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

BÁO CÁO BÀI TẬP LỚN

Nghiên cứu các bài toán điều độ tiến trình

Cao Huy Thịnh
MSSV: 20230069
CTTN-KHMT-K68

Nguyễn Đình Tuấn Minh
MSSV: 20230048
CTTN-KHMT-K68

Môn học: Nguyên lý hệ điều hành - IT3070
Mã lớp: 157021, CTTN-KHMT-K68
GVHD: Thầy Phạm Đăng Hải

Hà Nội, ngày 12 tháng 05 năm 2025

Mục lục

1 Giới thiệu

Trong bối cảnh khoa học máy tính và công nghệ thông tin ngày càng phát triển mạnh mẽ, vấn đề quản lý truy cập đồng thời đến các tài nguyên dùng chung giữa các tiến trình đang đóng vai trò then chốt trong việc bảo đảm hiệu quả và độ tin cậy của hệ thống. Đây là một trong những thách thức trọng yếu trong lĩnh vực hệ điều hành, nơi mà các tiến trình phải phối hợp hoạt động một cách hợp lý để tránh xung đột, điều kiện cạnh tranh (race conditions) hoặc hiện tượng bế tắc (deadlock).

Một trong những cơ chế đồng bộ hóa cổ điển và có ảnh hưởng sâu rộng nhất trong lĩnh vực này là semaphore (hay còn gọi là đèn báo), được giới thiệu bởi nhà khoa học máy tính **Edsger Wybe Dijkstra**. Semaphore cho phép kiểm soát truy cập đến tài nguyên dùng chung thông qua các thao tác nguyên tử, từ đó giúp bảo đảm rằng chỉ một số lượng hữu hạn tiến trình có thể truy cập tài nguyên tại một thời điểm nhất định. Nhờ vào tính chất đơn giản trong cài đặt, hiệu quả trong vận hành và khả năng mở rộng cho nhiều tình huống thực tiễn, semaphore đã trở thành công cụ nền tảng trong việc thiết kế các giải thuật đồng bộ hóa trong lập trình hệ thống.

Việc tìm hiểu chi tiết cách hoạt động của semaphore giúp nâng cao hiểu biết về lập trình hệ thống, đặc biệt là lập trình đa tiến trình và xử lý đồng thời.

1.1 Mục tiêu nghiên cứu

Mục tiêu của đề tài là:

- Tìm hiểu lý thuyết và cấu trúc hoạt động của semaphore.
- Phân tích và mô phỏng hai thao tác cơ bản là `wait()` và `signal()`.
- Làm rõ vai trò của semaphore trong việc đồng bộ hoá tiến trình.
- Xây dựng ví dụ minh họa hoặc cài đặt thử nghiệm để kiểm chứng tính đúng đắn của cơ chế.

2 Khái niệm cơ bản

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)`.

Thao tác `wait(S)`

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

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. `wakeup(P)`: Thực hiện tiếp tiến trình P dừng bởi lệnh `block()`.

3 Các bài toán điều độ

3.1 Bài toán bộ tộc ăn tối (The Dining Savages Problem)

1. Mô tả bài toán

Đây là một bài toán kinh điển tương tự như bài toán "Triết gia ăn tối", nhưng tập trung vào vấn đề quản lý tài nguyên dùng chung (nồi thức ăn) giữa nhiều tiến trình (các thành viên bộ tộc). Ta có ngữ cảnh bài toán như sau:

- Một bộ tộc hoang dã ăn chung từ một nồi lớn chứa món thịt hầm thầy tu.
- Nồi có thể chứa tối đa M suất ăn.
- Có nhiều người (savage) trong bộ tộc. Mỗi người ăn khi họ đói:
  - Nếu nồi còn thức ăn, họ tự lấy một suất ăn và ăn.
  - Nếu nồi trống, người đó sẽ đánh thức người đầu bếp, rồi chờ cho đến khi đầu bếp nấu xong và làm đầy nồi trở lại.
- Đầu bếp chỉ được đánh thức khi nồi rỗng, và khi đó sẽ nấu lại M suất mới cho vào nồi.

Yêu cầu ràng buộc:

- Đồng bộ hóa hợp lý giữa nhiều savage (người ăn) và cook (đầu bếp).
- Không được để nhiều savage cùng lúc đánh thức đầu bếp khi nồi trống.
- Đảm bảo an toàn dữ liệu, không để hai savage cùng lấy suất ăn cuối cùng một cách không hợp lệ.
- Đầu bếp chỉ được đánh thức khi cần thiết, không bị đánh thức thừa.

