm bảo nguyên tắc này luôn đúng.

**Mặt tốt của Asynchronous**

Có thể xử lý nhiều công việc một lúc mà không cần phải chờ đợi nên tạo cảm giác thoải mái :) Ví dụ bạn đi ký một văn bản ở Xã, Phường thì nếu bạn có tiền bạn sẽ bỏ qua được một vài công đoạn phải không nào, lúc đó măt sẽ tươi rói và đương nhiên là anh nhân viên cũng tươi không kém :)

**Mặt xấu của Asynchronous**

Nếu một chuong trình đòi hỏi phải có quy trình thì bạn không thể sử dụng Asynchronous được, điển hình là trong quy trình sản xuât **một sản phẩm** của các nhà máy công nghiệp không thể áp dụng kỹ thuật làm nhiều công việc một lúc thế này được. Còn về chương trình trong lập trình thì sao? Một thao tác thêm dữ liệu phải thông qua hai công đoạn là validate dữ liệu và thêm dữ liệu, nếu thao tác validate xảy ra sau thao tác thêm thì còn gì tệ hại hơn nữa :).

**Deadlock vs Starvation**

Deadlock và starvation là đều là điều kiện mà trong đó các tiến trình yêu cầu tài nguyên thì bị trì hoãn trong một thời gian dài. Mặc dù Deadlock và starvation đều khác nhau về nhiều mặt.

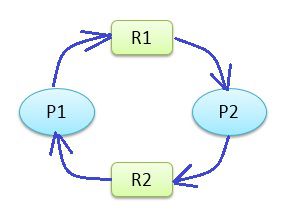
Deadlock là một điều kiện trong đó không có tiến trình nào tiến hành thực thi và phải chờ đợi tài nguyên mà tiến trình khác đang sử dụng.

Starvation, tiến trình có mức độ ưu tiên cao liên tục sử dụng các tài nguyên làm ngăn chặn tiến trình có độ ưu tiên thấp có được tài nguyên.

| **BASIS FOR COMPARSION** | **DEADLOCK** | **STARVATION** |
| --- | --- | --- |
| Basic | Không có tiến trình nào chạy tiếp, bị chặn. | Các tiến trình có độ ưu tiên thấp bị chặn, Tiến trình có độ ưu tiên cao chạy tiếp |
| Điều kiện nảy sinh | Sự xuất hiện của Mutual exclusion, Hold and wait, No preemption and Circular wait simultaneously. | Thực thi các ưu tiên, quản lý tài nguyên mất kiểm soát. |
| Other name | Circular wait. | Lifelock. |
| Resources | tài nguyên được yêu cầu bị chặn bởi các quy trình khác. | các tài nguyên được yêu cầu liên tục được sử dụng bởi các quy trình ưu tiên cao. |

**Deadlock**

là tình huống trong đó một số quy trình trong CPU cạnh tranh về số lượng tài nguyên hữu hạn có sẵn trong CPU. Ở đây, mỗi quy trình giữ một tài nguyên và chờ đợi để có được một tài nguyên được nắm giữ bởi một số quy trình khác. Tất cả các quy trình chờ tài nguyên theo kiểu vòng tròn. Trong hình ảnh bên dưới, bạn có thể thấy rằng Quá trình P1 đã có được tài nguyên R2 được yêu cầu bởi quy trình P2 và Quy trình P1 đang yêu cầu tài nguyên R1 được giữ lại bởi R2. Vì vậy, quá trình P1 và P2 tạo thành một bế tắc.



Có bốn điều kiện phải xảy ra đồng thời :

* **Mutual exclusion:** Chỉ một quy trình tại một thời điểm có thể sử dụng tài nguyên nếu quy trình khác yêu cầu cùng một tài nguyên, nó phải đợi cho đến khi quy trình sử dụng tài nguyên giải phóng tài nguyên đó.
* **Hold and Wait:** Một quy trình phải được giữ một tài nguyên và chờ đợi để có được một tài nguyên khác được nắm giữ bởi một quy trình khác.
* **No Preemption:** Quá trình nắm giữ các tài nguyên không thể được ưu tiên. Quá trình giữ tài nguyên phải giải phóng tài nguyên một cách tự nguyện khi nó đã hoàn thành nhiệm vụ.
* **Circular wait:** Quá trình phải chờ tài nguyên theo kiểu vòng tròn. Giả sử chúng ta có ba quy trình {P0, P1, P2}. P0 phải chờ tài nguyên do P1 nắm giữ; P1 phải chờ để có được tài nguyên được giữ bởi quy trình P2 và P2 phải chờ để có được quy trình do P0 nắm giữ.

