Semaphore - Đèn báo

Tạ Hữu Bình, Trần Trọng Hiệp, Lê Huy Hoàng, Võ Thục Khánh Huyền

Ngày 5 tháng 6 năm 2021

Tóm tắt nội dung

Semaphore - Đèn báo là giải pháp để giải quyết sự xung đột giữa các tiến trình. Bài báo cáo này đi tìm hiểu về cách thức hoạt động của semaphore trong các hệ thống đa tiến trình, cụ thể hơn là áp dụng trong các bài toán cổ điển như Producer-Consumer hay bài toán triết gia ăn tối. Mỗi bài toán đều đưa ra những phương pháp sử dụng semaphore khác nhau, tuy nhiên vẫn hướng đến mục tiêu chung là ngăn ngừa deadlock xảy ra và không một tiến trình nào bị bỏ đói.

Mục lục

1 Giới thiệu về đèn báo

Đèn báo là biến tài nguyên S, khởi tạo bằng khả năng phục vụ của tài nguyên nó điều độ. Đèn báo S được khởi tạo như sau:

```
struct Semaphore {
    int value;
    struct process* Ptr;
};
```

Chỉ có thể thay đổi giá trị bởi hai thao tác cơ bản là wait(S) và signal(S).

1. Thao tác wait(S)

```
void wait(Semaphore S) {
    S.value--;
    if(S.value < 0) {
        Add process to S.Ptr;
        block();
    }
}
```

2. Thao tác signal(S)

```
void signal(Semaphore S) {
    S.value++;
    if(S.value <= 0) {
        Pull out process P from S.Ptr;
        wakeup(P);
    }
}
```

Ngoài ra, chúng ta còn kết hợp thêm hai thao tác khác là block() và wakeup(P).

1. block(): Ngừng tạm thời tiến trình đang thực hiện
2. Thực hiện tiếp tiến trình P dừng bởi lệnh block();

2 Các bài toán về đèn báo

2.1 Bài toán Producer-Consumer

Lập trình đa luồng thường đòi hỏi sự phân chia giữa các luồng. Bài toán Producer-consumer xét luồng ra thành hai loại: **luồng sản xuất** (Producer) và **luồng tiêu thụ** (Consumer). Luồng sản xuất sẽ tạo ra các item (*tạm dịch* "đơn vị sản phẩm") vào trong một cấu trúc dữ liệu nào đó còn luồng tiêu thụ sẽ đưa sản phẩm ra và xử lý chúng.

Có hai ràng buộc sẽ được thảo luận liên quan đến tính đồng bộ của hai loại luồng:

• Thứ nhất, khi thêm hoặc loại bớt các item, hai loại luồng sẽ phải loại trừ lẫn nhau.

• Nếu luồng tiêu thụ muốn lấy ra item trong buffer (*tạm dịch* "vùng đệm") rỗng, nó sẽ block (*tạm dịch* "tự chặn lại") cho đến khi ít nhất một item được thêm vào.

Giả thiết luồng sản xuất tiến hành như sau:

Basic producer code

```
event = waitForEvent ()
buffer . add ( event )
```

Giả thiết luồng tiêu thụ tiến hành như sau:

Basic consumer code

```
event = buffer . get ()
event . process ()
```

Trong đó, waitForEvent và event.process có thể chạy song song. Câu hỏi đặt ra là: Cần phải chỉnh sửa đoạn code thế nào để thỏa mãn các ràng buộc nêu trên.

2.1.1 Hướng giải bài toán Producer-Consumer

Có một số biến sẽ sử dụng như sau:

Producer-consumer initialization

```
mutex = Semaphore (1)
items = Semaphore (0)
local event
```

Trong đó, mutex sẽ "quy định" luồng sẽ được được thực hiện đoạn lệnh nào đó. Nếu items dương, nó "biểu hiện" số lượng item trong buffer, ngược lại trị số của nó là số luồng tiêu thụ trong hàng đợi. Còn lại, event là biến địa phương của mỗi luồng.

Có một số cách sử dụng loại biến này trong các môi trường khác nhau:

- Mỗi luồng có run-time stack riêng.
- Các luồng có vai trò là các đối tượng (objects) chứa thuộc tính nào đó.
- Các luồng có ID riêng.

Xét đoạn code sau:

Producer solution

```
event = waitForEvent ()
mutex . wait ()
    buffer . add ( event )
    items . signal ()
mutex . signal ()
```

Consumer solution

```
items . wait ()
mutex . wait ()
    event = buffer . get ()
mutex . signal ()
event . process ()
```

Luồng sản xuất sẽ không phải chờ sự cho phép nào từ bên ngoài để vào buffer cho đến khi nó nhận được một event (*tạm dịch* "sự kiện") nào đó. Có thể có nhiều hơn một luồng có thể thạy waitForEvent đồng thời. Items semaphore sẽ "lưu" số item hiện có trong buffer.

mutex sẽ quy định luồng vào buffer. Cụ thể, nếu luồng sản xuất gọi thành công mutex.wait(), nó sẽ được quyền "tăng" items lên một đơn vị trong khi luồng tiêu thụ phải chờ đợi, sau khi mutex.signal() được thực hiện, hai luồng sẽ đều có quyền gọi mutex.wait(), nhưng tất nhiên chỉ một luồng thành công. Điều ngược lại tương tự với luồng tiêu thụ.

