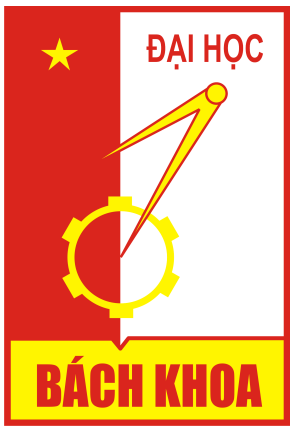

Research Project

NGUYÊN LÝ HỆ ĐIỀU HÀNH

ĐỀ TÀI

Tổng quan về Semaphores

Cao Như Đạt - Trịnh Tùng Dương - Nguyễn Trung Hiếu - Nguyễn Huy Hoàng - Lê Quốc Hưng - Nguyễn Văn Thành
(Chương trình Tài năng Khoa học máy tính K64)

Mục lục

1 Tổng quan về Semaphore

1.1 Khái niệm đồng bộ hóa

Trong cuộc sống, từ đồng bộ dùng để chỉ hai hay nhiều sự kiện xảy ra đồng thời. Trong công nghệ thông tin, từ "đồng bộ" mang nghĩa rộng hơn, nó dùng để chỉ mối quan hệ về mặt thời gian giữa những sự kiện trong máy tính (trước, trong và sau).
Trong hệ thống máy tính một vi xử lí, nếu hệ thống chỉ chạy một chương trình từ câu lệnh này sang câu lệnh khác thì việc đồng bộ hóa là không cần thiết vì chỉ cần nhìn và chương trình ta có thể xác định được lệnh nào trước lệnh nào sau. Song trong các trường hợp khác, ví dụ như máy tính xử lí đa lõi hoặc cùng trên chương trình có nhiều luồng chạy đồng thời thì việc đồng bộ hóa trở nên cần thiết vì lập trình viên không biết được thứ tự các câu lệnh sẽ được thực thi như thế nào, hệ điều hành sẽ quyết định việc đó.
Việc đồng bộ các tiến trình càng trở nên quan trọng hơn khi các tiến trình sử dụng chung các tài nguyên (được gọi là tài nguyên găng). Do khi nhiều tiến trình cùng truy cập tài nguyên một lúc có thể dẫn đến không đảm bảo tính toàn vẹn dữ liệu và kết quả có thể sai lệch hoặc không đúng với mong muốn.

1.2 Khái niệm đèn báo

Trong lĩnh vực khoa học máy tính, đèn báo là một biến hay là một kiểu dữ liệu trừu tượng dùng để kiểm soát quyền truy nhập vào tài nguyên găng của nhiều tiến trình hay hệ thống hoạt động đồng thời. Đèn báo được sử dụng để giải quyết các vấn đề liên quan đến tài nguyên găng.
Semaphore được tạo ra bởi Edsger Dijkstra, một nhà khoa học lỗi lạc trong lĩnh vực khoa học máy tính. Trong suốt quá trình phát triển của Semaphore đã có nhiều thay đổi so với bản gốc nhưng ý tưởng thì vẫn giữ nguyên. Về cơ bản Semaphore giống như một biến số nguyên với những đặc điểm khác biệt sau đây:
1. Khi khởi tạo Semaphore, ta có thể gán cho nó bất kì giá trị nguyên nào, nhưng sau đó chỉ có các tiến trình mới có thể thay đổi giá trị của nó, có thể (tăng lên một) hoặc giảm (giảm đi một).
2. Khi một tiến trình giảm giá trị của Semaphore, nếu giá trị trả về là âm, tiến trình sẽ tự block chính nó và không thể nào tiếp tục cho đến khi có một tiến trình khác tăng giá trị của Semaphore
3. Khi một tiến trình tăng giá trị của Semaphore, nếu có những tiến trình khác đang đợi thì một trong số đó sẽ được unblock.

Ý nghĩa giá trị của Semaphore đó là nếu giá trị này dương thì sẽ cho biết số tiến trình có thể giảm Semaphore mà không bị block. Còn nếu giá trị âm nó cho ta biết số tiến trình đã bị block và đang chờ đợi. Nếu giá trị bằng 0 tức là

không có tiến trình nào đang chờ, nhưng nếu có một tiến trình giảm giá trị của Semaphore, nó sẽ bị block.

1.3 Lợi ích và bất lợi khi sử dụng Semaphore

Semaphore có những ưu điểm như: chỉ cho phép một tiến trình truy cập vào tài nguyên găng, chúng tuân theo nguyên tắc loại trừ lẫn nhau một cách nghiêm ngặt và hiệu quả hơn nhiều so với một số phương pháp điều độ cấp thấp khác như phương pháp khóa trong hay kiểm tra và xác lập; không có sự lãng phí tài nguyên hay thời gian của vi xử lý trong khi các tiến trình chờ đợi để truy cập vào đoạn găng; semaphore được triển khai trong mã máy độc lập của kênh vi mô, vì vậy, chúng độc lập với máy móc.