2. Các biến sử dụng

- **servings**: Số suất ăn còn lại trong nồi.
- **mutex**:Semaphore nhị phân (hoặc mutex) dùng để đảm bảo rằng chỉ một savage được truy cập và cập nhật biến servings tại một thời điểm.
- **emptyPot**: Semaphore dùng cho savage đánh thức đầu bếp khi nồi trống.
- **fullPot**: Semaphore dùng cho đầu bếp báo cho các savage biết nồi đã đầy sau khi nấu xong.

Ngoài ra còn có các hàm:

- `getServingFromPot()`: Lấy khẩu phần ăn ra (Savages không thể gọi getServingFromPot nếu nồi trống).
- `putServingsInPot(M)`: Đẩy khẩu phần ăn vào (Đầu bếp chỉ có thể gọi putServingsInPot nếu nồi trống).
- `eat()`: thực hiện ăn.

3. Thuật toán cho đầu bếp

```
1  while (True){
2      emptyPot.wait();
3      putServingsInPot(M);
4      fullPot.signal();
5  }
```

4. Thuật toán cho savages

```
1  while (True){
2      mutex.wait();
3          if (servings == 0){
4              emptyPot.signal();
5              fullPot.wait();
6              servings = M;
7          }
8          servings-= 1;
9          getServingFromPot();
10     mutex.signal();
11     eat()
12 }
```

5. Giải thích hoạt động

- Đầu bếp chờ đợi cho đến khi nồi trống thì `putServingsInPot(M)`. Sau khi thêm thì đánh thức `fullPot.signal()`.
- Mỗi khi savages qua được mutex, anh ta sẽ kiểm tra nồi. Nếu trống thì ra hiệu cho đầu bếp và đợi. Nếu không thì giảm khẩu phần ăn trong nồi.

3.2 Bài toán về tiệm cắt tóc (The Barbershop Problem)

1. Mô tả bài toán

- Một thợ cắt tóc làm việc trong một tiệm có ghế cắt tóc và một số lượng ghế chờ có giới hạn (n ghế).
- Nếu không có khách, thợ cắt tóc ngủ.
- Khi khách đến:
  - Nếu còn ghế chờ trống, khách sẽ ngồi đợi đến lượt mình.
  - Nếu không còn chỗ ngồi, khách bỏ đi.
- Khi thợ cắt tóc xong cho một khách, anh ta sẽ gọi khách tiếp theo từ hàng đợi (nếu có).

Yêu cầu của bài toán:

- Quản lý đồng thời giữa:
  - Một thợ cắt tóc (barber).
  - Nhiều khách hàng (customer threads).
  - Số lượng ghế chờ có giới hạn.
- Đồng bộ việc:
  - Khách chờ đến lượt, mỗi khách hàng gọi `getHairCut()`.
  - Thợ cắt tóc chỉ cắt khi có khách, thợ cắt tóc gọi `cutHair()`.
  - Nếu tiệm đầy, khách gọi `balk()` và rời đi ngay lập tức.
  - Tại mọi thời điểm, khi `cutHair()` đang chạy thì chỉ đúng 1 khách hàng đang thực thi `getHairCut()`.

2. Các biến sử dụng

- **mutex**: Semaphore nhị phân bảo vệ biến waiting.
- **customer**: Semaphore báo hiệu cho barber khi có khách.
- **customers**: Số lượng khách hàng trong cửa hàng, được bảo vệ bởi **mutex**.
- **customerDone**: Semaphore nhị phân thông báo khách đã cắt xong.
- **barberDone** : Semaphore nhị phân thông báo thợ cắt tóc cắt xong

3. Thuật toán cho khách hàng

```
mutex.wait();
    if (customers == n){
        mutex.signal();
        balk();
    }
    customers += 1;
 mutex.signal();

 customer.signal();
 barber.wait();

 getHairCut();

 customerDone.signal();
 barberDone.wait();

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

4. Thuật toán cho thợ cắt tóc