Một điều đáng chú ý, ở luồng tiêu thụ, trước khi có thể tác động đến mutex, nó phải "giảm" được items đi một đơn vị. Nếu mà items không dương, luồng tiêu thụ phải chờ đợi.

Thuật toán trên không sai nhưng ta có thể cải tiến đi một chút. Giả sử rằng, ngay sau khi luồng sản xuất làm tăng items và luồng tiêu thụ lập tức giảm items. Nếu luồng sản xuất chưa "giải phóng" mutex, luồng tiêu thụ sẽ block. Mà quá trình **Blocking** và **"Waking up"** (dừng block) tương đối tốn kém và có thể dẫn đến giảm hiệu năng, vì vậy ta có để cải tiến đoạn code của luồng sản xuất như sau:

Improved producer solution

```
event = waitForEvent ()
mutex . wait ()
    buffer . add ( event )
mutex . signal ()
items . signal ()
```

Lúc đó, trừ phi ngay sau khi luồng sản xuất tăng items, có luồng sản xuất khác giành được mutex tiếp, luồng tiêu thụ sẽ luôn được tác động đến mutex khi đã tiến hành giảm items.

Tuy nhiên, ở trên items được cho rằng "lưu trữ" số lượng item có trong buffer; nhưng nếu mà nhiều luồng tiêu thụ cùng lần lượt làm giảm items, điều đó tạm thời bị sai. Nhưng sẽ là sai lầm nếu cố gắng thực hiện đoạn code này (xảy ra Deadlock sẽ được nói ngay sau đây):

Broken consumer solution

```
mutex . wait ()
    items . wait ()
    event = buffer .get ()
mutex . signal ()
event . process ()
```

2.1.2 Deadlock

Deadlock là hiện tượng tranh chấp tài nguyên giữa hai hay nhiều lệnh trong đó lệnh này giữ tài nguyên mà lệnh kia cần dẫn tới việc không lệnh nào có thể kết thúc để giải phóng tài nguyên.

Xét lại đoạn code của luồng tiêu thụ như sau:

Broken consumer solution

```
1  mutex . wait ()
2      items . wait ()
3      event = buffer .get ()
4  mutex . signal ()
5  event . process ()
```

Hiện tương Deadlock có thể xảy ra là vì, nếu mà buffer đang rỗng, mà luồng tiêu thụ lại giành lấy mutex, rõ ràng bản thân nó sẽ phải chờ đợi mà luồng sản xuất lại không thể lấy được mutex để thực hiện tiếp. Đây là một lỗi phổ biến khi lập trình các vấn đề về đồng bộ. Nếu một luồng chờ đợi semaphore mà nó lại giữ mutex, deadlock sẽ có thể xảy ra.

2.1.3 Bài toán Producer-Consumer trong trường hợp buffer hữu hạn

Trong thực tế, buffer thường có kíc thước cố định. Khi này, xuất hiện một ràng buộc mới:
• Nếu buffer đã đầy mà luồng sản xuất lại muốn thực hiện, nó sẽ phải block cho đến khi ít nhất một item bị bỏ đi.

Ký hiệu bufferSize là "khả năng chứa" của buffer, items tiếp tục là semaphore lưu "số lượng item". Xét đoạn lệnh:

Broken finite buffer solution

```
1  if items >= bufferSize :
2      block ()
```

Điều này không được phép thực hiện bởi items là semaphore và thay vào đó, chúng ta chỉ có thể sử dụng thủ tục **wait** và **signal**.

Để giải quyết vấn đề, trước hết ta sẽ thêm một semaphore thứ hai là spaces, dùng để "lưu" số lượng item còn có thể thêm vào buffer. Mỗi khi luồng tiêu thụ lấy ra một items, nó phải signal spaces. Ngược lại với luồng sản xuất, nếu tạo ra items mới trong buffer thì phải giảm spaces và nó có thể sẽ block nếu spaces không dương cho đến khi luồng tiêu thụ làm tăng spaces.

Finite-buffer producer-consumer initialization

```
1  mutex = Semaphore (1)
2  items = Semaphore (0)
3  spaces = Semaphore ( buffer . size () )
```

2.1.4 Giải pháp cho bài toán trong trường hợp buffer hữu hạn

Ta sẽ có đoạn lệnh sau:

Finite buffer consumer solution

```
items . wait ()
mutex . wait ()
    event = buffer . get ()
mutex . signal ()
spaces . signal ()
event . process ()
```

Finite buffer producer solution

```
event = waitForEvent ()
spaces . wait ()
mutex . wait ()
    buffer . add ( event )
mutex . signal ()
items . signal ()
```

Để tránh deadlock, các luồng sẽ đều kiểm tra trạng thái của buffer trước khi giành lấy mutex. Đồng thời, cách tốt nhất để làm tăng hiệu năng tính đến thời điểm này là giải phóng mutex trước khi muốn thực hiện signal với items hay spaces.

