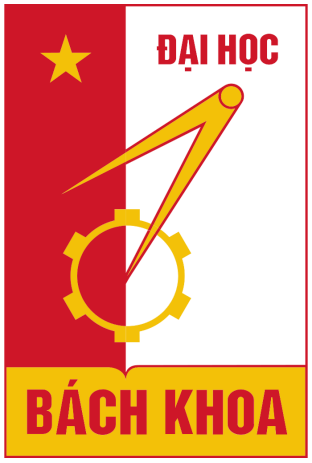
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

———————— * ————————



BÀI TẬP MÔN HỌC
NGUYÊN LÝ HỆ ĐIỀU HÀNH

CÁC BÀI TOÁN ĐỒNG BỘ TIẾN TRÌNH CỔ ĐIỂN

Đào Quốc Tuấn 20210891
Nguyễn Hoàng Lâm 20210517
Nguyễn Trọng Tuấn 20210906

Giáo viên hướng dẫn: **TS.Phạm Đăng Hải**

HÀ NỘI
Ngày 25 tháng 6 năm 2023

Mục lục

Chương 1

Lời nói đầu

`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`, `Reader-Writer` 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, hoặc ngăn ngừa starvation xảy ra để không một tiến trình nào bị bỏ đói.

Đề tài: Tìm hiểu các bài toán đồng bộ tiến trình cổ điển
Môn học: Nguyên lý hệ điều hành
Giáo viên hướng dẫn: Phạm Đăng Hải
Phát hành: 28/05/2023

Chương 2

Các bài toán về đèn báo

2.1 Reader-Writer Problem

2.1.1 Đặt vấn đề

Bài toán Reader-Writer là một bài toán đồng bộ hóa trong lập trình đa luồng hoặc đa tiến trình, liên quan đến việc quản lý truy cập vào một tài nguyên chia sẻ giữa các tiến trình hoặc luồng. Trong bài toán này, có hai loại tiến trình hoặc luồng: đọc (Reader) và viết (Writer), cùng truy cập đến 1 cơ sở dữ liệu (critical section).

Các yêu cầu của bài toán:

- Nhiều tiến trình Readers cùng truy cập 1 CSDL và 1 số tiến trình Writer cập nhật CSDL.
- Cho phép số lượng tùy ý các tiến trình Readers cùng truy cập CSDL (nghĩa là đang tồn tại một tiến trình Reader truy cập CSDL, mọi tiến trình Readers khác đều được truy cập CSDL, còn các tiến trình writers phải xếp hàng chờ đợi).
- Khi một tiến trình Writer đang sử dụng CSDL thì không một tiến trình nào được phép truy cập CSDL đó nữa.

2.1.2 Giải pháp sử dụng Semaphore

Các biến sử dụng cho bài toán:

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

Trong đó,

- `readers` đại diện cho số tiến trình readers có trong CSDL
- `mutex` dùng để đảm bảo chỉ có 1 tiến trình có thể thay đổi giá trị biến đếm `readers` trong 1 thời điểm

- `roomEmpty` có giá trị là 1 nếu không có tiến trình readers, writers nào đang sử dụng CSDL, bằng 0 nếu ngược lại.