```
customer.wait();
barber.signal();
cutHair();
customerDone.wait();
barberDone.signal();
```

5. Giải thích hoạt động

- Khi khách hàng vào kiểm tra số lượng ghế.
  - Nếu mà số lượng khách hàng là n thì rời đi ngay lập tức.
  - Nếu không thì tăng biến **customers** lên 1.
- Sau khi kiểm tra, nếu thỏa mãn điều kiện, báo với thợ cắt tóc rằng có khách đang chờ (`customer.signal()`) và chờ thợ cắt tóc gọi mình vào ghế cắt bằng `barber.wait()`.
- Nếu một khách hàng khác đến trong khi thợ cắt tóc đang bận, thì trong lần lặp lại tiếp theo, thợ cắt tóc sẽ không ngủ mà phục vụ tiếp.
- Sau khi kết thúc cắt thì đồng bộ bằng các semaphore **customerDone** và **barberDone** (đảm bảo rằng việc cắt tóc được thực hiện trước khi thợ cắt tóc lặp lại để cho

khách hàng tiếp theo vào critical section). Khi ấy mới giảm số lượng khách đi 1 đơn vị.

6. Mở rộng bài toán: FIFO thợ cắt tóc (The FIFO Barbershop Problem)

- Các yêu cầu giống như bài toán thợ cắt tóc ở trên.
- Ngoài ra, bài toán này còn thêm yêu cầu là khách hàng phải được phục vụ theo đúng thứ tự họ đến (FIFO – First In First Out).

6.1. Các biến sử dụng

- Ngoài các biến được sử dụng trong bài toán trên, ta thêm hàng đợi **queue** và bỏ đi biến **barber**. Ngoài ra ta sẽ khởi tạo riêng từng semaphore cho từng khách hàng.

6.2. Thuật toán cho khách hàng

```
sem_t* mySem = new sem_t;
sem_init(mySem, 0, 0);
mutex.wait();
 if (customers == n){
    mutex.signal();
    balk();
 }
    customers += 1;
    queue.push(mySem);
 mutex.signal();

 customer.signal();
 mySem.wait();

 getHairCut();

 customerDone.signal();
 barberDone.wait();

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

6.3. Thuật toán cho thợ cắt tóc

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

sem.signal();

cutHair();

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

6.4. Giải thích thuật toán

- Tương tự như thuật toán phần trên, nhưng ta thêm queue là hàng đợi FIFO chứa semaphore riêng của từng khách. Khi khách đến thì xếp semaphore của khách này vào hàng đợi bằng câu lệnh `queue.push(mySem)`.
- Khi thợ cắt tóc thực hiện thì lấy ra theo lần lượt khách hàng đã vào hàng đợi bằng câu lệnh `sem = queue.pop()` và `sem.signal()`.

3.3 Bài toán vấn đề tiệm cắt tóc của Hilzer (Hilzer's Barbershop problem)

1. Mô tả bài toán

- Một tiệm cắt tóc có:
  - 3 thợ cắt tóc và 3 ghế cắt tóc.
  - 1 quầy thu ngân (chỉ phục vụ từng người).
  - 1 ghế sofa 4 chỗ cho khách chờ. Nếu sofa đầy thì khách sẽ đứng đợi.
  - Tối đa 20 khách được phép trong tiệm (do quy định an toàn cháy nổ).
- Hành vi của khách hàng. Khách thực hiện lần lượt gọi:
  - `enterShop`: Vào tiệm nếu chưa đủ 20 người.
  - `sitOnSofa`: Ngồi vào sofa nếu còn chỗ.
  - `getHairCut`: Được cắt tóc nếu có thợ trống.
  - `pay`: Trả tiền tại quầy thu ngân.
- Hành vi của thợ cắt tóc. Thợ thực hiện vòng lặp:
  - `cutHair`: Cắt tóc cho khách nếu có người chờ.
  - `acceptPayment`: Nhận tiền từ khách (theo thứ tự).
  - Khách hàng không thể gọi `enterShop` nếu cửa hàng hết công suất.
  - Nếu ghế sofa đầy, khách hàng đến không thể gọi `sitOnSofa`.
  - Khi khách hàng gọi `getHairCut` sẽ có một thợ cắt tóc tương ứng thực hiện `cutHair` đồng thời và ngược lại
  - Có thể có tối đa ba khách hàng thực hiện `getHairCut` đồng thời và tối đa ba thợ cắt tóc thực hiện `cutHair` đồng thời.
  - Khách hàng phải `pay` trước khi thợ cắt tóc có thể `acceptPayment`.
  - Thợ cắt tóc phải `acceptPayment` trước khi khách hàng có thể thoát.