Tuy nhiên, khi sử dụng semaphore cho điều độ tiến trình, cũng có một số khó khăn như: semaphore rất phức tạp, vì vậy các hoạt động chờ đợi và tín hiệu phải được thực hiện theo đúng thứ tự để tránh bế tắc; semaphore là không thực tế cho việc sử dụng quy mô cuối cùng vì việc sử dụng chúng dẫn đến mất tính modulo, điều này xảy ra bởi vì các hoạt động chờ đợi và tín hiệu ngăn cản việc tạo ra một bố cục có cấu trúc cho hệ thống; semaphore có thể dẫn đến sự đảo ngược mức độ ưu tiên khi các quá trình có mức độ ưu tiên thấp có thể truy cập vào đoạn găng trước các quá trình có mức độ ưu tiên cao.

2 Tìm hiểu bài toán điều độ tiến trình

2.1 Bài toán người hút thuốc lá

Phát biểu bài toán:

Đây là một trong những bài toán kinh điển về tiến trình được đưa ra vào năm 1971 bởi Suhas Patil. Bài toán được phát biểu như sau: Có bốn luồng tham gia trong bài toán này: một luồng đóng vai trò là đại lý và ba luồng còn lại đóng vai trò là người hút thuốc lá. Người hút thuốc lá sẽ lặp đi lặp lại các công đoạn: chờ đợi nguyên vật liệu đến từ phía đại lý, sản xuất ra điếu thuốc lá và hút điếu thuốc lá đó. Nguyên vật liệu bao gồm có ba loại là lá thuốc (tobacco), giấy (paper) và diêm (matches).
Giả sử rằng người đại lý có thể cung cấp cả ba loại nguyên vật liệu với số lượng vô hạn còn mỗi người hút thuốc chỉ sở hữu một trong ba loại nguyên vật liệu với số lượng vô hạn. Ví dụ một người chỉ có lá thuốc, người kia chỉ có giấy và người cuối cùng chỉ có diêm.
Người đại lý sẽ liên tục chọn ngẫu nhiên hai trong 3 nguyên liệu và đặt lên trên bàn. Lúc đó thì người hút thuốc sở hữu nguyên liệu còn lại sẽ lấy hai nguyên liệu trên bàn và tiến hành các công đoạn còn lại.
Ví dụ khi người đại lý đạt giấy và lá thuốc lên trên bàn, người hút thuốc Sở hữu diêm sẽ lấy hai nguyên vật liệu đó, làm ra điếu thuốc và báo lại với người đại lý.
Trong bài toán này người đại lý tượng trưng cho hệ điều hành phân phối tài nguyên và mỗi người hút thuốc đại diện cho một ứng dụng yêu cầu những tài nguyên đó để vận hành.
Có ba phiên bản của bài toán thường được đề cập đến:
The impossible version: Đây là phiên bản được Patil đề xuất là vào năm 1971 đi kèm với hai ràng buộc:
1. Không được can thiệp vào mã nguồn của người đại lý.
2. Không được sử dụng các câu lệnh điều kiện hoặc sử dụng mảng các đèn báo (array of semaphores).
Với hai điều kiện trên, bài toán được khẳng định là không thể nào giải được. Ràng buộc thứ nhất là hoàn toàn hợp lý do người đại lý đại diện cho hệ điều hành, do đó không thể bị thay đổi khi có một ứng dụng mới trong hệ thống. Song ràng buộc thứ hai lại mang tính chất không bắt buộc.
The interesting version: Đây là phiên bản trong đó ta giữ lại ràng buộc thứ nhất và bỏ đi ràng buộc thứ hai.
The trivial version: Trong một số sách, người đại lý sẽ ra hiệu (bằng thủ tục signal() của semaphore) cho người hút thuốc sẽ nhận nguyên vật liệu tiếp theo. Tuy nhiên phiên bản này làm mất đi ý nghĩa của bài toán vì nó xóa đi mối liên hệ giữa nguyên liệu và điếu thuốc lá. Hơn nữa khi đặt trong điều kiện thực tế, một hệ điều hành thường không biết trước được về những luồng chạy trong nó và những tài nguyên được yêu cầu bởi những luồng này.

Vì vậy chúng ta sẽ đi tìm hiểu phiên bản thứ hai của bài toán.

