Điều độ tiến trình trong hệ điều hành: phân tích và giải pháp quản lý đoạn găng

Lưu Thịnh Khang[1], Nguyễn Viết Tuấn Kiệt[1,2]
Bùi Quang Phong[1,2], Nguyễn Thanh Tuyển[1]

[1] *Chương trình tài năng - Khoa học máy tính K67*[*]
[2] *Phòng thí nghiệm Mô hình hóa, Mô phỏng và Tối ưu hóa*[†]

Ngày 12 tháng 6 năm 2025

Tóm tắt nội dung

Bài tập lớn này tập trung nghiên cứu và trình bày các cơ chế, thuật toán điều độ tiến trình trong quản lý đoạn găng của hệ điều hành. Báo cáo nhấn mạnh vào việc phân tích các bài toán kinh điển trong hệ điều hành và đề xuất các phương án giải quyết. Các vấn đề tương tranh, bế tắc, chết đói và khóa sống được phân tích chi tiết cùng với các giải pháp đồng bộ hóa như Mutex, Semaphore, và Monitor. Các bài toán điển hình như Người hút thuốc lá, Chờ xe buýt, Người nhập cư và Quán rượu Sushi được sử dụng để minh họa các phương pháp đồng bộ hóa tiến trình. Báo cáo cung cấp cái nhìn tổng quan về cách các hệ điều hành hiện đại duy trì tính nhất quán và hiệu suất hệ thống, đồng thời giải quyết các vấn đề quản lý tài nguyên.

1 Giới thiệu

1.1 Tiến trình và luồng

Tiến trình (process) là một trong những khái niệm cơ bản nhất trong lĩnh vực hệ điều hành và là đơn vị cơ bản của sự phân bổ tài nguyên và thực thi. Một tiến trình thường được định nghĩa là một chương trình đang thực thi [2], bao gồm mã thực thi của chương trình, các biến toàn cục, ngăn xếp (stack), và vùng nhớ động (heap). Điều quan trọng là phân biệt giữa chương trình, là thực thể thụ động chứa một danh sách các chỉ thị được lưu trữ trên đĩa, và tiến trình, là thực thể động với bộ đếm chương trình (program counter) chỉ ra chỉ thị tiếp theo cần thực hiện và một tập hợp các tài nguyên liên kết.

[*]Trường Công nghệ thông tin và Truyền thông, Đại học Bách khoa Hà Nội
[†]Trung tâm nghiên cứu quốc tế về trí tuệ nhân tạo, BKAI

Khi một tệp thực thi được tải vào bộ nhớ, nó trở thành một tiến trình. Có nhiều cách để khởi động một tiến trình, từ việc nhấp đúp vào biểu tượng đại diện cho tệp thực thi cho đến việc nhập tên của tệp thực thi trong dòng lệnh của hệ điều hành. Ví dụ, khi người dùng muốn chạy một chương trình được biên dịch trong Java là `Program.class`, người đó có thể nhập lệnh:

```
java Program
```

để khởi động Máy ảo Java (JVM), một tiến trình, để thực thi chương trình.

Cấu trúc bộ nhớ của một tiến trình [4] có thể được chia thành nhiều phần, mỗi phần đảm nhận các nhiệm vụ riêng biệt:

- **Phần mã thực thi** (Text Section): Đây là khu vực chứa mã thực thi của chương trình. Kích thước của phần này thường cố định vì nó không thay đổi trong suốt thời gian chạy của chương trình.
- **Phần dữ liệu** (Data Section): Phần này chứa các biến toàn cục và tĩnh của chương trình. Giống như phần mã thực thi, kích thước của phần dữ liệu cũng cố định trong suốt quá trình chạy chương trình.
- **Phần ngăn xếp** (Stack Section): Ngăn xếp chứa dữ liệu tạm thời như tham số hàm, địa chỉ trở về và biến cục bộ. Mỗi lần một hàm được gọi, một bản ghi kích hoạt được đẩy vào ngăn xếp, và khi hàm trả lại, bản ghi này được lấy ra. Kích thước của ngăn xếp có thể thay đổi động trong suốt quá trình thực thi chương trình.
- **Phần vùng nhớ động** (Heap Section): Đây là khu vực bộ nhớ được cấp phát động khi chương trình yêu cầu thêm bộ nhớ, ví dụ thông qua các hàm như `malloc` trong C. Kích thước của heap có thể tăng lên hoặc giảm xuống tùy thuộc vào nhu cầu của chương trình.

Hệ điều hành có trách nhiệm đảm bảo rằng ngăn xếp và heap không tràn vào nhau, điều này có thể dẫn đến lỗi và sự cố hệ thống. Để quản lý điều này, hệ điều hành sử dụng các kỹ thuật như kiểm tra giới hạn ngăn xếp và cấp phát trang linh hoạt cho heap.

Như đã đề cập, tiến trình là một chương trình đang thực thi, được hệ thống hỗ trợ bằng cách cung cấp các tài nguyên cần thiết như bộ nhớ, thời gian CPU, và tài nguyên hệ thống. Mỗi tiến trình có không gian địa chỉ riêng, cho phép nó hoạt động một cách độc lập với các tiến trình khác. Điều này đảm bảo rằng sự cố trong một tiến trình không ảnh hưởng trực tiếp đến các tiến trình khác. Tuy nhiên, mỗi tiến trình độc lập cũng đồng nghĩa với việc có chi phí cao hơn về mặt tạo và quản lý, bao gồm chi phí liên quan đến việc khởi tạo không gian bộ nhớ và thời gian chuyển đổi giữa các tiến trình.