2. Các biến sử dụng

- **n = 20**: số lượng khách tối đa trong quán
- **customers**: Biến đếm số khách hiện đang có trong tiệm.
- **mutex**: Bảo vệ biến customers và hàng đợi queue1.
- **sofa**: Biến biểu thị chỉ cho phép tối đa 4 khách ngồi sofa.

- **customer1**: Semaphore báo hiệu có khách đang đợi lên sofa.
- **customer2**: Semaphore báo hiệu có khách đang đợi được cắt tóc (trên ghế cắt tóc).
- **barber**: Semaphore báo hiệu có thợ cắt tóc sẵn sàng cắt tóc cho khách.
- **payment**: Semaphore để chỉ khách dùng để báo đã trả tiền.
- **receipt**: Semaphore báo hiệu thợ cắt tóc đã nhận tiền hay chưa.
- **queue1**: Hàng đợi FIFO gồm các semaphore của khách đang chờ ngồi lên sofa.
- **queue2**: Hàng đợi FIFO gồm các semaphore của khách đang chờ được cắt tóc.
- **mutex2**: Semaphore bảo vệ **queue2**.

3. Thuật toán cho khách hàng

```
sem_init(sem1, 0, 0), sem_init(sem2, 0, 0);
mutex.wait();
    if (customers == n){
        mutex.signal();
        balk();
    }
    customers += 1;
    queue1.push(sem1);
mutex.signal();

enterShop();
customer1.signal();
sem1.wait();

sofa.wait();
    sitOnSofa();
    sem1.signal();
    mutex.wait();
        queue2.push(sem2);
    mutex.signal();
    customer2.signal();
    sem2.wait();
sofa.signal();

sitInBarberChair();

pay();
payment.signal();
receipt.wait();
```

```
30
31 mutex.wait();
32     customers-= 1;
33 mutex.signal();
```

4. Thuật toán cho thợ cắt tóc

```
1  customer1.wait();
2  mutex.wait();
3      sem = queue1.pop();
4      sem.signal();
5      sem.wait();
6  mutex.signal();
7  sem.signal();
8
9  customer2.wait();
10 mutex.wait();
11     sem = queue2.pop(0);
12 mutex.signal();
13 sem.signal();
14
15 barber.signal();
16 cutHair();
17 payment.wait();
18 acceptPayment();
19  receipt.signal();
```

5. Giải thích thuật toán

- Thuật toán đối với khách hàng theo một thứ tự như sau: Đợi vào tiệm nếu chưa đủ 20 người $\longrightarrow$ Đợi lên sofa theo thứ tự đến $\longrightarrow$ Đợi lên ghế cắt tóc khi có barber rảnh $\longrightarrow$ Thanh toán và chờ xác nhận $\longrightarrow$ Rời khỏi tiệm.

- Khi khách hàng thoát khỏi hàng đợi, nó sẽ đi vào multiplex, ngồi trên ghế dài và thêm chính nó vào hàng đợi thứ hai.

- Đối với mỗi thợ cắt tóc đợi một khách hàng vào, ra hiệu cho semaphore của khách hàng để đưa nó ra khỏi hàng đợi, sau đó đợi họ yêu cầu một chỗ ngồi trên ghế sofa. Điều này thực thi yêu cầu FIFO.

- Mỗi thợ cắt tóc nhận một khách hàng vào ghế, vì vậy có thể cắt tóc đồng thời ba lần. Vì chỉ có một máy tính tiền, khách hàng phải lấy **mutex**. Khách hàng và thợ cắt tóc gặp nhau tại quầy thu ngân, sau đó cả hai đều thoát ra.

3.4 Bài toán Ông già Noel (Santa Claus Problem)

1. Mô tả bài toán

Ông già Noel chỉ thức dậy khi:

- Cả 9 con tuần lộc đã quay về từ kỳ nghỉ.
- Hoặc 3 yêu tinh cùng lúc gặp sự cố trong việc làm đồ chơi.

Yêu cầu đồng bộ:

- Khi 9 tuần lộc đều có mặt, ông Noel sẽ gọi `prepareSleigh()`, sau đó 9 tuần lộc sẽ gọi `getHitched()`.
- Khi 3 yêu tinh gặp sự cố, ông Noel sẽ gọi `helpElves()`, và cả 3 yêu tinh sẽ gọi `getHelp()` đồng thời.
- Các yêu tinh khác phải chờ đến khi nhóm 3 yêu tinh trước được giúp xong mới được vào.
- Nếu cả 9 tuần lộc và 3 yêu tinh cùng lúc chờ, ông Noel ưu tiên xử lý tuần lộc.