**Starvation** có thể được định nghĩa là khi tiến trình yêu cầu tài nguyên và tài nguyên đó đã được sử dụng liên tục bởi các tiến trình khác thì quá trình yêu cầu phải đối mặt với nạn đói. Trong tình trạng đói, một quá trình sẵn sàng để thực thi chờ CPU phân bổ tài nguyên. Nhưng quá trình này phải chờ vô thời hạn vì các quy trình khác liên tục chặn các tài nguyên được yêu cầu.

Vấn đề starvation thường xảy ra trong thuật toán lập lịch ưu tiên. Trong thuật toán lập lịch ưu tiên, quy trình có mức ưu tiên cao hơn luôn được phân bổ tài nguyên, ngăn quá trình ưu tiên thấp hơn nhận được tài nguyên được yêu cầu.

**Một số nguyên nhân phổ biến của Starvation như sau:**

1. Nếu một quy trình không bao giờ được cung cấp các tài nguyên mà nó yêu cầu để thực thi do các quyết định phân bổ tài nguyên bị lỗi, thì việc chết đói có thể xảy ra.
2. Một quy trình ưu tiên thấp hơn có thể đợi mãi nếu các quy trình ưu tiên cao hơn liên tục độc quyền bộ xử lý.
3. Đói có thể xảy ra nếu không có đủ tài nguyên để cung cấp cho mọi quy trình theo yêu cầu.
4. Nếu lựa chọn ngẫu nhiên các quy trình được sử dụng thì một quy trình có thể chờ trong một thời gian dài vì không chọn.

**Một số giải pháp có thể được thực hiện trong một hệ thống để xử lý Starvation như sau:**

1. Một bộ phận quản lý độc lập có thể được sử dụng để phân bổ các nguồn lực. Bộ quản lý tài nguyên này phân phối tài nguyên một cách công bằng và cố gắng tránh chết đói.
2. Nên tránh lựa chọn ngẫu nhiên các quy trình để phân bổ tài nguyên hoặc phân bổ bộ xử lý vì chúng khuyến khích chết đói.
3. Lược đồ ưu tiên phân bổ tài nguyên nên bao gồm các khái niệm như aging, trong đó mức độ ưu tiên của một quá trình được tăng lên khi nó chờ đợi lâu hơn. Điều này tránh chết đói.

| **BASIS FOR COMPARISON** | **SEMAPHORE** | **MUTEX** |
| --- | --- | --- |
| Basic | Cơ chế báo hiệu | Cơ chế khóa |
| Existence | Là biến số nguyên | Là đối tượng |
| Function | Semaphore cho phép nhiều luồng chương trình truy cập vào một thể hiện hữu hạn của tài nguyên. | Mutex cho phép nhiều luồng chương trình truy cập vào một tài nguyên nhưng không đồng thời. |
| Ownership | Giá trị semaphore có thể được thay đổi bởi bất kỳ tiến trình có được hoặc giải phóng tài nguyên. | Khóa đối tượng Mutex chỉ được giải phóng bởi tiến trình đã có được khóa trên nó. |
| Categorize | Semaphore có thể được phân loại thành đếm semaphore và semaphore nhị phân. | Ko có |
| Operation | Semaphore value is modified using wait() and signal() operation. | Đối tượng Mutex bị khóa hoặc mở khóa bởi tiến trình yêu cầu hoặc giải phóng tài nguyên. |
| Resources Occupied | Nếu tất cả các tài nguyên đang được sử dụng, quá trình yêu cầu tài nguyên thực hiện thao tác Wait () và tự chặn cho đến khi số lượng semaphore trở nên lớn hơn một. | Nếu một đối tượng mutex đã bị khóa, quá trình yêu cầu tài nguyên chờ và được hệ thống xếp hàng cho đến khi khóa được giải phóng. |

vd vùng race condition A thì mutex chỉ cho 1 thread vào đó thôi, còn sermaphore có thể giới hạn hơn 1

còn binary sermaphore thì nó chỉ có 2 giá trị là 0 vs 1 nên cũng chỉ cho được 1 thread vào A nhưng khác vs mutex là mutex chỉ có thể tự nó unlock cái nó khóa

Semaphore tốt hơn trong trường hợp có nhiều instances of resources có sẵn. Trong trường hợp single shared resource thì mutex tốt hơn.

**Cấp phát bộ nhớ**

<https://cpp.daynhauhoc.com/8/4-cap-phat-bo-nho-dong/>

<https://cpp.daynhauhoc.com/8/10-phan-loai-cac-vung-nho-stack-va-heap/>