Trong khi tiến trình là một thực thể hoàn toàn độc lập, luồng lại là một phần của tiến trình, cho phép thực thi đồng thời nhiều tác vụ trong cùng một ứng dụng. Luồng (thread), còn được gọi là "tiến trình nhẹ" (lightweight process), chia sẻ không gian bộ nhớ và tài nguyên với các luồng khác trong cùng một tiến trình, điều này làm giảm đáng kể chi phí về mặt tài nguyên so với việc sử dụng nhiều tiến trình. Mỗi luồng có bộ đếm chương trình riêng, ngăn xếp riêng, và trạng thái riêng, nhưng chia sẻ dữ liệu và các tài nguyên khác như bộ nhớ heap. Điều này cho phép luồng thực hiện tác vụ một cách hiệu quả, đồng thời giảm thiểu thời gian phản hồi và tăng cường khả năng phản ứng của ứng dụng.

Sự khác biệt chính giữa tiến trình và luồng nằm ở mức độ chia sẻ tài nguyên. Tiến trình cung cấp môi trường độc lập cho mỗi ứng dụng, trong khi luồng tận dụng hiệu quả tài nguyên đã có bằng cách chia sẻ chúng với các luồng khác trong cùng một tiến trình. Do đó, luồng thích hợp hơn cho các ứng dụng yêu cầu hiệu suất cao và đa nhiệm trong khi tiến trình phù hợp với các tác vụ cần mức độ bảo mật và độc lập cao hơn.

1.2 Đồng bộ tiến trình

Đồng bộ tiến trình là một khái niệm trọng tâm trong hệ điều hành và lập trình đa luồng, nhằm đảm bảo rằng các tiến trình hoặc luồng khác nhau hoạt động một cách có trật tự và hợp lý khi truy cập vào các tài nguyên chung hoặc thực hiện các hoạt động có liên quan đến nhau. Các kỹ thuật đồng bộ hóa giúp phòng tránh tình trạng đua (race conditions), deadlocks và giúp duy trì tính nhất quán dữ liệu.

Trong một hệ thống đa tiến trình hoặc đa luồng, tiến trình hoặc luồng có thể cần truy cập vào các tài nguyên chung, như bộ nhớ, tập tin, hoặc các biến được chia sẻ. Việc không có đồng bộ hóa có thể dẫn đến các vấn đề về tính nhất quán dữ liệu, gây ra kết quả không dự đoán được hoặc sai lệch. Vì vậy, đồng bộ hóa tiến trình là thiết yếu để đảm bảo rằng mỗi tiến trình hoặc luồng thực hiện các hoạt động của nó một cách hợp lý và an toàn.

1.2.1 Tài nguyên găng

Trong một hệ thống máy tính, "tài nguyên găng" (critical resource) là những tài nguyên có tính chất quan trọng đến mức việc truy cập vào chúng phải được quản lý một cách cẩn thận để tránh xung đột và đảm bảo hoạt động ổn định và hiệu quả của hệ thống. Tài nguyên găng thường bao gồm các thành phần phần cứng hoặc phần mềm mà có khả năng ảnh hưởng trực tiếp đến nhiều tiến trình hoặc luồng trong một hệ thống. Việc quản lý chúng đòi hỏi sự đồng bộ hóa chặt chẽ, vì việc truy cập không kiểm soát có thể dẫn đến các vấn đề nghiêm trọng như deadlock, starvation, hoặc race conditions.

Một số ví dụ về tài nguyên găng:

- **Bộ nhớ chia sẻ:** Trong môi trường đa tiến trình hoặc đa luồng, bộ nhớ chia sẻ (như heap memory) là một tài nguyên găng vì nhiều tiến trình có thể cố gắng truy cập và thay đổi nội dung của nó đồng thời.
- **Cơ sở dữ liệu:** Đây là tài nguyên găng khi nhiều ứng dụng hoặc dịch vụ cần đọc và ghi dữ liệu từ cùng một nguồn.
- **Tệp hệ thống:** Các tệp quan trọng như log files hoặc configuration files, khi được nhiều tiến trình sử dụng cùng lúc, cần được quản lý để tránh việc ghi đè hoặc mất mát dữ liệu.
- **Thiết bị ngoại vi:** Máy in, scanner, và các thiết bị ngoại vi khác, nếu không được quản lý truy cập một cách hợp lý, có thể gây ra xung đột và lỗi hệ thống.
- **Ports và Socket:** Trong lập trình mạng, port và socket là tài nguyên găng khi chúng được nhiều ứng dụng sử dụng để gửi và nhận dữ liệu qua mạng.

Những tài nguyên này thường có đặc điểm chung là không thể được nhân bản một cách dễ dàng hoặc có giới hạn về số lượng sử dụng đồng thời, dẫn đến việc cần phải có cơ chế đồng bộ hóa để quản lý truy cập. Tính chất không thể chia sẻ của tài nguyên găng là yếu tố trung tâm trong việc xác định chúng. Ví dụ, một máy in không thể phục vụ hai yêu cầu in cùng một lúc mà không có sự quản lý để xếp hàng hoặc phân bổ nhiệm vụ. Tương tự, truy cập vào một cơ sở dữ liệu phải được điều chỉnh để tránh việc các truy vấn đồng thời ghi đè lên nhau, điều này có thể dẫn đến mất dữ liệu hoặc kết quả không nhất quán.

Một đặc điểm khác của tài nguyên găng là sự cần thiết của việc bảo vệ tính toàn vẹn dữ liệu. Khi nhiều tiến trình cố gắng sử dụng cùng một tài nguyên, như bộ nhớ chia sẻ hoặc tập tin cấu hình, việc đảm bảo rằng dữ liệu không bị hư hỏng hoặc thay đổi một cách không mong muốn là cực kỳ quan trọng. Các cơ chế như khóa (locks), semaphores, hoặc biến điều kiện thường được sử dụng để giải quyết vấn đề này, giúp đảm bảo rằng mọi thay đổi đều được kiểm soát và tuân theo các quy tắc nhất định.