Cách giải

Với người đại lý chúng ta sẽ sử dụng 4 đèn báo:

```
agentSem = Semaphore (1)
tobacco = Semaphore (0)
paper = Semaphore (0)
match = Semaphore (0)
```

Có thể nhận thấy rằng người đại lý thực ra được cấu tạo bởi 3 luồng chạy song song:
Luồng A:

```
agentSem.wait()
tobacco.signal()
paper.signal()
```

Luồng B:

```
agentSem.wait()
paper.signal()
match.signal()
```

Luồng C:

```
agentSem.wait()
tobacco.signal()
match.signal()
```

Khi đó, mỗi luống này sẽ nhận tín hiệu từ đèn báo của người đại lý (agentSem) và cung cấp hai nguyên vật liệu cần thiết bằng cách sử dụng phương thức signal() để truyền tín hiệu đến hai đèn báo của những vật liệu đó. Bài toán này khó bởi vì nếu chúng ta sử dụng cách giải thông thường sẽ dẫn đến tình trạng hệ thống rơi vào trạng thái bế tắc (Deadlock). Ví dụ thông thường chúng ta sẽ viết Code của ba người hút thuốc lần lượt như sau:

Lời giải chưa đúng:
Người hút thuốc có diêm:

```
tobacco.wait()
paper.wait()
agentSem.signal()
```

Người hút thuốc có lá thuốc:

```
paper.wait()
match.wait()
agentSem.signal()
```

Người hút thuốc có giấy:

```
tobacco.wait()
match.wait()
agentSem.signal()
```

Song, để ý thấy rằng hai câu lệnh chờ nguyên vật liệu một và hai là các câu lệnh xử lý tách rời nhau. Giả sử khi người đại lý đặt lá thuốc và giấy ở trên bàn, như vậy người hút thuốc có diêm đang ở câu lệnh "tobacco.wait()" sẽ được chuyển từ trạng thái chờ đợi sang sẵn sàng. Nhưng đồng thời ngay lúc đó người hút thuốc có lá thuốc có thể đang ở câu lệnh "paper.wait() và được đánh thức. Kết quả là cả hai người đều bị block ở câu lệnh "match.wait()". Hệ thống rơi vào bế tắc.

Lời giải đúng:
Lời giải này được đưa ra bởi David Parnas, bằng cách sử dụng các luồng trợ giúp gọi là "pusher" các luồng này sẽ phản hồi các tín hiệu tới từ phía người đại lý, theo dõi các nguyên liệu có trên bàn và gửi tín hiệu đánh thức người hút thuốc. Code cho "pusher A":

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

Trong đó 3 biến isPaper, isMatch, isTobacco là ba biến nhị phân cho biết nguyên liệu tương ứng có ở trên bàn hay không và đèn báo "tobaccoSem = Semaphore(0) dùng để đánh thức người hút thuốc lá có lá thuốc (tobacco), tương tự đối với "matchSem" và "paperSem".
Mỗi khi có lá thuốc được đặt trên bàn, "Pusher A" sẽ được đánh thức và bắt đầu vào kiểm tra. Nếu nó tìm thấy giấy có trên bàn (isPaper = True) thì nó sẽ

lấy giấy (gán isPaper = False) và gửi tín hiệu đến người hút thuốc có diêm.
Nếu "Pusher A" chạy trước khi hai "Pusher" còn lại chạy (tức là is Match và isPaper lúc này đều có giá trị là False) nó sẽ rơi vào trường hợp thứ 3, gán isTobacco = True và rời khỏi đoạn găng. Toàn bộ quá trình trên là đoạn găng vì khi người đại lý phân phối hai loại tài nguyên cùng một lúc thì có thể cả hai "Pusher" cùng được đánh thức. Do đó ta cần đặt đoạn chương trình trong cụm đèn báo mutex để đảm bảo chỉ một "Pusher" có thể thay đổi các biến nhị phần tại một thời điểm.
Và sau đó ta có Code của người hút thuốc Sở hữu là thuốc:

```
tobaccoSem.wait()
makeCigarette(1)
agentSem.signal()
smoke()
```

Bài toán người hút thuốc tổng quát

Parnas có đề xuất bài toán người hút thuốc tổng quát, trong đó người đại lý liên tục cung cấp các nguyên vật liệu mà không chờ cho đến khi người hút thuốc gửi tín hiệu lại.
Như vậy ta chỉ cần thay đổi các biến nhị phân isTobacco, isPaper, isMatch thành các biến kiểu nguyên cho biết số nguyên vật liệu có trên bàn. Code cho "pusher A" trong trường hợp tổng quát:

```
tobacco.wait()
mutex.wait()
            if numPaper > 0:
                    numPaper -= 1
                    matchSem.signal()
            else if numMatch > 0:
                    numMatch -= 1
                    paperSem.signal()
            else:
                    numTobacco += 1
mutex.signal()
```

2.2 Bài toán chăm sóc trẻ con

Phát biểu bài toán:

Max Hailperin đã viết vấn đề này cho sách Hệ điều hành và Mid-dleware. Tại một trung tâm chăm sóc trẻ em, các quy định của nhà nước yêu cầu: mỗi một người lớn thì trông 3 ba trẻ em. Hãy viết luồng người lớn và luồng trẻ em thỏa mãn.