2. Các biến sử dụng

- **elves, reindeer**: Biến đếm số yêu tinh và tuần lộc đang chờ.
- **santaSem**: Semaphore để đánh thức ông Noel.
- **reindeerSem**: Semaphore cho các tuần lộc chờ đến khi được gọi để kéo xe.
- **elfTex**: Semaphore nhị phân để chặn yêu tinh thứ 4 trở đi trong khi 3 yêu tinh đang được giúp.
- **mutex**: Semaphore nhị phân để bảo vệ các biến đếm khi truy cập đồng thời.

3. Thuật toán cho Ông già Noel

```
santaSem.wait();
mutex.wait();
if (reindeer == 9) {
    prepareSleigh();
    reindeerSem.signal(9);
    reindeer -= 9;
} else if (elves == 3) {
    helpElves();
}
mutex.signal();
```

4. Thuật toán cho tuần lộc

```
mutex.wait();
reindeer += 1;
if (reindeer == 9) {
    santaSem.signal();
}
mutex.signal();

reindeerSem.wait();
getHitched();
```

5. Thuật toán cho 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();
```

6. Giải thích hoạt động

- Tuần lộc: Khi tuần lộc cuối cùng (thứ 9) đến, nó đánh thức ông Noel. Sau đó, cả 9 tuần lộc chờ ông gọi để được kéo xe.
- Yêu tinh: Chỉ 3 yêu tinh được phép vào gặp ông Noel cùng lúc. Những yêu tinh khác bị chặn bởi `elfTex`.
- Ông Noel: Chạy trong vòng lặp vô hạn, chỉ thức dậy khi được đánh thức bởi đủ 9 tuần lộc hoặc 3 yêu tinh. Nếu cả hai cùng đến, ông ưu tiên tuần lộc.
- Sau khi 3 yêu tinh được giúp xong, yêu tinh cuối cùng sẽ giải phóng `elfTex` để nhóm tiếp theo có thể vào.

3.5 Xây dựng phân tử H_2O (Building H_2O)

1. Mô tả bài toán

Bài toán yêu cầu đồng bộ các luồng **oxy** và **hydro** để tạo thành phân tử nước (H_2O). Mỗi phân tử nước cần đúng **2 hydro** và **1 oxy**. Các luồng chỉ được tiếp tục khi đủ bộ ba này cùng nhau gọi hàm `bond()`.

- Nếu một luồng **oxy** đến mà chưa có đủ hai **hydro** đang chờ, nó phải đợi.
- Nếu một luồng **hydro** đến mà chưa có một **oxy** và một **hydro** khác, nó cũng phải đợi.
- Không cần xác định chính xác ba luồng nào ghép với nhau, chỉ cần đảm bảo các luồng được xử lý theo nhóm đúng tỉ lệ.

2. Các biến sử dụng

- **mutex**: Semaphore nhị phân (1) để đảm bảo truy cập đồng thời an toàn cho các biến đếm.
- **oxygen, hydrogen**: Các biến đếm số luồng đang chờ.
- **barrier**: Hàng rào để đảm bảo cả ba luồng gọi `bond()` trước khi các luồng tiếp theo được ghép.
- **oxyQueue, hydroQueue**: Semaphore để luồng oxy và hydro chờ khi chưa đủ bộ ba.

3. Thuật toán cho luồng Oxy

```
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();
```

4. Thuật toán cho luồng Hydro