Cuối cùng, tài nguyên găng cũng yêu cầu một cách tiếp cận linh hoạt trong quản lý. Vì mỗi tài nguyên có thể có đặc điểm và yêu cầu riêng, phương pháp quản lý phù hợp cho một tài nguyên có thể không hiệu quả cho một tài nguyên khác. Ví dụ, một ổ đĩa có thể cần một chiến lược quản lý khác biệt so với một cổng mạng, mặc dù cả hai đều là tài nguyên găng. Do đó, hiểu biết về bản chất và yêu cầu của từng loại tài nguyên găng là chìa khóa để xây dựng hệ thống đáng tin cậy và hiệu quả.

1.2.2 Đoạn găng

Đoạn găng được định nghĩa là phần của chương trình mà trong đó một luồng truy cập vào tài nguyên chung, có thể gây ra xung đột nếu không được điều phối cẩn thận. Vì tài nguyên trong đoạn găng có tính chất chia sẻ và có khả năng bị

nhiều luồng truy cập đồng thời, việc đảm bảo rằng chỉ một luồng có thể thực hiện đoạn găng tại một thời điểm là cần thiết để tránh các hậu quả không mong muốn.

Khi một luồng bước vào đoạn găng, nó cần phải được cấp quyền độc quyền để sử dụng tài nguyên cụ thể đó. Điều này đảm bảo rằng không có luồng khác có thể thay đổi tài nguyên khi luồng đang hoạt động trong đoạn găng, từ đó giữ cho tài nguyên được bảo vệ và dữ liệu liên quan ổn định.

Quản lý đoạn găng hiệu quả yêu cầu việc triển khai các chiến lược thông minh để tránh tình trạng deadlock, khi mà nhiều luồng đồng thời chờ đợi nhau giải phóng tài nguyên, hoặc starvation, nơi một hoặc nhiều luồng không bao giờ có cơ hội truy cập vào tài nguyên cần thiết. Các nhà phát triển phải lựa chọn phương pháp đồng bộ hóa phù hợp để tối ưu hóa hiệu suất và đảm bảo công bằng, đồng thời phải xem xét kỹ lưỡng trật tự và thời gian truy cập đến đoạn găng trong thiết kế ứng dụng của họ.

Ngoài ra, điều quan trọng là phải minh bạch và rõ ràng về phạm vi và định nghĩa của đoạn găng trong mã nguồn. Điều này không chỉ giúp tránh lỗi lập trình mà còn tạo điều kiện cho việc bảo trì và mở rộng mã trong tương lai. Ví dụ, một chương trình có thể có nhiều đoạn găng, mỗi đoạn đều cần quản lý riêng biệt để đảm bảo hiệu suất tối ưu và tránh gây tắc nghẽn hệ thống.

1.3 Cơ chế đồng bộ hóa

Các cơ chế đồng bộ hóa là công cụ thiết yếu trong lập trình đa luồng và hệ điều hành để quản lý truy cập vào các tài nguyên chia sẻ. Dưới đây là một số cơ chế đồng bộ hóa phổ biến được sử dụng trong lập trình và thiết kế hệ thống:

1.3.1 Mutex

Mutex là một công cụ đồng bộ hóa dùng để quản lý quyền truy cập vào tài nguyên được chia sẻ bởi nhiều luồng. Mục đích chính của mutex là ngăn chặn tình trạng "race condition", nơi mà nhiều luồng cùng truy cập và thay đổi tài nguyên chia sẻ một cách không kiểm soát được, có thể dẫn đến dữ liệu bị hỏng hoặc các lỗi khác.

Mutex hoạt động dựa trên nguyên tắc cơ bản là chỉ cho phép một luồng kiểm soát tài nguyên tại một thời điểm. Khi một luồng muốn truy cập vào một tài nguyên chia sẻ, nó phải trước tiên yêu cầu khóa mutex. Nếu mutex đó không bị khóa bởi luồng nào khác, luồng yêu cầu sẽ khóa nó và tiếp tục thực thi. Khi hoàn thành, luồng sẽ giải phóng (unlock) mutex để các luồng khác có thể sử dụng tài nguyên.

Xét ví dụ sau, giả sử có một biến số nguyên dùng chung, có thể được truy cập toàn cục bên trong chương trình. Tạo một hàm để tăng số lên 1 lên 1000000 lần bằng vòng lặp `for`. Tạo hai luồng có tên `t1` và `t2` để chạy cùng một hàm tăng `increment`. Trong trường hợp này, `t1` sẽ tăng số lên 1 với 1000000 lần và `t2` sẽ tăng số lên 1 với 1000000 lần. Vì vậy, `number` dự kiến là 2000000.

Tuy nhiên, có khả năng xảy ra tình trạng tương tranh khi nhiều luồng cố gắng sửa đổi một tài nguyên cùng lúc. Vì vậy, giá trị của số không thể dự đoán được. Đoạn mã không có `mutex`:

```cpp
#include <iostream>
#include <thread>
using namespace std;

// Tài nguyên găng
int number = 0;

// Hàm tăng biến đếm
void increment(){
    // Tăng number thêm 1, thực hiện 100000 lần
    for(int i=0; i<1000000; i++){
        number++;
    }
}

int main()
{
    // Tạo luồng t1 để thực hiện hàm increment()
    thread t1(increment);

    // Tạo luồng t1 để thực hiện hàm increment()
    thread t2(increment);

    // Bắt đầu cả 2 luồng đồng thời
    t1.join();
    t2.join();

    // In ra number sau khi chạy
    cout << "Number after execution of t1 and t2 is " << number;

    return 0;
}
```