2.2 Bài toán Readers and Writers

2.2.1 Phát biểu bài toán

Người đọc và người viết thực hiện các đoạn chương trình khác nhau trước khi vào phần tranh chấp (kí hiệu là **CS − critical section**). Có các ràng buộc sau:

1. Mọi người đọc đều có thể truy cập đồng thời
2. Nhà văn có quyền truy nhập độc quyền vào **CS**. Nói cách khác, nhà văn không thể truy nhập vào **CS** nếu có một luồng bất kì (nhà văn hoặc người viết) đang ở trong đó, và khi nhà văn ở trong **CS** thì không một luồng nào có quyền truy nhập.

Bài toán 1: Sử dụng semaphores để thực thi các ràng buộc này, đồng thời cho phép người đọc và người viết truy cập dữ liệu và tránh khả năng bị bế tắc.

2.2.2 Hướng giải bài toán

Dưới đây là tập hợp các biến dùng để giải bài toán này:

Readers-Writers initialization

```
int readers = 0
mutex = Semaphore (1)
roomEmpty = Semaphore (1)
```

Biến đếm **readers** theo dõi có bao nhiêu người đọc trong phòng. Biến **mutex** bảo vệ sự chia sẻ giữa các người đọc.

Biến **roomEmpty** bằng 1 nếu không có người đọc hay nhà văn nào ở trong phòng, bằng 0 nếu ngược

lại.

Đoạn chương trình cho nhà văn:

Write solution

```
roomEmpty . wait ()
critical section for writers
roomEmpty . signal ()
```

Đoạn chương trình cho người đọc:

Read solution

```
        mutex . wait ()
    readers += 1
    if readers == 1:
        roomEmpty . wait () # first in locks
    mutex . signal ()

    # critical section for readers

    mutex . wait ()
    readers -= 1
    if readers == 0:
    roomEmpty . signal () # last out unlocks
    mutex . signal ()
```

Biến **roomEmpty** đảm bảo việc truy cập độc quyền cho nhà văn. Khi một nhà văn truy cập vào **CS**, biến **roomEmpty** sẽ được khóa lại và các luồng khác sẽ không thể truy cập vào.

Ta có thể theo dõi số lượng người đọc trong phòng thông qua biến **readers**. Từ đó có thể đưa ra nhiệm vụ đặc biệt để đưa người đầu tiên đi vào và người cuối cùng rời đi.

Người đọc đầu tiên đến phải chờ đợi biến **roomEmpty** mở. Nếu phòng đang trống, người đọc được cấp quyền truy cập, trong cùng thời điểm đó, ngăn chặn nhà văn. Những người đọc tiếp theo vẫn có thể tham gia bởi họ không phải chờ **roomEmpty**.

Nếu người đọc đến trong khi đã có một nhà văn ở trong phòng, nó sẽ phải đợi ở biến **roomEmpty**. Vì nó chiếm giữ **mutex** nên những người đọc tiếp theo phải xếp hàng ở **mutex**.

Starvation

Trong ví dụ này, bế tắc không xảy ra. Tuy nhiên có một vấn đề khác là: nhà văn có thể bị bỏ đói (**starve**).

Nếu một nhà văn đến trong khi có người đọc trong **CS**, nó có thể phải xếp hàng chờ trong khi người đọc đến và đi. Miễn là người đọc mới đến trước khi người đọc cuối cùng trong số người đọc hiện tại rời đi, thì sẽ luôn có ít nhất một người đọc trong phòng.

Bài toán 2: Phát triển lời giải trên sao cho khi có một nhà văn đến, những người đọc hiện tại có thể hoàn thành, tuy nhiên không có thêm người đọc nào có thể vào.

2.2.3 Giải pháp bài toán No-starve readers-writers

Đoạn chương trình cho nhà văn:

No-starve write solution

```
turnstile . wait ()
roomEmpty . wait ()
# critical section for writers
turnstile . signal ()

roomEmpty . signal ()
```

Đoạn chương trình cho người đọc:

No-starve read solution

```
turnstile . wait ()
turnstile . signal ()

mutex . wait ()
readers += 1
if readers == 1:
    roomEmpty . wait ()
mutex . signal ()

# critical section for readers
mutex . wait ()
readers -= 1
if readers == 0:
    roomEmpty . signal ()
mutex . signal ()
```

Khi có một nhà văn đến trong khi có người đọc trong phòng, nhà văn sẽ phải chờ ở câu lệnh "$roomEmpty.wait()$", đồng nghĩa với đèn báo **turnstile** bị đóng. Nó ngăn chặn người đọc khác truy cập khi nhà văn đang trong hàng chờ.

Khi người đọc cuối cùng rời khỏi phòng, nó mở khóa **roomEmpty**, mở khóa nhà văn đang chờ. Nhà văn ngay lập tức có quyền truy cập vào **CS**, vì không có người đọc nào trong hàng chờ vượt qua được **turnstile**.