Đoạn chương trình cho Writer

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

Đoạn chương trình cho Reader

```
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 tiến trình Writers. Khi một Writer truy cập vào CSDL, biến roomEmpty sẽ được khóa lại và các luồng khác sẽ không thể truy cập vào.

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

Readers đầu tiên đến phải chờ đợi biến roomRoomEmpty mở. Nếu phòng đang trống, Readers được cấp quyền truy cập, trong cùng thời điểm đó, ngăn chặn Writer. Những tiến trình Readers tiếp theo vẫn có thể tham gia bởi họ không phải chờ roomEmpty.

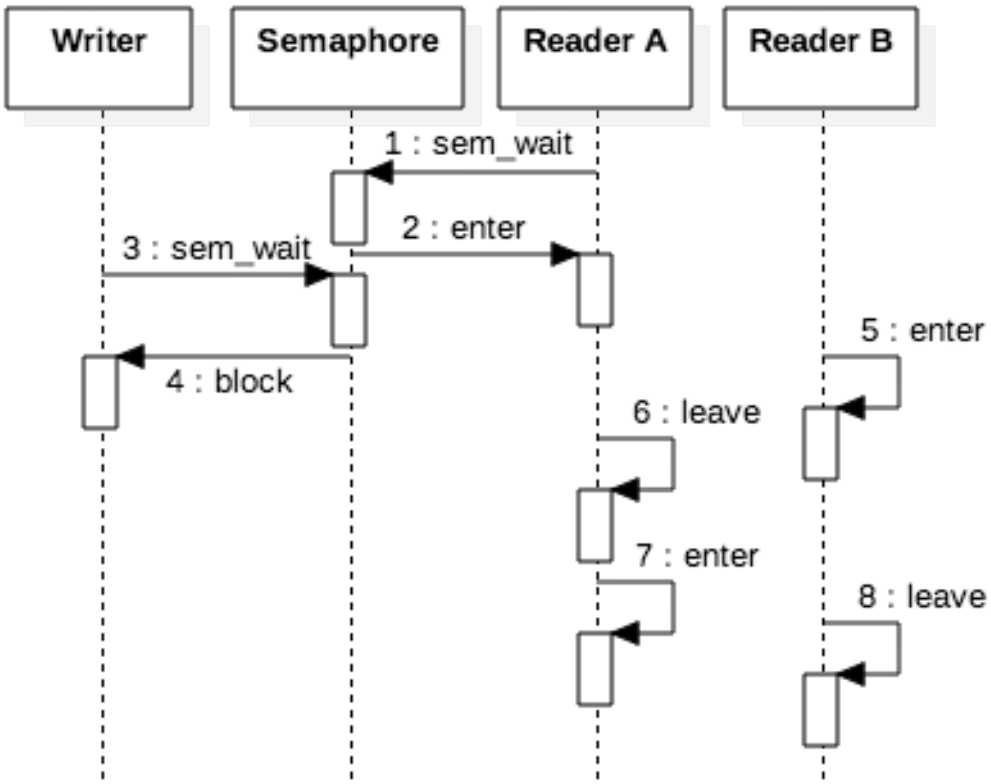
Nếu 1 tiến trình Reader đến trong khi đã có tiến trình Writer ở trong CSDL, nó sẽ phải đợi ở biến roomEmpty. Vì nó chiếm giữ mutex nên những tiến trình Readerstiếp theo phải xếp hàng ở mutex.

Starvation

Trong cách giải quyết bài toán trên, sự bế tắc không xảy ra. Tuy nhiên có thể 1 tiến trình Writer sẽ bị bỏ đói (Stavation)

Nếu có 1 tiến trình Writer đến CSDL nó phải chờ đến khi CSDL trống, trong khi các tiến trình Readers chỉ đến là có quyền truy cập vào CSDL. Miễn là tiến trình Readers mới đến trước khi tiến trình Readerscuối cùng trong số tiến trình Readers hiện tại rời đi, thì sẽ luôn có ít nhất một tiến trình Readers trong phòng.

Ví dụ được minh họa ở hình dưới:



Cách giải quyết: Khi có 1 tiến trình Writer xin truy cập vào CSDL, các tiến trình Readers đang ở trong CSDL tiếp tục thực hiện, tuy nhiên không có tiến trình Readers khác được thêm vào.

2.1.3 Giải pháp bài toán No-starve readers-writers

Giải pháp no-starvation cho Writer

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

roomEmpty.signal()
```

Giải pháp no-starvation cho Reader

```
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 tiến trình Writer đến trong khi có tiến trình Reader trong phòng, tiến trình Writer sẽ phải chờ ở câu lệnh "roomEmpty.wait()", đồng nghĩa với đèn báo turnstile bị đóng. Nó ngăn chặn tiến trình Reader khác truy cập khi tiến trình Writer đang trong hàng chờ.

Khi tiến trình Reader cuối cùng rời khỏi phòng, nó mở khóa roomEmpty, mở khóa tiến trình Writer đang chờ. Tiến trình Writer ngay lập tức có quyền truy cập vào CS, vì không có tiến trình Reader 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 tiến trình Readershoặc nhà văn. Vì vậy, giải pháp này đảm bảo rằng ít nhất một tiến trình Writer được tiếp tục, nhưng vẫn có thể cho một tiến trình Reader vào trong khi có những tiến trình Writer đang trong hàng chờ.

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

2.2 Dining Philosophers Problem

2.2.1 Đặt vấn đề

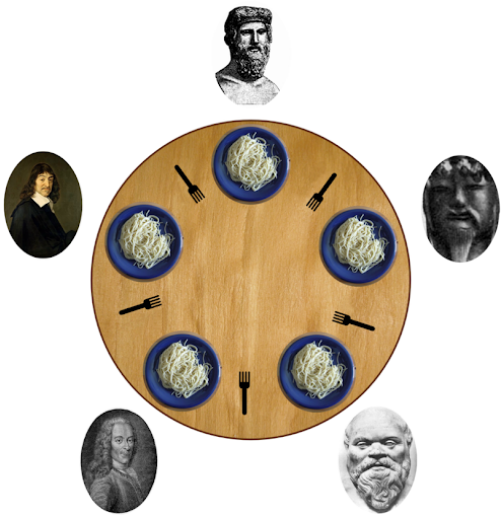
Năm 1965, Dijkstra đưa ra và giải quyết một vấn đề đồng bộ hóa mà ông gọi là vấn đề nhà triết gia ăn tối. Kể từ đó, tất cả những người phát minh ra một phương tiện đồng bộ hóa mới đều cảm thấy nghĩa vụ phải chứng minh rằng phương tiện đồng bộ hóa mới tuyệt vời đến đâu bằng cách cho thấy nó giải quyết vấn đề nhà triết gia ăn tối.

- Năm nhà triết gia ngồi quanh một bàn tròn. Mỗi nhà triết gia có một đĩa mì spaghetti.
- Mỗi nhà triết gia cần hai cái nĩa để ăn. Giữa mỗi cặp nĩa có một chiếc đĩa.

Vòng lặp cơ bản của một triết gia