Cách giải:

Hailporin gợi ý rằng có thể giải quyết vấn đề này bằng semaphore.

```
multiplex = Semaphore (0)
```

Biến multiplex đếm số lượng mã thông báo có sẵn, mỗi thông báo cho phép 1 luồng trẻ em để vào. Khi người lớn vào, ra tín hiệu multiplex 3 lần cũng như khi người lớn rời đi (đợi 3 lần).
Lời giải sai:
Code cho người lớn:

```
multiplex.signal(3)
#critical section
multiplex.wait()
multiplex.wait()
multiplex.wait()
```

Lời giải này có thể dẫn tới bế tắc (deadlock). Giả sử có 3 đứa trẻ và 2 người lớn. Như vậy một trong hai người lớn có thể rời đi. Nhưng có thể cả 2 người lớn có thể cùng rời đi. Họ sẽ lần lượt gửi mutiplex.signal() và kết quả là hai người sẽ bị chờ ở câu lệnh mutiplex.wait() và cả 2 block.

Lời giải đúng:
Code cho người lớn:

```
multiplex signal (3)
#critical section
mutex.wait()
        multiplex.wait()
        multiplex.wait()
        multiplex.wait()
mutex.signal()
```

Bây giờ cả 3 câu lệnh multiplex.wait() đều được xử lí đồng thời. Nếu như có 3 thông báo thì chỉ một luồng nhận được cả 3 mã thông báo và thoát. Nếu có ít mã thông báo hơn thì luồng đầu tiên sẽ block trong mutex và luồng tiếp theo sẽ xếp hàng đợi để có quyền truy cập mutex.

Vấn đề chăm sóc trẻ em mở rộng

Vấn đề: Một đặc điểm của bài toán này này là một luồng người lớn đang chờ rời đi có thể ngăn luồng trẻ em vào.
Ví dụ: Có 4 trẻ em và 2 người lớn vì vậy giá trị của multiplex là 2. Nếu một trong những người lớn cố gắng rời đi, cô ấy sẽ lấy 2 mã thông báo và sau đó block. Nếu một luồng trẻ em đến, nó sẽ phải đợi mặc dù nó được vào.
Cách giải:

```
children = adults = waiting = leaving = 0
mutex = Semaphore (1)
childQueue = Semaphore (0)
adultQueue = Semaphore (0)
```

Các biến đếm children, adults, waiting, leaving theo dõi số lượng trẻ em, người lớn, số trẻ em đợi để vào và số người lớn chờ đợi để rời đi. Tất cả được bảo vệ bởi mutex. Children chờ trên childQueue để vào nếu thật sự cần thiết. Adults chờ vào adultQueue để rời đi.
Code cho trẻ em:

```
mutex.wait()
         if children < 3 * adults:
               children++
         mutex.signal()
         else :
         waiting++
         mutex.signal()
         childQueue.wait()
# critical section
mutex.wait()
        children --
        if leaving and children <= 3 * (adults -1):
               leaving --
               adults --
               adultQueue.signal()
mutex.signal()
```

Khi trẻ em đi vào trung tâm, chúng kiểm tra xem có đủ người lớn hay không hay và xảy ra hai trường hợp
1. Nếu có đủ người lớn thì chúng sẽ tăng biến đếm children lên 1 và đi vào trung tâm.
2. Nếu không chúng sẽ chờ đợi.
Khi trẻ em rời đi, chúng kiểm tra xem có người lớn nào đang đợi không và sẽ gửi tín hiệu đến nếu có thể.
Code cho người lớn:

```
mutex.wait()
     adults++
     if waiting :
        n = min (3, waiting)
        child Queue.signal(n)
        waiting -= n
        children += n
mutex.signal()
# critical section
mutex.wait()
```

```
        if children <= 3(adults 1):
            adults --
            mutex.signal()
        else:
            leaving++
            mutex.signal()
            adultQueue.wait()
```

Khi người lớn vào, họ cho gửi tín hiệu cho trẻ em chờ đợi, nếu có. Trước khi họ rời đi, họ kiểm tra xem có đủ người lớn còn lại không. Nếu là như vậy, họ giảm biến adults đi 1 và thoát. Nếu không, họ tăng leaving và block. Trong khi một luồng người lớn đang chờ để rời đi, nó được tính là một trong những người lớn trong đoạn găng, vì vậy trẻ em bổ sung có thể vào trong trung tâm.

2.3 Bài toán phòng họp

Bài toán: Viết mã đồng bộ hóa cho sinh viên và trưởng khoa thực thi tất cả các ràng buộc sau:
• Trong phòng có thể chứa vô hạn sinh viên.
• Trưởng khoa chỉ có thể vào phòng nếu không có sinh viên bên trong hoặc trong phòng có hơn 50 sinh viên(sinh viên đang tổ chức tiệc).
• Trong khi trưởng khoa đang ở trong phòng, không có sinh viên nào có thể vào, nhưng sinh viên có thể rời đi.
• Trưởng khoa sinh viên không được rời khỏi phòng cho đến khi tất cả các sinh viên đã rời đi.
• Chỉ có một trưởng khoa.

```
students = 0
dean = "not here"
mutex = Semaphore (1)
turn  = Semaphore (1)
clear = Semaphore (0)
lieIn = Semaphore (0)
```