Chạy chương trình một vài lần, kết quả `number` là không giống nhau và khó kiểm soát: $1031943, 1128145, 1043152, 1045864$. Đoạn mã sau đây sẽ sử dụng `mutex` để nhất quán dữ liệu. Kết quả in ra là 200000 trong mọi trường hợp:

```cpp
#include <iostream>
#include <thread>
using namespace std;

// import mutex từ thư viện chuẩn C++
#include <mutex>

// Tạo đối tượng mutex
mutex mtx;

// Tài nguyên găng
int number = 0;

// Hàm để tăng number
void increment(){
    // Khoá luồng bằng mutex
    mtx.lock();

    // Tăng number thêm 1, thực hiện 100000 lần
    for(int i=0; i<1000000; i++){
        number++;
    }

    // Giải phóng mutex bằng unlock
    mtx.unlock();
}

int main()
{
    // Tạo luồng t1 để thực hiện hàm increment()
    thread t1(increment);

    // Tạo luồng t2 để thực hiện hàm increment()
    thread t2(increment);

    // Bắt đầu cả 2 luồng đồng thời
    t1.join();
    t2.join();

    // In ra number sau khi chạy
    cout<<"Number after execution of t1 and t2 is "<<number;

    return 0;
}
```

1.3.2 Semaphore

Semaphore là một cấu trúc dữ liệu quan trọng trong lập trình đa luồng và hệ điều hành, được sử dụng để điều khiển truy cập vào các tài nguyên chia sẻ. Nó cơ bản là một biến nguyên dùng để đếm số lượng đơn vị tài nguyên có sẵn và kiểm soát số lượng tiến trình hoặc luồng được phép truy cập vào một tài nguyên chia sẻ cùng một lúc.

Semaphore quản lý truy cập đến tài nguyên thông qua hai thao tác cơ bản:

- **Wait (hoặc P operation):** Thao tác này được sử dụng bởi một tiến trình hoặc luồng để yêu cầu tài nguyên. Khi một tiến trình gọi Wait, semaphore giảm giá trị của nó đi 1. Nếu giá trị của semaphore là không âm, tiến trình tiếp tục thực hiện; nếu âm, tiến trình bị chặn cho đến khi một đơn vị tài nguyên được giải phóng.
- **Signal (hoặc V operation):** Thao tác này được sử dụng để giải phóng tài nguyên. Khi một tiến trình gọi Signal, giá trị của semaphore được tăng lên 1. Nếu có tiến trình nào đang bị chặn do giá trị semaphore âm, một trong số các tiến trình đó được giải phóng.

Đoạn mã C++ sau đây là một ví dụ minh họa cách sử dụng `semaphore` để quản lý truy cập đồng thời vào một tài nguyên hoặc phần của mã.

```cpp
#include <iostream>
#include <semaphore>
#include <thread>
using namespace std;

// Một semaphore có tối đa 10 "điểm" mà các thread có thể sử dụng.
counting_semaphore<10> semaphore(3);

void worker(int id)
{
    // Luồng gọi hàm này sẽ cố gắng giảm số lượng của semaphore đi 1
    semaphore.acquire();

    // Thực hiện công việc nào đó.
    cout << "Thread " << id << " acquired the semaphore."
         << endl;

    // Giải phóng
    semaphore.release();
    cout << "Thread " << id << " released the semaphore."
         << endl;
}
```

```
23
24  int main()
25  {
26      thread t1(worker, 1);
27      thread t2(worker, 2);
28      thread t3(worker, 3);
29      t1.join();
30      t2.join();
31      t3.join();
32      return 0;
33  }
```

1.3.3 Monitor

Monitor là một cấu trúc dữ liệu đồng bộ hóa cao cấp được sử dụng trong lập trình đa luồng để kiểm soát truy cập vào các tài nguyên chia sẻ. Monitor bao gồm các biến, phương thức và các biến điều kiện bên trong một đối tượng được bảo vệ để chỉ một luồng có thể thực hiện các phương thức của đối tượng tại một thời điểm.

Monitor hoạt động dựa trên nguyên tắc rằng tất cả các phương thức tương tác với tài nguyên chia sẻ phải được bao đóng trong một đối tượng đặc biệt. Khi một luồng muốn thực hiện một hoặc nhiều hành động liên quan đến tài nguyên này, nó phải trước tiên "nhập" monitor. Trong khi một luồng đang trong monitor, không có luồng khác nào có thể thực hiện bất kỳ phương thức nào của monitor, đảm bảo rằng tài nguyên chia sẻ được truy cập một cách an toàn và tuần tự.

Monitor cũng hỗ trợ các biến điều kiện, cho phép một luồng chờ đợi cho đến khi một điều kiện nhất định được thỏa mãn. Điều này rất hữu ích trong các tình huống mà các luồng cần phải đợi cho đến khi một tài nguyên được giải phóng hoặc một trạng thái được đạt tới. Khi một luồng đang trong monitor gọi phương thức chờ (`wait`) của biến điều kiện, nó sẽ được tạm dừng và cho phép các luồng khác nhập monitor. Sau khi điều kiện đã đủ, luồng có thể tiếp tục bằng cách được "đánh thức" (`notify` hoặc `notifyAll`) bởi luồng khác trong monitor.

Monitor giúp đơn giản hóa quá trình lập trình đa luồng bằng cách cung cấp một khuôn khổ thống nhất cho việc đồng bộ hóa, làm giảm nguy cơ sai sót và giúp mã nguồn dễ hiểu hơn. Dưới đây là một ví dụ về cách thiết kế và sử dụng một hàng đợi an toàn cho luồng dựa trên khái niệm monitor.

```
1  #include <queue>
2  #include <iostream>
3  #include <mutex>
4  #include <thread>
5
```

```
6  class Monitor {
7  public:
8      void lock() const {
9          monitMutex.lock();
10     }
11
12     void unlock() const {
13         monitMutex.unlock();
14     }
15
16     void notify_one() const noexcept {
17         monitCond.notify_one();
18     }
19
20     template <typename Predicate>
21     void wait(Predicate pred) const {
22         std::unique_lock<std::mutex> monitLock(monitMutex);
23         monitCond.wait(monitLock, pred);
24     }
25
26 private:
27     mutable std::mutex monitMutex;
28     mutable std::condition_variable monitCond;
29 };
```