```
while (TRUE) :
    think()
    getfork()
    eat()
    putfork()
```

Ta biểu diễn qua hình vẽ:



Bài toán được phát biểu như sau: Cuộc sống của một nhà triết gia bao gồm các giai đoạn thay phiên giữa ăn và suy nghĩ. (Điều này là một sự trừu tượng, ngay cả đối với những nhà triết học, nhưng các hoạt động khác là không liên quan ở đây.) Khi một nhà triết gia đói, triết gia sẽ lấy nĩa trái và phải của mình, một cái sau cái kia, theo bất kỳ thứ tự nào. Nếu thành công trong việc lấy được hai cái nĩa,triết gia ấy ăn một lúc, sau đó đặt xuống nĩa và tiếp tục suy nghĩ.

- Một triết gia có thể lấy chỉ một chiếc nĩa tại một thời điểm.
- Chú ý, ông ta không thể lấy chiếc đũa mà nó đang được dùng bởi người láng giềng.
- Khi một triết gia đói và có hai chiếc nĩa cùng một lúc, ông ta ăn mà không đặt nĩa xuống. Khi triết gia ăn xong, ông ta đặt nĩa xuống và bắt đầu suy nghĩ tiếp.

Câu hỏi chính là: bạn có thể viết một chương trình cho mỗi nhà triết gia sao cho các triết gia hoạt động và không bị kẹt lại?

2.2.2 Phân tích

Mỗi triết gia sẽ tượng trưng cho một tiến trình, và những chiếc nĩa là tài nguyên găng. Các triết gia sẽ có 4 trạng thái lần lượt theo thứ tự là:

1. Suy nghĩ (Thinking)
2. Lấy nĩa (Get Fork)
3. Đang ăn (Eating)
4. Thả đũa (Put Fork)

Mỗi tiến trình sẽ thực hiện việc lấy tài nguyên và trả lại sau khi một thời gian chiếm giữ, tương ứng với hành động lấy và trả lại các chiếc nĩa của triết gia tương ứng.

2.2.3 Giải pháp

Cần viết hai hàm `getFork` và `putFork` 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
- Có thể có nhiều hơn thứ nhất 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.

Thiết lập biến sử dụng cho dining philosophers

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

Dining philosophers non-solution

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

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

Deadlock

Deadlock xảy ra: 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.

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).

2.2.4 Giải pháp cho Deadlock

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 đói để ăn và một đèn báo để chỉ nếu nhà triết gia có thể bắt đầu ăn.

Có 2 cách để một nhà triết gia có thể bắt đầu ăn:

- Cách đầu tiên là người đấy thực hiện `getFork()`, tìm 2 nĩa và tiến hành ăn.
- Cách thứ hai là, khi láng giềng đang ăn và nhà triết gia bị block bởi đèn báo. Sau khi người láng giềng ăn xong, triết gia có thể ăn. Để có thể 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.

Cài đặt các biến sử dụng cho Tanenbaum's solution

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

Tanenbaum's solution

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

def putfork(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()
```

2.3 The Sleeping Barber Problem

Bài toán Người thợ cắt tóc ngủ gật (The Sleeping Barber) là một trong những bài toán kinh điển về đồng bộ tiến trình. Bài toán gốc được đề xuất bởi Dijiktra vào năm 1965. Bài toán này lấy bối cảnh một tiệm cắt tóc, nơi một thợ cắt tóc tuần tự phục vụ nhiều khách hàng, được miêu tả cụ thể dưới đây.

2.3.1 Đặt vấn đề

Hãy tưởng tượng một tiệm cắt tóc giả định có một thợ cắt tóc, một ghế cắt tóc và một hàng chờ có n ghế (n có thể bằng 0) dành cho những khách hàng đang chờ. Bài toán có yêu cầu như sau:

- Một khách hàng bước vào tiệm tại một thời điểm bất kỳ để cắt tóc và nếu tất cả các ghế đều không còn trống, thì khách hàng đó sẽ rời khỏi tiệm.
- Nếu thợ cắt tóc bận (cắt tóc cho khách hàng khác) nhưng vẫn còn ghế, thì khách hàng sẽ ngồi vào một trong những chiếc ghế trống.
- Khi thợ cắt tóc kết thúc quá trình cắt tóc với một khách hàng, anh ta kiểm tra xem có khách hàng nào đang ngồi đợi không
  - Nếu có, anh ta bắt đầu cắt tóc cho khách hàng tiếp theo trong hàng ghế.
  - Nếu không có khách đợi, anh ta sẽ ngủ gật trên ghế.
- Nếu thợ cắt tóc đang ngủ, vị khách đến tiệm sẽ đánh thức thợ cắt tóc.