Chúng ta sẽ sử dụng các biến sau:
• students đếm số lượng sinh viên trong phòng
• dean là trạng thái của trưởng khoa (“waiting” or “in the room”)
• mutex bảo vệ students và dean
• turn giúp giữ sinh viên không vào khi trưởng khoa đang ở trong phòng.
• clear và lieIn được sử dụng như một điểm hẹn giữa sinh viên và trưởng khoa

Sau đây là bài giải của bài toán trên:
Lời giải cho luồng trưởng khoa:

```
mutex.wait()
```

```
        if students > 0 and students < 50:
                dean = "waiting"
                muttex.signal()
                lieIn.wait()  # and get mutex from the student
        # students must be 0 or >= 0
        if students >= 50:
                dean = "in the room"
                breakup()
                turn.wait()   # lock the turnstile
        else:
            search()
dean = "not here"
mutex.signal()
```

Khi mà trưởng khoa đến, sẽ xảy ra 3 trường hợp sau: nếu số lượng sinh viên trong phòng nhỏ hơn 50 và lớn hơn 0 thì trưởng khoa sẽ phải đợi, nếu trong phòng có ít nhất 50 sinh viên thì trưởng khoa sẽ chấm dứt bữa tiệc và đợi cho tất cả sinh viên ra ngoài, và cuối cùng là nếu không có sinh viên nào trong phòng, trưởng khoa sẽ tìm kiếm và rời đi.
Ở trong 2 trường hợp đầu tiên, trưởng khoa phải đợi cho một điểm hẹn với sinh viên, sau đó anh ấy sẽ phải từ bỏ mutex để tránh bế tắc. Khi mà trưởng khoa đã chờ đợi xong, thì anh ấy lại cần lại mutex.

2.4 Bài toán quầy sushi

Phát biểu bài toán:

Bài toán này dựa trên một vấn đề được đưa ra bởi Kenneth Reek vào năm 2004. Nó được phát biểu như sau:
Trong một quầy sushi có 5 chiếc ghế, nếu như có một vị khách đến vào lúc quầy sushi có một ghế trống thì người đó có thể ngồi cái ghế đó ngay lập tức. Nhưng nếu vị khách đó đến khi cả 5 chiếc ghế đều có người ngồi thì có nghĩa là trong quán đang có 5 người đang dùng bữa cùng nhau và vị khách mới tới sẽ phải chờ cho tới khi cả 5 người ở trong quán rời đi thì họ mới có thể vào ngồi.

Cách giải:

Ta sẽ sử dụng các biến cùng giá trị khởi tạo như sau:

```
eating = waiting = 0
mutex = Semaphore (1)
block = Semaphore (0)
must_wait = False
```

Trong đó hai biến eating và waiting để chỉ số khách đang ăn và đợi ở quầy sushi,

do hai biến này là tài nguyên căng nên sẽ được bảo vệ bởi đèn báo mutex. Biến nhị phân must_wait dùng để cho biết rằng quầy sushi có đang đầy khách hay không, biến được gần bằng True nếu quán đầy khách và vị khách mới đến sẽ phải đợi, bằng False nếu ngược lại.

Lời giải sai:
Code cho khách hàng:

```
mutex.wait()
if must_wait:
          waiting += 1
          mutex signal()
          block. wait()
          mutex.wait()
          waiting -= 1
eating += 1
must_wait = (eating == 5)
mutex.signal()

# eat sushi

mutex.wait()
eating -= 1
if (eating == 0)
           n = min (5, waiting)
           block.signal(n)
           must_wait = False
mutex.signal()
```

Khi một vị khách đến quầy sushi, vị khách này sẽ chờ đèn báo mutex để tiến vào thay đổi biến waiting và biến eating. Nếu quán đầy thì waiting sẽ tăng lên 1 và trả lại đèn báo mutex cho các vị khách khác và vị khách này sẽ bị block(). Còn nếu không thì eating sẽ tăng lên 1. Vị khách cuối cùng rời đi, nếu có ít hơn 5 vị khách đang chờ đợi thì sẽ đánh thức tất cả, còn không thì chỉ đánh thức 5 vị khách và đặt lại biến must_wait về False.
Tuy nhiên vấn đề ở đây là sau khi vị khách cuối cùng trong quán rời đi và đánh thức các vị khách đang chờ, những vị khách này muốn vào trong quản thì sẽ phải tranh giành đèn báo mutex cùng với những vị khách vừa mới tới. Có thể xảy ra trường hợp các vị khách mới tới sẽ lấy hết ghế trước những vị khách đang chờ. Dẫn đến có thể có tới hơn 5 luồng cùng hoạt động trong đoạn găng. Điều này vi phạm vào điều kiện loại trừ lẫn nhau trong điều độ tiến trình.