```
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();
```

5. Giải thích hoạt động

- Mỗi luồng khi đến sẽ cập nhật biến đếm và kiểm tra có thể hình thành một phân tử hay chưa.
- Nếu đủ (1 oxy + 2 hydro), semaphore sẽ đánh thức ba luồng tương ứng để chúng cùng gọi `bond()`.
- Tất cả các luồng sẽ chờ ở **barrier** để đảm bảo bộ ba đồng bộ với nhau.
- **Luồng oxy** chịu trách nhiệm giải phóng `mutex` sau khi hàng rào mở, đảm bảo rằng mỗi nhóm hoàn thành trước khi nhóm sau bắt đầu.
- Việc chỉ có duy nhất một luồng oxy trong mỗi nhóm giúp tránh việc nhiều luồng cùng giải phóng `mutex`.

3.6 Vấn đề qua sông (River crossing problem)

1. Mô tả bài toán

Bài toán yêu cầu đồng bộ các luồng **hacker** và **serf** (nông dân) để họ có thể cùng nhau qua sông. Chiếc thuyền có thể chở chính xác 4 người, không nhiều hơn và cũng không ít hơn. Để đảm bảo an toàn, không thể để một hacker đi cùng ba serf, hoặc một serf đi cùng ba hacker. Các kết hợp còn lại đều an toàn.

Mỗi khi một luồng đến, nó cần gọi hàm `board`. Chúng ta cần đảm bảo rằng tất cả bốn luồng từ mỗi chuyến thuyền đều gọi `board` trước khi bất kỳ luồng nào từ chuyến thuyền tiếp theo có thể được xử lý.

Sau khi tất cả bốn luồng đã gọi `board`, chỉ một trong số chúng cần gọi hàm `rowBoat` để chỉ định luồng đó sẽ cầm mái chèo. Không quan trọng luồng nào gọi hàm này, miễn là có một luồng thực hiện.

Chúng ta chỉ quan tâm đến luồng di chuyển theo một hướng cụ thể.

2. Các biến sử dụng

- **mutex**: Semaphore nhị phân (1) để đảm bảo việc truy cập đồng thời an toàn cho các biến đếm.
- **hackers, serfs**: Các biến đếm số luồng hacker và serf đang chờ.
- **barrier**: Hàng rào để đảm bảo tất cả bốn luồng gọi `board()` trước khi một trong chúng thực hiện `rowBoat()`.
- **hackerQueue, serfQueue**: Semaphore để điều khiển số lượng hacker và serf có thể qua sông đồng thời.
- **isCaptain**: Biến boolean cho biết luồng nào là thuyền trưởng (gọi hàm `rowBoat`).

3. Thuật toán

```
mutex.wait();
hackers += 1;
if (hackers == 4) {
    hackerQueue.signal(4);
    hackers = 0;
    isCaptain = True;
} else if (hackers == 2 and serfs >= 2) {
    hackerQueue.signal(2);
    serfQueue.signal(2);
    serfs -= 2;
    hackers = 0;
```

```
12      isCaptain = True;
13  } else {
14      mutex.signal();
15  }
16
17  hackerQueue.wait();
18  board();
19  barrier.wait();
20
21  if (isCaptain) {
22      rowBoat();
23  }
24  mutex.signal();
```

4. Giải thích hoạt động

- Mỗi luồng khi đến sẽ cập nhật biến đếm và kiểm tra xem có thể hình thành một nhóm đủ bốn người không.
- Nếu có đủ hacker và serf, semaphore sẽ đánh thức các luồng tương ứng để chúng cùng gọi `board()`.
- Các luồng sẽ chờ ở **barrier** cho đến khi tất cả các luồng trong nhóm cùng gọi `board()`.
- **Luồng thuyền trưởng** sẽ gọi `rowBoat` và giải phóng `mutex` sau khi thuyền đã rời bến.
- Hàng rào **barrier** giúp đồng bộ tất cả các luồng trong nhóm trước khi bắt đầu di chuyển.
- Biến **isCaptain** xác định luồng nào sẽ gọi `rowBoat`.

3.7 Bài toán Tàu lượn siêu tốc (Roller Coaster Problem)

1. Mô tả bài toán

Giả sử có n luồng hành khách và một luồng đại diện cho chiếc xe tàu lượn. Hành khách liên tục xếp hàng để được lên xe và đi một vòng. Mỗi xe chỉ chở được đúng C hành khách tại một thời điểm ($C < n$). Xe chỉ được khởi hành khi đã đầy chỗ.

Các ràng buộc đồng bộ:

- Hành khách chỉ được gọi `board()` sau khi xe gọi `load()`.
- Xe chỉ được khởi hành sau khi đủ C hành khách đã gọi `board()`.
- Hành khách chỉ được gọi `unboard()` sau khi xe gọi `unload()`.

Mục tiêu là viết mã đồng bộ đảm bảo các ràng buộc trên được giữ đúng.

2. Các biến sử dụng

- **mutex, mutex2**: Semaphore nhị phân dùng để bảo vệ truy cập tới biến đếm số hành khách lên và xuống.
- **boarders, unboarders**: Biến đếm số hành khách đã lên hoặc đã xuống xe.
- **boardQueue, unboardQueue**: Semaphore để điều phối hành khách chờ lên hoặc xuống xe.
- **allAboard, allAshore**: Semaphore để xe chờ cho đến khi tất cả hành khách đã lên hoặc đã xuống.

3. Thuật toán cho xe (1 xe)