Khi người viết thoát ra khỏi **CS**, nó mở khóa **turnstitle**, mở khóa một luồng trong hàng chờ, có thể là người đọc hoặc nhà văn. Vì vậy, giải pháp này đảm bảo rằng ít nhất một nhà văn được tiếp tục, nhưng vẫn có thể cho một người đọc vào trong khi có những nhà văn đang trong hàng chờ.

Trong một số ứng dụng, người ta thường ưu tiên nhiều hơn cho nhà văn. Ví dụ, nếu nhà văn đang thực hiện để cập nhật dữ liệu mới, tốt hơn hết là nên giảm thiểu số lượng người đọc nhìn thấy dữ liệu cũ trước khi nhà văn có cơ hội thực hiện.

Bài toán 3: Viết lời giải cho bài toán Readers-Writers, trong đó ưu tiên cho người viết. Có nghĩa là, một khi người viết đến, không người đọc nào được phép vào cho đến khi tất cả người viết đã rời khỏi hệ thống.

2.2.4 Giải pháp bài toán Writer-priority readers-writers

Đoạn chương trình cho nhà văn:

Writer-piority writer solution

```
1   Wirtermutex . wait ()
2   writers += 1
3   if writers == 1:
4       noReaders . wait ()
5   Writermutex . signal ()
6   noWriters . wait ()
7
8   # critical section for writers
9
10  noWriters . signal ()
11  Writermutex . wait ()
12  writers -= 1
13  if writers == 0:
14      noReaders . signal ()
15  Writermutex . signal ()
```

Đoạn chương trình cho người đọc:

Writer-piority reader solution

```
1   noReaders . wait ()
2   Readermutex . wait ()
3   readers += 1
4   if readers == 1:
5       noWriters . wait ()
6   Readermutex . signal ()
7   noReaders . signal ()
8
9   # critical section for readers
10
11  Readermutex . wait ()
12  readers -= 1
13  if readers == 0:
14      noWriters . signal ()
15  Readermutex . signal ()
```

Nếu một người đọc ở trong **CS**, thì nó chiếm giữ **noWriters**, nhưng nó sẽ không chiếm giữ **noReaders**. Vì vậy, nếu có một nhà văn đến, nó có thể khóa **noReaders**, điều này sẽ khiến những người đọc tiếp theo phải xếp hàng. Khi người đọc cuối cùng thoát ra, nó mở đèn báo **noWriters**, cho phép bất kỳ nhà văn nào trong hàng chờ được tiếp tục.

2.3 No-starve mutex

2.3.1 Lời giới thiệu

Trong phần trước, chúng ta đã đề cập đến chết đói theo phân loại, trong đó một loại luồng (người đọc) cho phép một loại luồng khác (nhà văn) chết đói. Ở cấp độ cơ bản hơn, chúng ta phải giải quyết vấn đề chết đói luồng, đó là khả năng một luồng có thể chờ vô thời hạn trong khi những luồng khác tiếp tục.

Đối với hầu hết các ứng dụng đồng thời, việc chết đói là không thể chấp nhận được, vì vậy chúng ta thực thi yêu cầu chờ có giới hạn, có nghĩa là thời gian một luồng chờ trên semaphore (hoặc bất kỳ nơi nào khác, cho vấn đề đó) phải là hữu hạn.

Việc bỏ đói là trách nhiệm của bộ lập lịch. Bất cứ khi nào nhiều luồng sẵn sàng chạy, bộ lập lịch sẽ quyết định cái nào hoặc trên bộ xử lý song song, bộ luồng nào được chạy. Nếu một luồng không bao giờ được lên lịch, thì nó sẽ chết đói, bất kể chúng ta làm gì với semaphores.

Vì vậy, để nói bất cứ điều gì về nạn đói, chúng ta phải bắt đầu với một số giả định về bộ lập lịch. Nếu chúng ta sẵn sàng đưa ra một giả định chắc chắn, chúng ta có thể giả định rằng bộ lập lịch sử dụng một trong nhiều thuật toán có thể được chứng minh để thực thi việc chờ có giới hạn. Nếu chúng ta không biết thuật toán mà bộ lập lịch sử dụng, thì chúng ta có thể sử dụng với một giả định yếu hơn:

Thuộc tính 1: Nếu chỉ có một luồng sẵn sàng chạy, bộ lập lịch phải để nó chạy.

Nếu chúng ta có thể giả định Thuộc tính 1, thì chúng ta có thể xây dựng một hệ thống có thể không bị chết đói. Ví dụ, nếu có một số lượng hữu hạn luồng, thì bất kỳ chương trình nào chứa rào cản (**barrier**) đều không thể chết đói, vì cuối cùng tất cả các luồng trừ một luồng sẽ đợi ở rào cản, tại thời điểm đó luồng cuối cùng phải chạy. Mặc dù vậy, việc viết các chương trình không phải là điều tầm thường trừ khi chúng ta đưa ra giả định mạnh mẽ hơn:

Thuộc tính 2: Nếu một luồng đã sẵn sàng để chạy, thì thời gian nó đợi cho đến khi nó chạy bị giới hạn.