Lời giải đúng:
Khi đưa ra bài toán, Kenneth Reek cũng đưa ra hai lời giải cho bài toán này.
Lời giải thứ nhất:
Lý do mà mỗi vị khách tới đều yêu cầu vào đoạn găng thông qua quyền kiểm

soát của mutex là để thay đổi tài nguyên găng (hai biến eating và waiting). Cách để giải quyết vấn đề này đó chính là ta sẽ để vị khách cuối cùng rời đi làm điều này. Đoạn code có dạng như sau:

```
mutex.wait()
if must_wait:
          waiting += 1
          mutex.signal()
          block.wait()
else
          eating += 1
          must_wait = (eating = 5)
          mutex.signal()

# eat sushi

mutex.wait()
eating -= 1
if (eating == 0)
         n = min (5, waiting)
         waiting -=n
         eating += n
         must_wait = (eating = 5)
         block. signal(n)
mutex. signal()
```

Khi vị khách cuối cùng rời đi, người đó kiểm tra xem có bao nhiêu vị khách đang đợi, từ đó thay đổi trạng thái của quầy sushi (đầy khách hay chưa). Như vậy khi các vị khách mới đến, họ sẽ không thể nào tranh quyền kiểm soát mutex với những vị khách đang chờ do lúc đó biến must_wait = False và chỉ những vị khách đang chờ được đánh thức bởi câu lệnh block.signal(n) mới có thể vào quán. Reek gọi phương pháp này là "I'll do it for you"(Tôi sẽ giúp bạn làm điều đó) bởi lẽ luồng chuẩn bị thoát ra khỏi đoạn găng làm công việc mà lẽ ra những luồng đang chờ đợi phải làm.
Nhược điểm của phương pháp này chính là ta khó có thể xác định liệu trạng thái của quầy sushi có được cập nhật đúng hay không.

Lời giải thứ hai:
Lời giải thứ hai do Reek đề xuất được dựa trên khái niệm rằng ta có thể chuyển quyền kiểm soát đèn báo mutex từ luồng này sang luồng khác. Nghĩa là mutex có thể bị khóa bởi một luồng và được mở bởi một luồng khác. Miễn là cả hai luồng đều ý thức được rằng đèn báo mutex đã được chuyển qua lại giữa các luồng. Đoạn code có dạng như sau:

```
mutex.wait()
if must_wait:
          waiting += 1
```

```
          mutex.signal()
          block.wait()
          waiting -= 1
eating +=1
must_wait = (eating = 5)
if waiting and not must_wait
          block. signal()
else:
          mutex.signal()
# eat sush
mutex.wait()
eating -= 1
if eating == 0
must_wait = False
if waiting and not must wait :
block signal()
else:
       mutex signal()
```

Nếu có ít hơn 5 khách đang ở trong quầy sushi và không có khách nào chờ đợi, một vị khách mới đến sẽ chỉ tăng chỉ số eating lên 1 và sau đó giải phóng đèn báo mutex. Người khách hàng cuối cùng sẽ gán biến must_wait = True.
Nếu biến must_wait bằng True, những vị khách mời đến sẽ bị block cho đến khi vị khách cuối cùng rời khỏi quầy sushi, đặt biến must_wait về False và bảo hiệu với đèn báo block qua câu lệnh block.signal().
Khi một vị khách được đánh thức nó tiếp tục tại câu lệnh block.wait() và hiện tại đang kiểm soát đèn báo mutex. Nếu có các vị khách khác đang đợi và quán sushi chưa đầy khách (qua câu lệnh if waiting and not must_wait) thì vị khách này sẽ chuyển đèn báo mutex cho vị khách tiếp theo qua lệnh block.signal() và quá trình này tiếp diễn cho đến khi không còn vị khách nào đợi hoặc quán kín chỗ. Vị khách cuối cùng sẽ giải phóng đèn báo mutex và ngồi xuống ăn sushi.
Reek gọi phương pháp này là "Pass the baton" (Chuyền gậy) vì quyền kiểm soát đèn báo mutex được chuyển từ vị khách này sang vị khách khác như chiếc gậy trong một cuộc thi chạy tiếp sức. Ưu điểm của phương pháp này là dễ dàng để cập nhật các biến eating và waiting cho đúng, nhược điểm là khiến cho lập trình viên khó có thể chắc chắn rằng đèn báo mutex được sử dụng đúng.

2.5 Bài toán ông già Noel

Phát biểu bài toán:

Bài toán này được nêu ra bởi John Trono của đại học St.Michael và được nêu trong cuốn sách Operating Systems: Internals and Design Principles của William Stallings. Bài toán được phát biểu như sau:

Ông già Noel đang ngủ trong xưởng đồ chơi của mình tại Bắc Cực và chỉ có thể bị đánh thức khi một trong hai sự kiện sau đây xảy ra:
1.Tất cả 9 chủ tuần lộc đều quay trở về từ chuyến du lịch ở phía nam Thái Bình Dương.
2.Những chú yêu tinh gặp khó khăn trong việc làm ra những món đồ chơi.
Để cho ông già Noel có thể nghỉ ngơi, chỉ khi nào có 3 chú yêu tinh cùng gặp rắc rối thì mới được đánh thức ông dậy. Đồng thời, trong khi 3 chú yêu tinh này được ông già Noel giải quyết khó khăn của mình thì những chú yêu tinh khác muốn gặp ông già Noel phải đợi cho tới khi 3 chú yêu tinh kia trở về. Nếu ông già Noel tỉnh dậy và thấy có 3 chú yêu tinh đang đứng đợi ở cửa, nhưng cùng lúc đó thì chú tuần lộc cuối cùng trở về từ vùng nhiệt đới, ông sẽ để cho những chú yêu tinh đợi cho tới khi qua Giáng Sinh và đi chuẩn bị xe kéo của mình (giả sử rằng tuần lộc không muốn đi khỏi xứ nhiệt đới, và chỉ trở về vào lúc cận kề Giáng Sinh). Chú tuần lộc cuối cùng phải gọi ông già Noel trong khi những chú tới trước sẽ sưởi ấm, chờ đợi để được móc vào xe kéo.
Một số dữ kiện thêm vào bài toán:
1. Khi cả 2 chú tuần lộc tới thì ông già Noel sẽ đi chuẩn bị xe kéo qua phương thức prepareSleigh, cùng lúc đó thì cả 2 chú tuần lộc sẽ được nối vào xe kéo qua phương thức getHitched
2. Sau khi 3 chú yêu tinh tới thì ông già Noel sẽ sử dụng phương thức helpElves, đồng thời 3 chú yêu tinh sẽ nhận được sự trợ giúp getHelp
3. Cả 3 chú yêu tinh đều phải vào trạng thái getHelp trước khi có thêm một chú yêu tinh nào đó tới.

Ông già Noel sẽ làm việc này liên tục (sử dụng vòng lặp vô hạn) để giúp nhiều nhóm yêu tinh khác nhau. Giả sử chỉ có 9 chú tuần lộc nhưng số lượng yêu tinh không được biết trước.

Cách giải:

Ta sử dụng các biến sau:

```
elves = 0
reindeer = 0
santaSem = Semaphore (0
reindeerSem = Semaphore (0)
elfTex = Semaphore (1)
mutex = Semaphore (1)
```

Trong đó, elves và reindeer là hai biến đếm được coi là tài nguyên găng và cần được bảo vệ bởi đèn báo mutex. Ông già Noel sẽ sử dụng hai biến này để kiểm tra số lượng yêu tinh và tuần lộc

Ông già Noel sẽ chờ cho đến khi được báo bởi đèn báo santaSem khi yêu tinh hoặc tuần lộc gửi tín hiệu đến.

Ba chú yêu tinh sẽ sử dụng đèn báo elfTex để ngăn không cho các chú yêu tinh khác vào trong khi 3 chú đang nhận được sự giúp đỡ từ ông già Noel.
Lời giải:
Code cho ông già Noel:

```
santaSem.wait()
mutex.wait()
       if reindeer = 9:
              prepareSleigh()
              reindeerSem.signal(9)
       else if elves = 3:
              helpElves()
mutex signal()
```

Khi ông già Noel thức dậy, ông sẽ kiểm tra biến reindeer đã bằng 2 chưa, nếu đúng, ông sẽ đi chuẩn bị xe kéo qua câu lệnh prepareSleigh() và đồng thời gửi 9 tín hiệu đến 9 chú tuần lộc thông qua đèn báo reindeerSem.signal(9). Những chú tuần lộc được báo sẽ được nối vào xe qua phương thức getHitched(). Tương tự như vậy khi có đủ số yêu tinh đợi ông già Noel, ông sẽ giúp chúng qua câu lệnh helpElves(). Yêu tinh không cần đợi ông già Noel vì chỉ cần đủ số lượng và gửi tín hiệu đến santaSem() thì sẽ được giúp ngay lập tức.

Code cho những chú tuần lộc:

```
mutex.wait()
          reindeer +=1
          if reindeer = 9:
              santa Sem.signal()
mutex.signal()
reindeer Sem.wait()
getHitched()
```

Những chú tuần lộc khi nào có đủ 9 sẽ gửi tín hiệu đến đèn báo santaSem và chờ ông già Noel gửi tín hiệu lại qua đèn báo reindeerSem. Sau đó sẽ tiến hành phương thức getHitched().
Code cho những chú yêu tinh:

```
elfTex.wait()
mutex.wait()
            elves += 1
            if elves == 3:
                  santaSem.signal()
            else
                  elfTex.signal()
mutex signal()
getHelp()
mutex.wait()
            elves -= 1
```

```
            if elves = 0:
                  elfTex.signal()
mutex.signal()
```

Hai chú yêu tinh đầu tiên nhận và giải phóng ngay đèn báo elfTex. Song chú yêu tinh thứ 3 sẽ giữ đèn báo này cho đến khi được giúp. Sau đó chú yêu tinh cuối cùng rời đi sẽ gửi elfTex.signal() để báo hiệu, cho phép nhóm yêu tinh tiếp theo vào.