Hình minh họa

2.3.2 Phân tích

Trong thực tế, có một số vấn đề có thể xảy ra minh họa cho các vấn đề về lập lịch biểu chung. Các hành động của thợ cắt tóc và khách hàng (như kiểm tra phòng chờ, vào tiệm, lấy ghế trong phòng chờ,....) đều mất một khoảng thời gian không xác định.

Ví dụ: Khi một khách hàng có thể đến và quan sát thấy thợ cắt tóc đang cắt tóc, vì vậy anh ta đi vào hàng ghế chờ. Trong khi người này đang trên đường đến hàng chờ, người thợ cắt tóc hoàn thành công việc cắt tóc hiện tại của ông ta và đi kiểm tra hàng chờ. Vì không có ai ở đó (có thể hàng chờ ở xa và người thợ cắt tóc đi nhanh hơn khi lướt qua khách hàng, ...), vì vậy người thợ cắt tóc đi trở lại ghế của ông ta và ngủ. Người thợ cắt tóc bây giờ đang đợi khách hàng, nhưng khách hàng đang đợi thợ cắt tóc. Hoàn cảnh trên sẽ dẫn đến hiện tượng **deadlock** - xảy ra khi cả khách hàng và thợ cắt tóc đều cùng chờ đợi lẫn nhau.

Trong một tình huống khác, **starvation** có thể xảy ra khi một khách hàng phải chờ đợi trong một thời gian dài, khi thợ cắt tóc gọi khách một cách ngẫu nhiên.

Để làm cho vấn đề cụ thể hơn một chút, nhiệm vụ của chúng ta chính là viết ra các hàm thỏa mãn được những yêu cầu sau:

- Luồng Customers có thêm hàm `getHairCut()`.
- Nếu một luồng Customer đến khi hàng ghế chờ đã đầy, luồng này gọi tới hàm `balk()` và luồng này sẽ rời đi.
- Luồng Barber có thêm hàm hàm `cutHair()` thỏa mãn khi nó gọi tới hàm `cutHair()`, phải có chính xác một luồng Customer gọi tới `getHairCut()` đồng thời.

Để giải quyết bài toán cũng như vấn đề deadlock, ta sử dụng các biến semaphore.

Ta khởi tạo các biến sẽ sử dụng

```
1  n = 4
2  customers = 0
3  mutex = Semaphore (1)
4  customer = Semaphore (0)
5  barber = Semaphore (0)
6  customerDone = Semaphore (0)
7  barberDone = Semaphore (0)
```

n là số lượng khách hàng ở thể ở trong tiệm: n-1 ở hàng ghế chờ và một ở ghế cắt tóc. `customers` đếm số lượng khác hàng trong tiệm, được bảo hộ bởi đèn báo `mutex`.

Thợ cắt tóc chờ ở biến `customer` cho đến khi có một khách bước vào tiệm, và khách hàng chờ đợi ở `barber` cho đến khi thợ cắt tóc cho phép anh ta ngồi vào ghế.

Sau khi cắt tóc, khách hàng gọi tới `customerDone.signal()` và `barberDone.wait()`.

Có tối đa n khách hàng có thể đi vào tiệm và gọi tới `customers.signal()` và `barber.wait()`. Khi thợ cắt tóc gọi tới `barber.signal()`, bất kỳ khách hàng nào cũng có thể được tiến hành cắt tóc. Để giải quyết vấn đề `stavartion` sao cho khách hàng được phục vụ theo thứ tự họ đi vào tiệm, chúng ta thêm các semaphore `queue`. Khi mỗi khách hàng đi vào tiệm, nó sẽ

tạo một luồng và đặt nó vào hàng đợi `queue`. Thay vì đợi tín hiệu từ thợ cắt tóc, mỗi luồng đợi trên semaphore của chính nó.

2.3.3 Giải pháp

Giải pháp FIFO barbershop (customer)

```
self.sem = Semaphore(0)
mutex.wait()
    if customers == n :
        mutex.signal()
        balk()
    customers += 1
    queue.append(self.sem)
mutex.signal()

customer.signal()
self.sem.wait()

getHairCut()

customerDone.signal()
barberDone.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

Giải pháp FIFO barbershop (barber)

```
customer wait()
mutex.wait()
    sem = queue.pop(0)
mutex.signal()

sem.signal()

cutHair()