Trong định nghĩa của semaphore, chúng ta đã nói rằng khi một luồng thực thi tín hiệu, một trong các luồng đang chờ sẽ được đánh thức. Nhưng chúng ta chưa nói là cái nào. Để nói bất cứ điều gì về nạn đói, chúng ta phải đưa ra các giả định về hành vi của các semaphores. Giả định yếu nhất giúp bạn có thể tránh được nạn đói là:

Thuộc tính 3: Nếu có luồng đang chờ trên semaphore trong khi có một luồng thực thi mở khóa semaphore, thì một trong các luồng chờ phải được thức dậy.

Với Thuộc tính 3, có thể tránh được nạn đói, nhưng ngay cả đối với một thứ đơn giản như mutex, điều đó cũng không dễ dàng. Ví dụ với đoạn mã sau:

```
1  while True :
2      mutex . wait ()
3      # critical section
4      mutex . signal ()
```

Câu lệnh **while** là một vòng lặp vô hạn; nói cách khác, ngay sau khi một luồng rời khỏi đoạn găng, nó sẽ quay lại đầu vòng lặp và cố gắng lấy lại mutex. Giả sử rằng luồng A nhận được mutex và Chủ đề B và C chờ đợi. Khi A rời đi, B đi vào, nhưng trước khi B rời đi, A sẽ vòng qua và tham gia cùng C trong hàng đợi. Khi B rời đi, không có gì đảm bảo rằng C sẽ đi tiếp. Trên thực tế, nếu A đi tiếp và B tham gia vào hàng đợi, thì có thể lặp lại chu kỳ mãi mãi và C bị chết đói. Sự tồn tại của mô hình này chứng tỏ rằng **mutex** rất dễ bị chết đói. Một giải pháp cho vấn đề này là thay đổi việc triển khai **semaphore** để nó đảm bảo thuộc tính mạnh hơn:

Thuộc tính 4: Nếu một luồng đang đợi ở semaphore, thì số trong tổng số các chủ đề sẽ được đánh thức trước khi nó bị giới hạn.

Ví dụ: Nếu semaphore duy trì một hàng đợi FIFO (First-in-first-out), thì Thuộc tính 4 sẽ được bảo

đảm vì khi một luồng tham gia vào hàng đợi, số lượng các luồng phía trước nó là hữu hạn và không có luồng nào đến sau có thể đi trước nó. Một semaphore có Thuộc tính 4 đôi khi được gọi là **semaphore mạnh**; cái chỉ có Thuộc tính 3 được gọi là **semaphore yếu**. Chúng ta đã chỉ ra rằng với các semaphore yếu, giải pháp mutex đơn giản dễ bị chết đói. Trên thực tế, Dijkstra phỏng đoán rằng không thể giải quyết vấn đề mutex không bị chết đói nếu chỉ sử dụng các semaphore yếu .

Năm 1979, J.M.Morris bác bỏ phỏng đoán trên bằng cách tìm ra lời giải với giả thiết rằng số luồng là hữu hạn.

Phát biểu bài toán: Viết giải pháp cho vấn đề loại trừ lẫn nhau bằng cách sử dụng **semaphore yếu**. Giải pháp phải đáp ứng yêu cầu sau: Khi một luồng đến và cố gắng nhập mutex, sẽ có một ràng buộc về số lượng luồng có thể tiến hành trước nó. Có thể giả sử rằng tổng số luồng là hữu hạn.

2.3.2 Giải pháp cho bài toán

Khởi tạo các biến

No-starve mutex initialization

```
room1 = room2 = 0
mutex = Semaphore (1)
t1 = Semaphore (1)
t2 = Semaphore (0)
```

Các biến **room1** và **room2** theo dõi số luồng trong mỗi phòng chờ.

Giải pháp cho bài toán

No-starve mutex solution

```
mutex . wait ()
room1 += 1
mutex . signal ()

t1 . wait ()
room2 += 1
mutex . wait ()
room1 -= 1

if room1 == 0:
    mutex . signal ()
    t2 . signal ()
else :
    mutex . signal ()
    t1 . signal ()

t2 . wait ()
room2 -= 1

    # critical section

if room2 == 0:
```

```
23          t1 . signal ()
24      else :
25          t2 . signal ()
```

2.4 Dining Philosopher

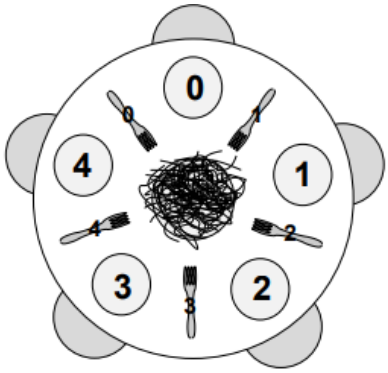
2.4.1 Giới thiệu bài toán

Bài toán triết gia ăn tối được giới thiệu bởi Dijiktra vào năm 1965, bài toán đề cập đến 5 vị triết gia đang ăn tối cùng nhau, mỗi người sẽ đến bàn ăn và thực hiện vòng lặp hành động như sau:

Vòng lặp hành động cơ bản của một triết gia

```
1   while (true) {
2       think();
3       getForks();
4       eat();
5       putForks();
6   }
```

Những cái nĩa là tài nguyên mà các luồng cần để có thể tiến hành thực hiện. Và một điểm tạo nên sự thú vị cho bài toán đó chính là, các nhà triết gia cần đến 2 cái nĩa để ăn, và nếu một người triết gia đói thì ông ấy cần phải đợi người bạn hàng xóm của mình ăn xong và đặt chiếc nĩa xuống.
Chúng ta đánh số những nhà triết gia từ 0 đến 4, và những cái nĩa cũng tương tự. Khi đó chúng ta có hình ảnh mô phỏng bài toán như sau:



Và nhiệm vụ của chúng ta chính là viết ra hai hàm **getForks** và **putForks** sao cho thỏa mãn được những yêu cầu sau:

- Chỉ một triết gia được cầm vào 1 cái nĩa tại một thời điểm
- Không được để hiện tượng deadlock xảy ra
- Nhà triết gia có thể đói trong khi chờ nĩa
- Có thể có nhiều hơn 1 nhà triết gia đang ăn tại một thời điểm
- Quá trình ăn và nghĩ sẽ diễn ra trong một khoảng thời gian hữu hạn, một nhà triết gia không thể cầm nĩa mãi mãi.

Để các nhà triết gia có thể dễ dàng lấy những chiếc nĩa, chúng ta tạo ra hai hàm sau:

Which fork?

```
def left(i): return i
def right(i): return (i + 1) % 5
```

Vì những chiếc nĩa là tài nguyên găng, chúng ta sẽ sử dụng đèn báo cho chúng.

```
forks = [Semaphore(1) for i in range(5)]
```

Sau khi đã có những giả thiết của bài toán, chúng ta đi đến với hướng giải đầu tiên như sau:

Solution 1

```
def getForks(i):
    fork[right(i)].wait()
    fork[left(i)].wait()

def putForks(i):
    fork[right(i)].signal()
    fork[left(i)].signal()
```

2.4.2 Deadlock

Theo cách giải thứ nhất, chúng ta có thể thấy nó thỏa mãn điều kiện 1, tuy nhiên điều kiện thứ hai là không thỏa mãn. Lý do là vì, giả sử tại cùng một thời điểm các nhà triết gia đều cầm lấy cái nĩa ở phía bên phải của mình, khi đó các nhà triết gia sẽ đợi chiếc nĩa còn lại mãi mãi (trong giả thiết của bài toán, các nhà triết gia sẽ lấy lần lượt đủ 2 chiếc nĩa rồi mới ăn).

2.4.3 Giải pháp 2

Nếu chỉ có 4 nhà triết gia ở trên bàn thì hiện tượng Deadlock sẽ không xảy ra nữa. Chúng ta sẽ kiểm soát số nhà triết gia ở trên bàn bằng cách khởi tạo một đèn báo **footman** có giá trị là 4.

Solution 2

```
def getForks(i):
    footman.wait()
    fork[right(i)].wait()
    fork[left(i)].wait()

def putForks(i):
    fork[right(i)].signal()
    fork[left(i)].signal()
    footman.signal()
```

2.4.4 Giải pháp 3

Chúng ta sẽ định nghĩa thứ tự lấy nĩa của các triết gia là khác nhau thỏa mãn: triết gia số hiệu chẵn lấy nĩa trái trước và triết gia số hiệu lẻ lấy nĩa phải trước.

Solution 3

```
    def getForks(i):
        j = i % 2
        fork[(i + j) % 5].wait()
        fork[(i + 1 − j) % 5].wait()

    def putForks(i):
        j = i % 2
        fork[(i + 1 − j) % 5].signal()
        fork[(i + j) % 5].signal()
```

2.4.5 Giải pháp của Tanenbaum

Đối với mỗi nhà triết gia, chúng ta sẽ định nghĩa một biến trạng thái để chỉ người đó đang ăn, đang nghĩ hay đang đợi để ăn và một đèn báo để chỉ nếu nhà triết gia có thể bắt đầu ăn. Sau đây là các biến và lời giải:

```
    state = ['thinking'] * 5
    sem = [Semaphore(0) for i in range(5)]
    mutex = Semaphore(1)
```

Tanenbaum's solution

```
    def getForks(i):
        mutex.wait()
        state[i] = 'hungry'
        test(i)
        mutex.signal()
        sem[i].wait()

    def putForks(i)
        mutex.wait()
        state[i] = 'thinking'
        test(right(i))
        test(left(i))
        mutex.signal()

    def test(i):
        if state[i] == 'hungry' and
        state[left(i)] != 'eating' and
        state[right(i)] != 'eating':
            state[i] = 'eating'
            sem[i].signal()
```

Có 2 cách để một nhà triết gia có thể bắt đầu ăn. Cách đầu tiên là người đấy thực hiện **getForks()**, tìm 2 nĩa và tiến hành ăn. Cách thứ hai là, khi hàng xóm đang ăn và nhà triết gia bị block bởi đèn báo. Sau khi người hàng xóm ăn xong, triết gia có thể ăn. Để có thê truy nhập biến trạng thái và thực hiện hàm test, chúng ta sử dụng thêm đèn báo **mutex** để kiểm soát. Lời giải này không gây ra hiện tượng deadlock.