```
load();
boardQueue.signal(C);
allAboard.wait();

run();

unload();
unboardQueue.signal(C);
allAshore.wait();
```

4. Thuật toán cho hành khách

```
boardQueue.wait();
board();

mutex.wait();
boarders += 1;
if (boarders == C) {
    allAboard.signal();
    boarders = 0;
}
mutex.signal();

unboardQueue.wait();
unboard();

mutex2.wait();
unboarders += 1;
if (unboarders == C) {
    allAshore.signal();
    unboarders = 0;
}
mutex2.signal();
```

5. Giải thích hoạt động

- Xe gọi `load()` để bắt đầu quy trình đón hành khách. Sau đó, nó mở semaphore `boardQueue` C lần để cho phép C hành khách được lên xe.
- Mỗi hành khách khi được phép lên xe sẽ gọi `board()`, sau đó tăng biến `boarders`.
- Hành khách cuối cùng (người thứ C) sẽ gửi tín hiệu qua semaphore `allAboard` để thông báo cho xe rằng tất cả hành khách đã lên xe.
- Xe bắt đầu chạy (`run()`) sau đó dừng lại và gọi `unload()` để chuẩn bị cho hành khách xuống.
- Xe mở semaphore `unboardQueue` C lần để cho phép C hành khách xuống xe.
- Tương tự như khi lên xe, hành khách gọi `unboard()` và người cuối cùng sẽ báo hiệu cho xe bằng `allAshore`.
- Xe chỉ tiếp tục quy trình sau khi tất cả hành khách đã xuống.

6.1 Mở rộng: Nhiều xe chạy đồng thời

Giả sử có m xe chạy đồng thời. Một số ràng buộc mới được đặt ra:

- Chỉ một xe được phép cho hành khách lên tại một thời điểm (tránh xung đột).
- Nhiều xe có thể chạy cùng lúc trên đường ray.
- Việc hành khách lên và xuống xe phải diễn ra theo thứ tự tuần tự giữa các xe.

Sử dụng 2 mảng semaphore `loadingArea[]` và `unloadingArea[]` để điều phối thứ tự xe hoạt động.

Hàm xác định xe tiếp theo:

```
int next(int i){
    return (i + 1) % m;
}
```

6.2 Thuật toán cho xe (nhiều xe)

```
loadingArea[i].wait();
load();
boardQueue.signal(C);
allAboard.wait();
loadingArea[next(i)].signal();

run();

unloadingArea[i].wait();
unload();
unboardQueue.signal(C);
allAshore.wait();
unloadingArea[next(i)].signal();
```

6.3 Thuật toán cho hành khách (giữ nguyên)

```
boardQueue.wait();
board();

mutex.wait();
boarders += 1;
if (boarders == C) {
    allAboard.signal();
    boarders = 0;
}
mutex.signal();