2.6 Bài toán xe bus Senate

Bài toán này được dựa trên xe bus Senate của trường cao đẳng Wellesley. Người đi xe chờ xe bus ở điểm dừng để chờ xe. Khi xe bus đến, tất cả các hành khách yêu cầu boardBus(lên xe), nhưng với những ai đi đến điểm dừng trong khi xe bus đang trong trạng thái boarding (đang đón khách) thì phải đợi đến lượt lên tiếp theo. Dung lượng của mỗi xe là 50 hành khách, khi có 50 hành khách đang chờ, một số người cần phải chờ đến xe tiếp theo.
Nếu mọi hành khách đang chờ đã lên xe, xe bus có thể yêu cầu depart(khởi hành) nếu xe bus đến trạm dừng xe mà không có hành khách nào, nó sẽ depart ngay lập tức.
Câu hỏi: viết đoạn code đồng bộ hóa thỏa mãn mọi ràng buộc ở trên

Gợi ý
Một số biến khởi tạo:

```
riders=0
mutex=semaphore(1)
multiplex= semaphore(50)
bus=semaphore(0)
allAboard=Semaphore(0)
```

mutex biến nội bộ của riders, theo dõi số lượng hành khách đang chờ, multiplex đảm bảo không có quá 50 hành khách lên xe hành khách chờ bus, được tín hiệu khi bus đến nơi. Bus đợi allAboard, được tín hiệu bởi hành khách cuối cùng lên xe.

Giải pháp 1
Code của xe bus: Sử dụng mẫu "Pass the baton" (truyền cây baton)

```
mutex .wait()
if riders > 0:
     bus.signal() # truyn mutex
     allAboard.wait() # tr   li   mutex
mutex.signal()
depart()
```

Khi bus đến, nó lấy mutex, ngăn chặn người mới đến vào vùng xuất phát. Nếu không có hành khách đợi, nó khởi hành ngay lập tức. Nếu không, nó ra hiệu bus và đợi hành khách để khởi hành.

Code của hành khách:

```
multiplex.wait()
     mutex.wait()
          riders += 1
     mutex.signal()
     bus.wait()     # and  get the mutex
multiplex.signal()
boardBus()
riders -= 1
if riders == 0
     allAboard.signal ()
else :
     bus.signal()   # and pass the mutex
```

Biến đồng bộ kiểm soát số lượng hành khách trong vùng chờ, thực tế hành khách không được vào vùng này cho đến khi riders được tăng
Riders phải đợi bus cho đến khi bus đến, khi hành khách được wake up, tức là họ được cấp mutex. Sau khi lên, mỗi hành khách làm giảm riders. Nếu có nhiều hành khách đang chờ, hành khác trước sẽ ra hiệu cho bus và truyền mutex đến hành khách tiếp theo, hành khách cuối cùng sẽ ra hiệu cho allAboard và truyền lại mutex về bus. Cuối cùng bus sẽ giải phóng mutex và khởi hành.

Giải pháp 2
Grant Hutchin đưa ra giải pháp sử dụng ít biến hơn giải pháp cũ, không cần truyền mutex.
Đây là các biến:

```
waiting = 0
mutex = new Semaphore (1)
bus = new Semaphore (0)
Boarded = new Semaphore (0)
```

Trong đó waiting là số lượng hành khách trên vùng chuẩn bị lên xe, là biến nội bộ của mutex. Bus ra hiệu khi khi đến nơi, tín hiệu board khi 1 hành khách đã yêu cầu sẵn sang đi.

Đây là code của xe bus:

```
mutex.wait()
n = min (waiting , 50)
for i in range (n):
     bus.signal()
```

```
     boarded.wait()
waiting = max (waiting -50 , 0)
mutex.signal()
depart()
```

Xe bus lấy mutex và giữ nó trong suốt quá trình lên xe. Vòng lặp ra hiệu mỗi hành khách đến lượt và đợi để lên xe. Bằng cách kiểm soát số lượng tín hiệu, bus tránh được việc có quá 50 hành khách lên xe.
Khi tất cả các hành khách đã được lên xe, bus cập nhật waiting, (là 1 ví dụ của"I'll do it for you")

Code của hành khách sử dụng 2 mẫu mutex và điểm hẹn:

```
mutex.wait()
     waiting += 1
mutex.signal()
bus.wait()
board()
boarded.signal()
```

Thách thức: nếu hành khách đến khi bus đang cho hành khách lên xe, họ có thể thấy khó chịu nếu bạn làm họ đợi vào đợt tiếp theo. Bạn có thể tìm giải pháp cho những người đến muộn mà không vi phạm các ràng buộc?

3 Tài liệu tham khảo

1. Bài giảng Nguyên lý Hệ điều hành - Phạm Đăng Hải
2. The little book of semaphores - Allen B.Downey
3. https://en.wikipedia.org/wiki/Semaphore_(programming)
4. https://en.wikipedia.org/wiki/Cigarette_smokers problem
5. https://www.it2051229.com/sushibarproblem.html
6. Các tài liệu khác trên Internet