Lớp `Monitor` cung cấp các cơ chế cơ bản để đồng bộ hóa việc truy cập vào tài nguyên chia sẻ:

- `monitMutex`: Một mutex được sử dụng để kiểm soát quyền truy cập đến monitor.
- `monitCond`: Một biến điều kiện giúp luồng có thể chờ đợi cho đến khi một điều kiện nhất định được thỏa mãn.
- `lock()` và `unlock()`: Các phương thức để khóa và mở khóa mutex.
- `notify_one()`: Phương thức để thông báo cho một luồng đang chờ trên biến điều kiện rằng điều kiện có thể đã được thỏa mãn.
- `wait()`: Phương thức nhận một predicate và sử dụng biến điều kiện để chờ cho đến khi predicate trả về true.

```cpp
template <typename T>
class ThreadSafeQueue: public Monitor {
public:
    void add(T val){
        lock();
        myQueue.push(val);
        unlock();
        notify_one();
    }

    T get(){
        wait( [this] { return ! myQueue.empty(); } );
        lock();
        auto val = myQueue.front();
        myQueue.pop();
        unlock();
        return val;
    }

private:
    std::queue<T> myQueue;
};
```

Lớp `ThreadSafeQueue` là một hàng đợi an toàn cho luồng, kế thừa từ Monitor:

- `add(T val)`: Thêm một giá trị vào hàng đợi. Sau khi thêm, phương thức unlock mutex và thông báo cho các luồng khác rằng có thể có tài nguyên sẵn có.
- `get()`: Trả về và loại bỏ phần tử đầu tiên từ hàng đợi. Nếu hàng đợi rỗng, luồng sẽ chờ cho đến khi hàng đợi có phần tử. Khi có phần tử, nó sẽ lấy giá trị và sau đó mở khóa mutex.

1.4 Phương pháp đồng bộ hóa hiện đại

Trong các hệ thống đa luồng hoặc đa tiến trình, việc xử lý đồng bộ hóa đóng vai trò quan trọng để đảm bảo tính nhất quán và toàn vẹn dữ liệu. Ngoài các phương pháp truyền thống như Mutex, Semaphore, và Monitor, có nhiều phương pháp hiện đại khác có thể được áp dụng. Dưới đây là các phương pháp hiện đại cùng với mã giả minh họa cho từng phương pháp.

1.4.1 Read-Write Locks (RWLocks)

Read-Write Locks cho phép nhiều luồng đọc đồng thời nhưng chỉ một luồng có thể ghi. Điều này hữu ích trong các trường hợp mà các hoạt động đọc thường xuyên hơn ghi.

```
class RWLock:
    def __init__():
        readers = 0
        writer = False

    def acquire_read():
        while writer:
            wait()
        readers += 1

    def release_read():
        readers -= 1
        if readers == 0:
            notify_all()

    def acquire_write():
        while writer or readers > 0:
            wait()
        writer = True

    def release_write():
        writer = False
        notify_all()

// Sử dụng RWLock
rwlock = RWLock()

def reader():
    rwlock.acquire_read()
    // Đọc dữ liệu
    rwlock.release_read()

def writer():
    rwlock.acquire_write()
    // Ghi dữ liệu
    rwlock.release_write()
```

1.4.2 Lock-Free Programming

Lock-Free Programming sử dụng các kỹ thuật như Compare-And-Swap (CAS) để đảm bảo tính nhất quán dữ liệu mà không cần khóa, giúp giảm thiểu độ trễ và tăng hiệu suất.

```
class LockFreeStack:
    def __init__():
        head = None

    def push(value):
        new_node = Node(value)
        while True:
            old_head = head
            new_node.next = old_head
            if CAS(head, old_head, new_node):
                break

    def pop():
        while True:
            old_head = head
            if old_head is None:
                return None
            new_head = old_head.next
            if CAS(head, old_head, new_head):
                return old_head.value

// Sử dụng LockFreeStack
stack = LockFreeStack()
stack.push(1)
value = stack.pop()
```

1.4.3 Transactional Memory

Transactional Memory cho phép các khối mã được thực thi như các giao dịch. Nếu có xung đột xảy ra trong quá trình thực thi, giao dịch sẽ được lặp lại.

```
def transactional_block():
    // Khởi tạo giao dịch
    transaction_start()

    try:
        // Thực hiện các thao tác
        update_shared_resource()
        transaction_commit()
    except TransactionConflict:
        // Hủy giao dịch và thử lại
        transaction_abort()
        transactional_block()

```

```
// Sử dụng Transactional Memory
transactional_block()
```

1.4.4 Futures và Promises

Futures và Promises cung cấp một cách để xử lý các kết quả không đồng bộ. Futures đại diện cho một kết quả có thể sẵn sàng trong tương lai, trong khi Promises được sử dụng để đặt giá trị cho một Future.

```
class Future:
    def __init__():
        result = None
        is_done = False
        callbacks = []

    def set_result(value):
        result = value
        is_done = True
        for callback in callbacks:
            callback(value)

    def add_done_callback(callback):
        if is_done:
            callback(result)
        else:
            callbacks.append(callback)

class Promise:
    def __init__():
        future = Future()

    def set_value(value):
        future.set_result(value)

    def get_future():
        return future

// Sử dụng Futures và Promises
promise = Promise()
future = promise.get_future()

def on_result(value):
    print("Kết quả:", value)