customerDone.wait()
barberDone.signal()
```

Mỗi khi có khách ra tín hiệu signal, người thợ cắt tóc thức dậy, ra hiệu cho thợ cắt tóc và cắt tóc cho một khách hàng. Nếu một khách hàng khác đến trong khi thợ cắt tóc đang bận, thì trong lần signal tiếp theo, thợ cắt tóc sẽ không ngủ.

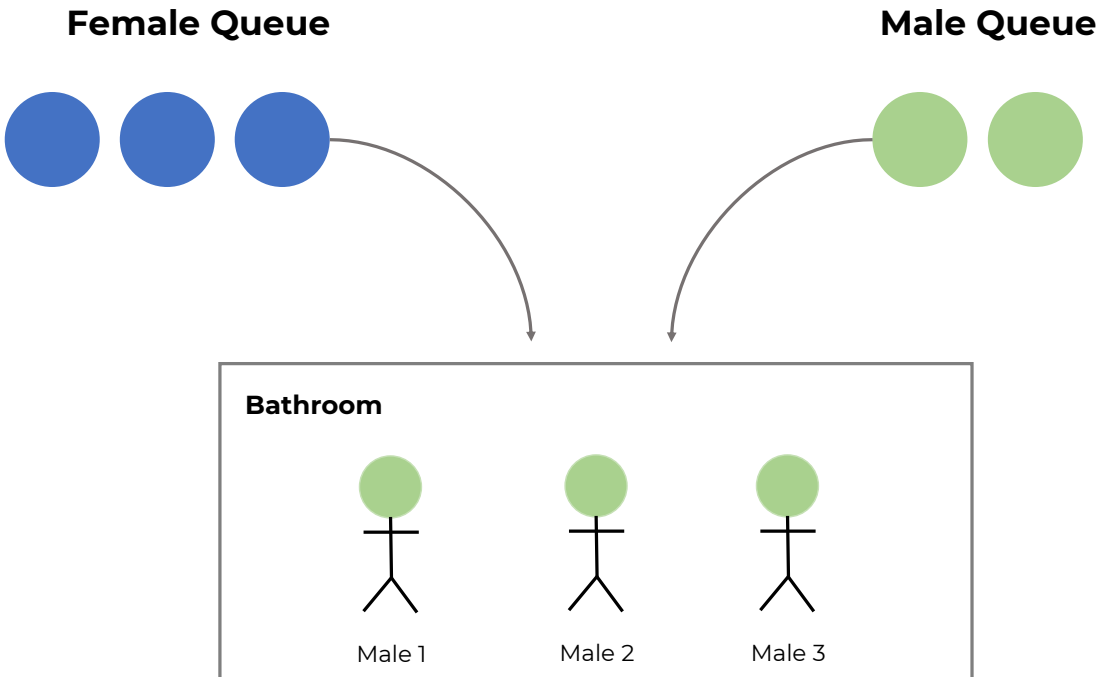
2.4 The Unisex Bathroom Problem

2.4.1 Đặt vấn đề

Nội dung bài toán Bathroom:

- Một phòng tắm phục vụ cho cả nam và nữ nhưng chỉ phục vụ cho nam (hoặc nữ) trong một thời điểm nhất định.
- Nếu phòng tắm chưa có người thì ai cũng có thể vào.
- Nếu phòng tắm đang được sử dụng thì chỉ có người cùng giới tính với người trong phòng tắm mới có thể vào.
- Số lượng người sử dụng phòng tắm trong một đơn vị thời gian là có giới hạn (trong trường hợp này giới hạn được nhập vào từ bàn phím).

Minh họa bài toán bằng hình vẽ:



2.4.2 Phân tích

- Có 2 kiểu tiến trình `male`() và `female`().
- Mỗi tiến trình ở trong Bathroom một khoảng thời gian ngẫu nhiên.
- Sử dụng kỹ thuật đèn báo để điều độ các tiến trình `male`() và `female`() qua đoạn găng (coi các thread như là các tiến trình độc lập).
- Tại một thời điểm chỉ có một loại tiến trình được sử dụng phòng tắm, các tiến trình muốn sử dụng, ở đây phòng tắm là tài nguyên găng được các tiến trình `male` và `female` trưng dụng.

2.4.3 Giải pháp

- `empty` là 1 nếu căn phòng trống và 0 nếu ngược lại.
- `maleswitch` cho phép male chặn female vào phòng tắm. Khi người đàn ông đầu tiên bước vào, `Lightswitch` khóa trống, chặn phụ nữ; Khi người đàn ông cuối cùng thoát ra, nó mở khóa trống, cho phép phụ nữ bước vào. Tương tự với females bằng cách sử dụng `Femaleswitch`.
- `maleMultiplex` and `femaleMultiplex` đảm bảo rằng không có quá ba male và ba woman ở hệ thống cùng một thời điểm.

Khởi tạo các biến

```
empty = Semaphore(1)
maleSwitch = Lightswitch()
femaleSwitch = Lightswitch()
maleMultiplex = Semaphore(3)
femaleMultiplex = Semaphore(3)
```

Unisex bathroom solution (female)

```
femaleSwitch.lock(empty)
femaleMultiplex.wait()
# bathroom code here
femaleMultiplex.signal()
female Switch.unlock(empty)
```

Unisex bathroom solution (male)

```
maleSwitch.lock(empty)
maleMultiplex.wait()
# bathroom code here
maleMultiplex.signal()
male Switch.unlock(empty)
```

Starvation

Có một vấn đề với lời giải trên là nó sẽ dẫn tới Starvation; Cụ thể, một hàng dài women đi đến và tiến vào trong khi đó man vẫn phải đợi và ngược lại.

2.4.4 No-starve unisex bathroom solution

No-starve unisex bathroom solution (male)