2.5 Bài toán Cigarette Smoker

2.5.1 Giới thiệu bài toán

Bài toán Cigarette Smoker là được đề xuất lần đầu bởi **Suhas Patil** và ông cho rằng nó không thể được giải quyết bằng semaphores. Bài toán được phát biểu như sau:
Cho 4 luồng bao gồm: 1 luồng cho agent và 3 luồng cho người hút thuốc. Luồng của những người hút thuốc lặp vô hạn với các thao tác lần lượt: đợi nhiên liệu, tạo điếu thuốc và hút thuốc. Các nhiên liệu bao gồm *yobacco, paper, match*

Smoker

```
1  while(1) {
2      sem.wait();
3      makeCigarette();
4      sem.sinal();
5      smoke();
6    }
```

Chúng ta giả sử rằng có thể cung cấp vô hạn các nhiên liệu trong 3 nhiên liệu đó, mỗi người hút thuốc được cung cấp vô hạn 1 trong các nhiên liệu khác nhau: 1 người được cung cấp diêm, 1 người khác thì cung cấp giấy và người còn lại được cung cấp lá thuốc
Các agent liên tục lặp lại hành động cung cấp ngẫu nhiên 2 nhiên liệu trong các nhiên liệu và phụ thuộc vào nhiên liệu mà agent đã cung cấp, người hút thuốc khi chọn được 2 nguyện liệu tương ứng sẽ tiến hành xử lý.
Ví dụ, nếu 1 agent cung cấp 2 nhiên liệu tobacco và paper thì người hút thuốc đã có matche sẽ chọn 2 nguyện liệu này, làm điếu thuốc và sau đó lại tiếp tục báo hiệu tới agent.
Chúng ta có thể coi rằng agent giống như hệ điều hành phân bổ tài nguyên, và những người hút thuốc giống như những yêu cầu cần tài nguyên. Vấn đề đảm bảo rằng nếu tài nguyên có sẵn thì sẽ cho phép nhiều hơn 1 yêu cầu được thực hiện và tránh các yêu cầu không thể thực hiện.
Dựa vào điều trên thì có 3 phiên bản vấn đề thường xuyên xuất hiện trong các tài liệu:
The impossible version: Phiên bản của Patil áp đặt các hạn chế trên những giải pháp. Đầu tiên, chúng ta không được phép chỉnh sửa code của agent. Thứ 2 là không được dùng những câu điều kiện hoặng môt mảng các semaphore. Với những hạn chế này thì bài toán sẽ không thể giải quyết, nhưng như **Parnas** đã chỉ ra, hạn chế thứ 2 khá không tự nhiên.Bên cạnh đó, Với những hạn chế như thế này thì nhiều bài toán khác trở nên không thể giải quyết
The interesting version: Phiên bải này giữ lại hạn chế đầu tiên nhưng nó sẽ thay đổi những cái khác.
The trivial version: Trong vài tài liệu, vấn đề cụ thể rằng các agent nên gửi tín hiệu tới người hút thuốc phù hợp với các nhiên liệu đã được cung cấp. Vấn đề này là vấn đề không thú vị vì nó sẽ làm cho bài toán này không có gì đặc biệt và nó thực sự quá dễ.
Như một cách tự nhiên, chúng ta sẽ tập chung vào interesting version. Để hoàn thành các câu lệnh cảu bài toán, chúng ta cần đưa ra code của agent. Agent sử dụng những đoạn code semaphore sau:

Agent semaphores

```
1  agentSem = Semaphore (1)
2  tobacco = Semaphore (0)
3  paper = Semaphore (0)
4  match = Semaphore (0)
```

Agent thực tế tạo ra 3 luồng: Agent A, Agent B, Agent C. Mỗi luồng chờ đợi tín hiệu agentSem, mỗi lần có tín hiệu agentSem, 1 trong những Agent được đánh thức và cung cấp các nguyên liệu bằng các báo hiệu cho 2 semaphore.