```

```
future.add_done_callback(on_result)
promise.set_value(42)
```

1.4.5 Event Loops và Asynchronous Programming

Event Loops và lập trình không đồng bộ sử dụng các sự kiện và callback để xử lý các hoạt động không đồng bộ mà không cần dùng đến khóa.

```
class EventLoop:
    def __init__():
        events = []

    def run():
        while events:
            event = events.pop(0)
            event()

    def add_event(event):
        events.append(event)

// Sử dụng Event Loop
event_loop = EventLoop()

def async_task():
    print("Thực hiện công việc không đồng bộ")

event_loop.add_event(async_task)
event_loop.run()
```

1.4.6 Thread-Local Storage (TLS)

Thread-Local Storage cung cấp các biến mà mỗi luồng có bản sao riêng, loại bỏ sự cần thiết phải đồng bộ hóa đối với các biến này.

```
class ThreadLocal:
    def __init__():
        storage = {}

    def get_value(thread_id):
        return storage.get(thread_id, None)

    def set_value(thread_id, value):
        storage[thread_id] = value
```

```

// Sử dụng Thread-Local Storage
tls = ThreadLocal()
thread_id = get_current_thread_id()
tls.set_value(thread_id, "dữ liệu riêng của luồng")

print(tls.get_value(thread_id))
```

1.4.7 Message Passing

Message Passing sử dụng các kênh hoặc hàng đợi để các luồng hoặc tiến trình giao tiếp với nhau, giảm thiểu xung đột dữ liệu.

```
class MessageQueue:
    def __init__():
        queue = []

    def send(message):
        queue.append(message)

    def receive():
        if queue:
            return queue.pop(0)
        else:
            return None

// Sử dụng Message Passing
message_queue = MessageQueue()

def producer():
    message_queue.send("tin nhắn")

def consumer():
    message = message_queue.receive()
    if message:
        print("Nhận được:", message)

producer()
consumer()
```

1.4.8 Actor Model

Actor Model là một mô hình lập trình trong đó các "actor" là các đơn vị tính toán độc lập, mỗi actor có trạng thái riêng và giao tiếp với nhau thông qua việc gửi

và nhận tin nhắn không đồng bộ.

```
class Actor:
    def __init__():
        inbox = []

    def send(message):
        inbox.append(message)

    def process_messages():
        while inbox:
            message = inbox.pop(0)
            handle_message(message)

    def handle_message(message):
        # Xử lý tin nhắn

// Sử dụng Actor Model
actor = Actor()

def send_message_to_actor():
    actor.send("tin nhắn")

send_message_to_actor()
actor.process_messages()
```

1.4.9 Barriers

Barriers là một cơ chế đồng bộ hóa cho phép các luồng đợi cho đến khi tất cả các luồng đến điểm đồng bộ hóa. Điều này thường được sử dụng trong các thuật toán song song.

```
class Barrier:
    def __init__(n):
        count = 0
        threshold = n
        waiting_threads = []

    def wait():
        count += 1
        if count == threshold:
            for thread in waiting_threads:
                thread.resume()
        else:
```

```
            waiting_threads.append(current_thread())
            current_thread().pause()

// Sử dụng Barrier
barrier = Barrier(3)

def worker():
    // Làm việc
    barrier.wait()
    // Tiếp tục làm việc

worker()
worker()
worker()
```

1.4.10 Spinlocks

Spinlocks là loại khóa rất nhẹ, trong đó luồng sẽ "spin" (vòng lặp liên tục) cho đến khi khóa có sẵn. Spinlocks hữu ích trong các trường hợp thời gian chờ đợi rất ngắn.

```
class Spinlock:
    def __init__():
        locked = False

    def acquire():
        while CAS(locked, False, True) == False:
            pass

    def release():
        locked = False

// Sử dụng Spinlock
spinlock = Spinlock()

def critical_section():
    spinlock.acquire()
    // Thực hiện thao tác trong phần quan trọng
    spinlock.release()
```

2 Các vấn đề về tiến trình trong đoạn găng

2.1 Tương tranh

Trong hệ điều hành, tương tranh (race condition) là một vấn đề có thể xảy ra khi hai hoặc nhiều tiến trình (hoặc luồng) cùng truy cập vào một tài nguyên chia sẻ (critical resource) và ít nhất một trong số các tiến trình đó thực hiện thao tác ghi vào tài nguyên đó. Vấn đề này trở nên đặc biệt nghiêm trọng khi kết quả của các thao tác phụ thuộc vào thứ tự chúng xảy ra, điều này là không thể đoán trước được. [3]

Trương tranh có thể gây ra nhiều hậu quả nghiêm trọng, bao gồm:

- Tương tranh có thể dẫn đến việc dữ liệu bị thay đổi không nhất quán, làm hỏng dữ liệu và gây ra lỗi trong chương trình.
- Các tiến trình có thể bị kẹt khi chúng chờ đợi nhau giải phóng tài nguyên, dẫn đến deadlocks.
- Hệ thống có thể bị treo hoặc sập khi gặp phải tương tranh, đặc biệt là khi dữ liệu hệ thống bị thay đổi không kiểm soát

2.2 Bế tắc

Bế tắc (deadlock) [2] là một tình trạng trong hệ điều hành xảy ra khi một nhóm các tiến trình bị kẹt, không thể tiến hành vì mỗi tiến trình trong nhóm đang chờ đợi một tài nguyên mà tài nguyên đó đang bị tiến trình khác trong nhóm chiếm giữ. Đây là một vấn đề nghiêm trọng trong hệ điều hành và các hệ thống đa nhiệm, làm giảm hiệu suất hệ thống và có thể dẫn đến tình trạng "đóng băng" (hang) của các tiến trình.