```
turnstile.wait ()
maleSwitch.lock(empty)
turnstile.signal()
```

```
    maleMultiplex.wait()

    # bathroom code here
    maleMultiplex.signal()
    maleSwitch.unlock(empty)
```

Ở đây ta sẽ dùng **turnstile** để cho phép một luồng dừng dòng thực thi của một luồng khác. Trên đây là thuật toán cho male.

Khi có male ở trong phòng, những người tới sau đó sẽ vào phòng qua **turnstile.** Nếu trong phòng có female, khi male đến **turnstile** sẽ chặn male không cho vào phòng. Điều này sẽ chặn tất cả những người đến sau muốn vào phòng tắm (male và female) cho đến khi những người trong phòng rời đi. Khi đó male bị chặn ở **turnstile** được bước vào và sẽ cho phép các tiền trình male tiếp theo vào phòng.

Tương tự với female, nếu có male ở trong phòng female sẽ bị chặn ở **turnstile** cho tới khi các male trong phòng rời đi.

Giải pháp này có thể không hiệu quả. Nếu hệ thống quá bận, thường sẽ có một số luồng male và female xếp hàng ở **turnstile**. Mỗi lần **empty** báo hiệu thì 1 luồng sẽ rời khỏi **turnstile** và các luồng khác sẽ vào phòng. Nếu giới tính người vào không giống người trong phòng thì sẽ bị **turnstile** chặn lại và các luồng đến sau sẽ phải xếp hàng đợi. Do đó thường chỉ có 1-2 người trong phòng tắm một lúc và hệ thống sẽ không tận dụng tối đa tài nguyên.

2.5 Building H_2O Problem

2.5.1 Đặt vấn đề

Bài toán này được giảng dạy tại lớp Hệ điều hành tại U.C. Berkeley trong hơn một thập kỷ qua - và nó dựa trên một bài tập trong Andrews's Concurrent Programming.

Trong bài toán này, có hai luồng (threads) - oxy và hydro. Để lắp ráp các luồng này thành các phân tử nước, chúng ta phải tạo ra một rào chắn (barrie) khiến mỗi luồng chờ đợi cho đến khi một phân tử nước (H2O) hoàn chỉnh sẵn sàng hình thành.

Khi mỗi luồng vượt qua barrie, nó sẽ gọi **bond** (kích hoạt liên kết). Phải đảm bảo rằng tất cả các luồng từ một phân tử gọi **bond** (tạo liên kết) trước khi một luồng của phân tử nước khác gọi tới thủ tục tạo liên kết.

Chi tiết hơn:

- Nếu một luồng oxy đến barrie khi không có luồng hydro, nó phải chờ hai luồng hydro.
- Nếu một luồng hydro đến barrie khi không có luồng nào khác, nó phải chờ một luồng oxy và một luồng hydro khác.

Điều quan trọng chỉ là các luồng vượt qua barrie tạo thành một phân tử H2O hoàn chỉnh; Do đó, chúng ta kiểm tra trình tự các luồng gọi liên kết và chia chúng thành các nhóm ba, mỗi nhóm nên chứa một oxy và hai hydro.

2.5.2 Phân tích

Ta khởi tạo các biến sẽ sử dụng

```
1 mutex = Semaphore(1)
2 oxygen = 0
3 hydrogen = 0
4 barrier = Barrier(3)
5 oxyQueue = Semaphore(0)
6 hydroQueue = Semaphore(0)
```

`Oxy` và `hydro` là những biến đếm, được bảo hộ bởi `mutex`. `barrie` là nơi mỗi bộ ba luồng gặp nhau sau khi gọi liên kết (bond) và trước khi cho phép tập hợp các luồng khác để tổng hợp phân tử H_2O khác.

`oxyQueue` là các luồng oxy đang chờ đợi; `hydroQueue` là các luồng hyđrô đang chờ đợi. `oxyQueue.wait()` có nghĩa là "thêm vào hàng đợi oxy" và `oxyQueue.signal()` có nghĩa là "giải phóng một luồng oxy khỏi hàng đợi".

2.5.3 Giải pháp

Ban đầu, `hydroQueue` và `oxyQueue` bị khóa, và khi một luồng oxy đến, nó sẽ báo hiệu `hydroQueue` hai lần, cho phép hai hydro tiếp tục.

Sau đó, các luồng oxy chờ các luồng hydro đến.

Oxygen code