Agent A code

```
1  agentSem.wait()
2  tobacco.signal()
3  paper.signal()
```

Agent B code

```
1  agentSem.wait()
2  paper.signal()
3  match.signal()
```

Agent C code

```
1  agentSem.wait()
2  tobacco.signal
3  match.signal()
```

Đây là một vấn đề khó vì giải pháp tự nhiên không hoạt động. Nó rất dễ gợi ý chúng ta viết chương trình như sau:

Smoker with matches

```
1  tobacco.wait()
2  paper.wait()
3  agentSem.signal()
```

Smoker with tobacco

```
1  paper.wait()
2  match.wait()
3  agentSem.signal()
```

Smoker with paper

```
1  tobacco.wait()
2  match.wait()
3  agentSem.signal()
```

2.5.2 Deadlock

Từ đoạn code trên ta thấy được rằng nếu xảy ra trường hợp chẳng hạn như agent đã cung cấp 2 nguyên liệu là tobacco và paper,2 luồng *Smoker with matches* và *Smoker with tobacco* đồng thời hoạt động, luồng *Smoker with matches* chạy câu lệnh tobacco.wait() và luồng *Smoker with tobacco* chạy câu lệnh paper.wait().Ssau câu lệnh đó, nguyên liệu tobacco và paper đã hết, 2 luồng trên sẽ tiếp tục chạy câu lệnh thứ 2 nhưng không thoát ra được và sẽ dẫn đến trường hợp chờ đợi vô hạn.
0.51cm]

2.5.3 Hướng giải bài toán Cigrette Smoker

Giải giáp này được đưa ra bải Parnas bằng cách sử dụng 3 luồng helper đươọc gọi là "pushers" cái mà phải hồi lại những tín hiệu từ agent, dựa vào các nguyên liệu có sẵn và báo hiệu tới người hút thuốc phù hợp.
Các biến và semaphore bổ sung là:

Smokers problem hint

```
isTobacco = isPaper = isMatch = False
tobaccoSem = Semaphore (0)
paperSem = Semaphore (0)
matchSem = Semaphore (0)
```

Các biến boolean thể hiện rằng có hay không nguyên liệu đó ở trên bàn. Những luồng *pusher* sử dụng biến **tobaccoSem** để báo hiệu người hút thuốc với tobacco và những tín hiệu semaphore khác cũng như vậy.

Giải quyết bài toán Cigarette Smoker

Đây là code cho 1 trong những pusher:

Pusher A

```
tobacco.wait()
mutex.wait()
if isPaper:
    isPaper = False
    matchSem.signal()
 else if isMatch:
    isMatch = False
    paperSem.signal()
else:
    isTobacco = True
mutex.signal()
```

Bằng code tương tự ta có Pusher B và Pusher C
Pusher này đánh thức bất cứ khi nào có tobacco ở trên bàn. Nếu nó thấy được isPaper = true, nó sẽ biết đươọc rằng Pusher B đã được chạy, vì vậy nó có thể tới *smoker with matches*. Tương tự nếu nó tìm thấy match trên bàn nó sẽ báo hiệu tới *smoker with paper*
Nhưng nếu Pusher A thực hiện đầu tiên, do isPaper = isMatch = False nên nó sẽ không có báo hiệu gì tới những người hút thuố vì vậy nó sẽ thiết lập isTobacco = True
Khi pusher thực hiện hết các công việc nên code của người hút thuốc khá đơn giản:

Smoker with tobacco

```
tobaccoSem.wait()
makeCigarette()
agentSem.signal()
smoke()
```

2.5.4 Tổng quát bài toán Cigarette Smoker

Parnas đề nghị rằng bái toán Cigarette Smoker trở nên khó hơn nó chúng ta điều chỉnh code của agent, xóa các yêu cầu chờ đợi của agent sau khi cung cấp nguyên liệu. Trong trường hợp này có thể có

nhiều nguyên liệu được cung cấp để trên bàn.
Chúng ta sẽ điều chỉnh giải pháp trước để xử lý các biến này
Nếu các agent không đợi những người hút thuốc, các nguyên liệu có thể tích tụ trên bàn. Thay vf sử dụng các biến boolean để biết được sự tồn tại của nguyên liệu, chúng ta sẽ sử dụng các biến iteger để đếm chung.

Generalized Smokers problem hint

```
numTobacco = numPaper = numMatch = 0
```

2.5.5 Giải pháp cho bài toán tổng quát Cigarette Smoker

Đây là code đã được sửa đổi cho Pusher A

Pusher A

```
tobacco.wait()
mutex.wait()
if isPaper:
    numPaper −= 1
    matchSem.signal()
else if numMatch:
    numMatch −= 1
    paperSem.signal()
else:
    numTobacco += 1
mutex.signal()
```

3 Tổng kết

Đèn báo là một trong những công cụ quan trọng để giải quyết vấn đề đồng bộ hóa giữa các tiến trình. Mặc dù có rất nhiều bài toán thú vị xoay quanh đèn báo, nhưng trong thời gian có hạn của kỳ học 2020-2 này, nhóm chỉ mới có thể tìm hiểu được 5 bài toán tiêu biểu nhất như trên, 5 bài toán này được dịch từ chương 4 của sách [1].

Để hoàn thành được báo cáo này, nhóm chúng em xin tỏ lòng biết ơn sâu sắc đến thầy Phạm Đăng Hải, cảm ơn thầy vì những tiết dạy tuyệt vời và đáng quý của thầy ở môn Nguyên lý hệ điều hành.

Toàn bộ source code minh họa cho các bài toán của nhóm được đặt ở đây: link.

Tài liệu

[1] Allen Downey. *The little book of semaphores*, volume 2. Green Tea Press, 2008.