Để deadlock xảy ra, bốn điều kiện sau đây phải đồng thời thỏa mãn:

- Mutual Exclusion (Loại trừ lẫn nhau): Tài nguyên không thể được chia sẻ; chỉ một tiến trình có thể sử dụng tài nguyên tại một thời điểm.
- Hold and Wait (Giữ và chờ): Các tiến trình đang giữ ít nhất một tài nguyên và đang chờ đợi để nhận thêm tài nguyên mà các tài nguyên này đang được tiến trình khác giữ.
- No Preemption (Không thu hồi trước): Tài nguyên không thể bị thu hồi từ tiến trình mà nó đang chiếm giữ; chỉ có tiến trình đó mới có thể giải phóng tài nguyên.
- Circular Wait (Chờ vòng tròn): Có một chuỗi các tiến trình $\{P_1, P_2, \ldots, P_n\}$ trong đó P_1 đang chờ tài nguyên mà P_2 giữ, P_2 đang chờ tài nguyên mà P_3 giữ, ..., và P_n đang chờ tài nguyên mà P_1 giữ.

  Các phương pháp xử lý bế tắc có thể kể đến:

2.2.1 Phòng ngừa bế tắc (Deadlock Prevention)

- Loại trừ lẫn nhau: Thay đổi thiết kế hệ thống để tài nguyên có thể chia sẻ, tuy nhiên điều này thường không khả thi với các tài nguyên không thể chia sẻ như máy in.
- Giữ và chờ: Yêu cầu tiến trình phải yêu cầu tất cả tài nguyên mà nó cần tại thời điểm bắt đầu. Nếu không được cấp phát tất cả tài nguyên, nó phải giải phóng tất cả các tài nguyên hiện có và chờ đến khi tất cả tài nguyên có sẵn.
- Không thu hồi trước: Cho phép hệ thống thu hồi tài nguyên từ tiến trình đang giữ chúng khi tiến trình đó đang chờ tài nguyên khác.
- Chờ vòng tròn: Áp dụng thứ tự ưu tiên cho các tài nguyên và yêu cầu tiến trình phải yêu cầu tài nguyên theo thứ tự ưu tiên đó.

2.2.2 Phòng tránh bế tắc (Deadlock Avoidance)

- Thuật toán Banker: Được sử dụng trong việc cấp phát tài nguyên để đảm bảo rằng hệ thống luôn ở trạng thái an toàn. Thuật toán này yêu cầu mỗi tiến trình phải khai báo trước số lượng tài nguyên tối đa mà nó cần. Hệ thống sẽ kiểm tra và chỉ cấp phát tài nguyên nếu việc cấp phát không dẫn đến trạng thái không an toàn.

2.2.3 Nhận biết và khắc phục (Deadlock Detection and Recovery)

- Nhận biết: Hệ thống sẽ kiểm tra định kỳ các tiến trình và trạng thái của các tài nguyên để phát hiện deadlock. Một đồ thị chờ tài nguyên (wait-for graph) có thể được sử dụng để kiểm tra vòng tròn trong hệ thống, từ đó xác định deadlock.
- Khắc phục: Sau khi phát hiện deadlock, hệ thống sẽ khôi phục bằng cách hủy bỏ (terminate) một hoặc nhiều tiến trình để phá vỡ deadlock. Có thể chọn tiến trình nào tiêu tốn ít tài nguyên nhất, hoặc tiến trình nào đã tiến hành lâu nhất để hủy bỏ.

2.2.4 Phớt lờ bế tác

- Đây là phương pháp mà một số hệ điều hành, như Unix, thường áp dụng, vì deadlock xảy ra không thường xuyên nên họ chọn cách không giải quyết trực tiếp mà để người dùng xử lý khi gặp phải.

2.3 Chết đói

Chết đói (starvation) là một vấn đề trong hệ điều hành và các hệ thống đa nhiệm, xảy ra khi một tiến trình không bao giờ nhận được tài nguyên cần thiết để tiếp tục thực hiện, vì các tài nguyên này liên tục được cấp phát cho các tiến trình khác. Starvation là một hiện tượng phổ biến trong các hệ thống cạnh tranh

tài nguyên và có thể gây ra các vấn đề nghiêm trọng về hiệu suất và tính công bằng.

Starvation có thể xảy ra do nhiều nguyên nhân khác nhau, bao gồm:

- Ưu tiên (Priority): Trong hệ thống lập lịch ưu tiên, các tiến trình có ưu tiên cao luôn được cấp phát tài nguyên trước, dẫn đến các tiến trình có ưu tiên thấp bị bỏ qua và không nhận được tài nguyên.
- Chính sách lập lịch (Scheduling Policies): Một số chính sách lập lịch như First-Come-First-Serve (FCFS) hoặc Shortest Job Next (SJN) có thể gây ra starvation cho các tiến trình lớn hoặc tiến trình đến sau.
- Đồng bộ hóa (Synchronization): Khi sử dụng các cơ chế đồng bộ hóa như mutexes hoặc semaphores, một tiến trình có thể bị kẹt vô thời hạn nếu không có cơ hội nhận được khóa hoặc tín hiệu cần thiết.
- Phân phối tài nguyên (Resource Allocation): Cách tài nguyên được phân phối và quản lý trong hệ thống có thể dẫn đến việc một số tiến trình không bao giờ nhận được tài nguyên, trong khi các tiến trình khác liên tục nhận được.

Có nhiều phương pháp khác nhau để giải quyết và ngăn ngừa starvation. Dưới đây là một số phương pháp phổ biến:

2.3.1 Điều chỉnh ưu tiên (Priority Adjustment)

- Tăng dần ưu tiên (Priority Aging): Một trong những phương pháp phổ biến nhất để ngăn ngừa starvation là tăng dần ưu tiên của các tiến trình khi chúng chờ đợi tài nguyên. Theo thời gian, ưu tiên của tiến trình chờ đợi sẽ tăng lên, và cuối cùng tiến trình đó sẽ nhận được tài nguyên.
- Phân phối ưu tiên công bằng: Sử dụng các chính sách lập lịch đảm bảo rằng tất cả các tiến trình đều nhận được tài nguyên một cách công bằng, tránh tình trạng ưu tiên quá cao cho một số tiến trình.