```
mutex.wait()
oxygen += 1
if hydrogen >= 2:
   hydroQueue.signal(2)
   hydrogen -= 2
   oxyQueue.signal()
   oxygen -= 1
else:
   mutex.signal()

oxyQueue.wait()
bond()

barrier.wait()
mutex.signal()
```

Khi mỗi luồng oxy đi vào, nó sẽ nhận được mutex và kiểm tra hàng đợi. Nếu có ít nhất hai luồng hydro đang đợi, nó báo hiệu hai trong số chúng và chính nó và sau đó tạo thành liên kết. Nếu không, nó sẽ giải phóng mutex và chờ đợi.

Sau khi liên kết, các luồng chỉ đợi ở barrier cho đến khi cả ba luồng liên kết với nhau, và sau đó luồng oxy giải phóng mutex. Vì chỉ có một luồng oxy trong mỗi bộ ba, do đó sẽ phát tín hiệu mutex một lần.

Giải pháp cho Hydrogen khá tương tự:

Hydrogen code

```
mutex.wait()
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
   hydroQueue.signal (2)
   hydrogen -= 2
   oxyQueue.signal ()
   oxygen -= 1
else :
   mutex.signal()

hydroQueue.wait()
bond()

barrier.wait()
```

Một điểm bất bất thường của giải pháp này là điểm thoát của mutex không rõ ràng. Trong một số trường hợp, các luồng đi vào mutex, cập nhật bộ đếm và thoát khỏi mutex. Nhưng khi một luồng đến tạo thành một bộ ba, nó phải giữ mutex để chặn các luồng tiếp theo cho đến khi tập hợp hiện tại đã gọi **bond** (tạo liên kết).

Sau khi tạo liên kết, ba luồng chờ đợi ở một barrie. Khi barrie mở ra, chúng ta biết rằng cả ba luồng đã gọi liên kết và một trong số chúng nắm giữ mutex. Chúng ta không biết luồng nào nắm giữ mutex, nhưng điều đó không quan trọng miễn là chỉ một trong số chúng giải phóng nó. Vì chỉ có một luồng oxy,

Điều này có vẻ sai, bởi vì cho đến nay thường đúng là một luồng phải giữ một khóa để giải phóng nó. Nhưng không có quy tắc nào nói rằng điều đó phải đúng. Đây là một trong những trường hợp có thể gây hiểu lầm khi coi mutex như một mã thông báo mà các luồng phải thu được và giải phóng.

2.6 Cigarette Smokers Problem

2.6.1 Đặt vấn đề

Vấn đề của người hút thuốc ban đầu được đề xuất bởi Suhas Patil, ông cho rằng vấn đề này không thể được giải quyết bằng các semaphores. Tuyên bố này đi kèm với một số điều kiện, nhưng trong mọi trường hợp, vấn đề này là thú vị và đầy thách thức.

- Có bốn luồng tham gia: một đại lý và ba người hút thuốc. Các người hút thuốc lặp vô tận, đầu tiên đợi nguyên liệu, sau đó làm và hút thuốc
- Các nguyên liệu bao gồm thuốc lá, giấy và diêm (tobacco, paper, and matches)

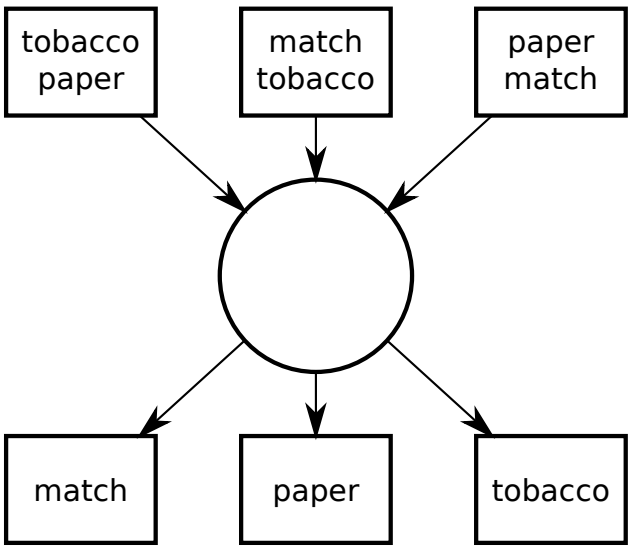
Bài toán được phát biểu như sau: Chúng ta giả định rằng đại lý có một nguồn cung vô hạn của cả ba nguyên liệu, và mỗi người hút thuốc có một nguồn cung vô hạn của một trong ba nguyên liệu; tức là, một người hút có diêm, người khác có giấy và người thứ ba có thuốc lá.

Đại lý lặp đi lặp lại chọn ngẫu nhiên hai nguyên liệu khác nhau và đưa chúng sẵn có cho các người hút thuốc. Tùy thuộc vào nguyên liệu nào được chọn, người hút thuốc có nguyên liệu bổ sung phải lấy cả hai nguyên liệu và tiếp tục.

Ví dụ, nếu đại lý đưa ra thuốc lá và giấy, người hút thuốc có diêm sẽ lấy cả hai nguyên liệu, làm một điếu thuốc và sau đó thông báo cho đại lý.

Để giải thích, đại lý đại diện cho một hệ điều hành phân bổ tài nguyên, và các người hút thuốc đại diện cho các ứng dụng cần tài nguyên. Vấn đề là đảm bảo rằng nếu có tài nguyên có sẵn mà cho phép một ứng dụng khác tiếp tục, các ứng dụng đó sẽ được đánh thức. Ngược lại, chúng ta muốn tránh đánh thức một ứng dụng nếu nó không thể tiếp tục được.

Minh họa bài toán bằng hình vẽ:



Dựa trên tiền đề này, có ba phiên bản của vấn đề này thường xuất hiện trong sách:

- ***Phiên bản bất khả thi (The impossible version):*** Phiên bản của Patil đặt ra hạn chế cho giải pháp. Trước tiên, bạn không được phép sửa đổi mã đại lý. Nếu đại lý đại diện cho một hệ điều hành, có lý để giả định rằng bạn không muốn sửa đổi nó mỗi khi có ứng dụng mới xuất hiện. Hạn chế thứ hai là bạn không thể sử dụng câu lệnh điều kiện hoặc một mảng semaphore. Với những ràng buộc như vậy, vấn đề không thể được giải quyết, nhưng như Parnas chỉ ra, hạn chế thứ hai khá nhân tạo. Với những ràng buộc như vậy, nhiều vấn đề trở nên không thể giải quyết.

- ***Phiên bản thú vị (The intersting version):*** Phiên bản này giữ nguyên ràng buộc đầu tiên - bạn không thể thay đổi mã đại diện - nhưng loại bỏ các ràng buộc khác.

- ***Phiên bản đơn giản (The trivial version):*** Trong một số sách, vấn đề chỉ định rằng đại lý nên thông báo cho người hút thuốc tiếp theo, tùy theo nguyên liệu có sẵn. Phiên bản này của vấn đề không thú vị vì nó làm cho tiền đề tổng thể, các nguyên liệu và điếu thuốc trở nên không liên quan. Ngoài ra, về mặt thực tế, có lẽ không phải là một ý kiến tốt để yêu cầu đại lý biết về các luồng khác và điều họ đang chờ đợi. Cuối cùng, phiên bản này của vấn đề quá dễ dàng.

2.6.2 Phân tích

Một cách tự nhiên, chúng ta tập trung vào vấn đề thú vị (The intersting version). Để hoàn thành phát biểu của vấn đề, chúng ta phải chỉ định code của đại lý (the agent code). Người đại lý dùng đoạn code dưới đây:

```
1  //Agent semaphores
2  agentSem = Semaphore(1)
3  tobacco = Semaphore(0)
4  paper = Semaphore(0)
5  match = Semaphore(0)
6
```

Thực tế, đại diện được tạo thành từ ba luồng đồng thời: Đại lý A, Đại lý B và Đại lý C. Mỗi luồng đợi trên agentSem; mỗi khi agentSem được thông báo, một trong các Đại lý được đánh thức và cung cấp nguyên liệu bằng cách thông báo cho hai semaphore.

```
1  //Agent A code
2  agentSem.wait()
3  tobacco.signal()
4  paper.signal()
5
6  //Agent B code
7  agentSem.wait()
8  paper.signal()
9  match.signal()
10
11 //Agent C code
12 agentSem.wait()
13 tobacco.signal()
14 match.signal()
```

Vấn đề này khó bởi vì lời giải thông thường không hoạt động. Rất dễ để viết:

```
//Smoker with matches
tobacco.wait()
paper.wait()
agentSem.signal()

//Smoker with tobacco
paper.wait()
match.wait()
agentSem.signal()

//Smoker with paper
tobacco.wait()
match.wait()
agentSem.signal()
```

Vấn đề này sẽ bế tắc khi hai smoker cùng lúc nắm giữ 1 loại tài nguyên do Agent cung cấp. Giả sử, Agent C được chạy, đèn báo của tobacco và match tăng lên 1, sau đó Smoker with matches lấy được tobacoo, đèn báo tobacco giảm còn 0, Smoker with paper bị block, Smoker with tobacco cũng bị block. Do đó đèn báo agentSem vẫn là 0. Tất cả hoạt động đã dừng lại.

Chương 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 2022-2 này, nhóm chỉ mới có thể tìm hiểu được 6 bài toán tiêu biểu nhất như trên.

Để hoàn thành được báo cáo này, chúng em xin chân thành bày tỏ lời cảm ơn tới thầy ***Phạm Đăng Hải***, người đã cung cấp cho chúng em những kiến thức vô cùng bổ ích, những sự hướng dẫn tận tình và những tiết dạy tuyệt vời và đáng quý ở học phần Nguyên lý hệ điều hành.

Mặc dù chúng em đã đưa ra nỗ lực tìm hiểu và nghiên cứu, bài báo cáo về tìm hiểu về đồng bộ tiến trình của chúng em vẫn có thể không trách khỏi những số thiếu sót cần được cải thiện và bổ sung. Kính mong nhận sự góp ý của thầy và các bạn.

Tài liệu tham khảo

Allen Downey. The little book of semaphores, volume 2. Green Tea Press, 2008

Abraham Silberschatz, Peter B. Galvin, Greg Gagne, A Silberschatz Operating System Concepts, 6th Edition