unboardQueue.wait();
unboard();
```

```
14
15 mutex2.wait();
16 unboarders += 1;
17 if (unboarders == C) {
18     allAshore.signal();
19     unboarders = 0;
20 }
21 mutex2.signal();
```

6.4 Giải thích hoạt động (nhiều xe)

- Xe thứ *i* phải chờ tại `loadingArea[i]` trước khi có quyền cho hành khách lên xe.
- Sau khi đã cho *C* hành khách lên xe và nhận tín hiệu `allAboard`, xe kích hoạt `loadingArea[next(i)]` để cho xe kế tiếp hoạt động.
- Quá trình xuống xe diễn ra tương tự, được điều phối qua `unloadingArea[i]` và tiếp nối bởi `unloadingArea[next(i)]`.
- Điều này giúp đảm bảo việc lên và xuống xe diễn ra tuần tự, tránh xung đột trong việc sử dụng các semaphore chung như `boardQueue` và `unboardQueue`.

3.8 Bài toán Người hút thuốc lá (The Cigarette Smokers Problem)

1. Mô tả bài toán

Đây là một bài toán đồng bộ kinh điển, mô phỏng sự phối hợp giữa một tiến trình trung tâm (agent) và ba tiến trình khác (người hút thuốc), với bối cảnh như sau:

- Một agent mỗi lần đặt ngẫu nhiên hai trong ba loại nguyên liệu sau lên bàn: *thuốc lá (tobacco)*, *giấy (paper)*, và *diêm (match)*.
- Có ba người hút thuốc, mỗi người sở hữu vô hạn một loại nguyên liệu:
  - Người thứ nhất có sẵn thuốc lá.
  - Người thứ hai có sẵn giấy.
  - Người thứ ba có sẵn diêm.
- Người hút thuốc chỉ có thể hút nếu lấy được hai nguyên liệu còn lại từ bàn.
- Sau khi hút xong, người đó sẽ thông báo cho agent để tiếp tục cấp phát nguyên liệu mới.

2. Yêu cầu ràng buộc

- Chỉ đúng người hút thuốc có đủ nguyên liệu mới được phép hành động.
- Tránh việc nhiều người cùng tranh nguyên liệu không phù hợp (gây sai sót hoặc deadlock).
- Giữ nguyên code của agent (tức là agent chỉ đơn thuần cấp phát hai nguyên liệu ngẫu nhiên).

3. Các biến sử dụng

- **agentSem**: Semaphore dùng để đồng bộ giữa agent và smokers. Agent chỉ đặt nguyên liệu khi được phép.
- **tobacco**, **paper**, **match**: Semaphore báo hiệu nguyên liệu được đặt lên bàn.
- **mutex**: Đảm bảo độc quyền khi pusher truy cập vào trạng thái bàn.
- **isTobacco**, **isPaper**, **isMatch**: Biến boolean đại diện cho trạng thái bàn hiện có nguyên liệu gì.
- **smokerWithTobacco**, **smokerWithPaper**, **smokerWithMatch**: Semaphore đánh thức đúng người hút thuốc tương ứng.

4. Thuật toán cho agent

Mỗi agent chỉ đơn thuần đặt hai nguyên liệu lên bàn. Ví dụ:

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

Các agent khác tương tự với các tổ hợp còn lại: (paper, match) và (tobacco, match).

5. Thuật toán pusher (trung gian)

Mỗi nguyên liệu có một pusher riêng để ghi nhận trạng thái nguyên liệu và đánh thức đúng người hút thuốc.

Ví dụ, với pusher thuốc lá:

```
tobacco.wait();
mutex.wait();
if (isPaper) {
isPaper = false;
smokerWithMatch.signal();
} else if (isMatch) {
isMatch = false;
smokerWithPaper.signal();
} else {
isTobacco = true;
}
mutex.signal();
```

Tương tự với pusher của giấy và diêm.

6. Thuật toán cho smoker

Khi được đánh thức bởi pusher, người hút thuốc sẽ thực hiện:

```
smokerWithMatch.wait();
makeCigarette();
agentSem.signal();
smoke();
```

Tương tự cho smoker có giấy và thuốc lá.

7. Giải thích hoạt động

- Mỗi lần agent đặt hai nguyên liệu, đúng một pusher sẽ xử lý và xác định loại người hút thuốc có thể hành động.
- Biến trạng thái `isTobacco`, `isPaper`, `isMatch` đảm bảo kiểm soát trạng thái bàn

hiện tại.

- Cơ chế semaphore và mutex giúp đảm bảo đồng bộ hóa, tránh deadlock và đánh thức sai tiến trình.

4 Tổng kết

Đèn báo là một công cụ quan trọng trong việc giải quyết vấn đề đồng bộ hóa giữa các tiến trình. Mặc dù có nhiều bài toán thú vị liên quan đến đèn báo, nhưng trong khoảng thời gian có hạn của kỳ học 2024-2, nhóm chúng em chỉ đưa ra một số bài toán tiêu biểu được trích từ cuốn sách [1].

Chúng em xin gửi lời cảm ơn chân thành tới thầy Phạm Đăng Hải vì những bài giảng giá trị và sự tận tâm trong môn Nguyên lý hệ điều hành, giúp chúng em có thêm kiến thức và kinh nghiệm trong quá trình nghiên cứu, qua đó có thể hoàn thành tài liệu này.

Mọi ý kiến đóng góp nhằm hoàn thiện tài liệu xin vui lòng gửi cho chúng em về hai địa chỉ email sau: `caohuythinh@gmail.com` và `ndt.minh2005@gmail.com`. Nhóm chúng em rất trân trọng mọi phản hồi từ thầy và các bạn.

Tài liệu tham khảo

[1] A. B. Downey, *The Little Book of Semaphores*, vol. 2. Green Tea Press, 2008.