2.3.2 Chính sách lập lịch công bằng (Fair Scheduling Policies)

- Round-Robin Scheduling: Một trong những phương pháp lập lịch đơn giản và công bằng là Round-Robin, trong đó mỗi tiến trình nhận được một khoảng thời gian nhất định để thực thi. Điều này giúp đảm bảo rằng không có tiến trình nào bị bỏ qua vô thời hạn.
- Fair Share Scheduling: Phân chia tài nguyên dựa trên các nhóm người dùng hoặc nhóm tiến trình, đảm bảo rằng mỗi nhóm nhận được một phần tài nguyên công bằng.

2.4 Khóa sống

Khóa sống (livelock) là một tình trạng trong đó hai hoặc nhiều tiến trình trong hệ điều hành hoặc hệ thống phần mềm đang hoạt động nhưng không tiến triển được trong công việc thực tế. Tình trạng này tương tự như khóa chết (deadlock, có thể tạm dịch: bế tắc) nhưng với một điểm khác biệt quan trọng: các tiến trình trong livelock không bị chặn lại hoàn toàn mà vẫn tiếp tục thay đổi trạng thái hoặc thực hiện các hành động nhưng không đạt được tiến bộ nào.

Một ví dụ đơn giản để hiểu về điều này, hãy tưởng tượng hai người đang cố gắng tránh nhau trong một hành lang hẹp. Khi một người di chuyển sang bên trái, người kia cũng di chuyển sang bên trái để tránh, và khi một người di chuyển sang bên phải, người kia cũng làm tương tự. Kết quả là cả hai người đều di chuyển liên tục nhưng không ai tiến lên được.

Livelock thường xuất hiện do các nguyên nhân sau:

- Giao thức phản ứng tự động: Khi các tiến trình hoặc hệ thống có cơ chế tự động điều chỉnh để tránh xung đột nhưng điều chỉnh này dẫn đến tình trạng không tiến bộ.
- Thiếu phối hợp giữa các tiến trình: Khi các tiến trình không được phối hợp một cách hiệu quả và dẫn đến việc chúng liên tục điều chỉnh nhưng không đạt được tiến bộ.
- Thiết kế hệ thống không hợp lý: Khi thiết kế hệ thống không đảm bảo việc các tiến trình có thể tiến triển mà không bị rơi vào vòng lặp vô tận của việc điều chỉnh.

Giả sử có hai tiến trình A và B cố gắng nắm giữ một tài nguyên chung. Cả hai tiến trình đều có cơ chế kiểm tra và nhường quyền nếu tài nguyên đã bị chiếm. Nếu cả hai tiến trình liên tục nhường quyền cho nhau, chúng sẽ rơi vào trạng thái livelock.

```
Process A:
while true:
    if resource is available:
        acquire resource
        do work
        release resource
    else:
        release resource

Process B:
while true:
    if resource is available:
        acquire resource
```

```
14          do work
15          release resource
16      else:
17          release resource
18
```

2.4.1 Phát hiện

Phát hiện livelock có thể khó khăn vì các tiến trình vẫn đang hoạt động và không bị chặn hoàn toàn. Tuy nhiên, có thể sử dụng các phương pháp sau để phát hiện:

- Giám sát trạng thái tiến trình: Theo dõi trạng thái và hành động của các tiến trình. Nếu thấy rằng các tiến trình liên tục thay đổi trạng thái mà không hoàn thành công việc, có thể đang xảy ra livelock.
- Phân tích hiệu suất hệ thống: Nếu hệ thống có hiệu suất giảm sút mặc dù các tiến trình vẫn hoạt động, cần kiểm tra xem có phải livelock đang xảy ra không.

2.4.2 Giải quyết

Có một số phương pháp để giải quyết livelock:

- Thiết kế lại giao thức hoặc thuật toán: Đảm bảo rằng các tiến trình có cơ chế phối hợp hợp lý để tránh rơi vào trạng thái livelock.
- Thêm cơ chế trì hoãn ngẫu nhiên: Khi các tiến trình gặp xung đột, thay vì ngay lập tức nhường quyền, chúng có thể đợi một khoảng thời gian ngẫu nhiên trước khi thử lại.
- Giới hạn số lần thử lại: Giới hạn số lần các tiến trình có thể thử lại trước khi thực hiện các hành động thay thế hoặc thông báo lỗi.

3 Các bài toán về điều độ tiến trình và luồng qua đoạn găng

3.1 Bài toán Người hút thuốc lá

Bài toán Người hút thuốc lá (The Cigarette Smokers Problem) được Suhas Patil đề xuất lần đầu, ông khẳng định vấn đề này không thể giải quyết được chỉ bằng `semaphores` trong một số điều kiện nhất định. [1]

3.1.1 Mô tả bài toán

Có 4 luồng tham gia: 1 đại lý và 3 người hút thuốc. Người hút thuốc liên tục lặp lại việc chờ nguyên liệu, sau đó làm và hút xì gà. Nguyên liệu bao gồm thuốc lá, giấy và diêm.

Giả sử đại lý có nguồn cung vô hạn 3 loại nguyên liệu này, và mỗi người hút thuốc có vô hạn 1 trong 3 nguyên liệu; ví dụ, 1 người có diêm, 1 người khác có giấy, và người thứ 3 có thuốc lá.

Đại lý lặp đi lặp lại việc chọn 2 loại nguyên liệu khác nhau 1 cách ngẫu nhiên và cung cấp cho người hút thuốc. Dựa vào những loại nguyên liệu nào được chọn, người hút thuốc có loại nguyên liệu tương ứng còn thiếu sẽ lấy 2 loại nguyên liệu được đại lý cung cấp đấy để làm và hút xì gà. Ví dụ, nếu đại lý lấy ra thuốc lá và giấy, người hút thuốc có diêm sẽ lấy cả 2 loại nguyên liệu được cung cấp đấy, tạo ra xì gà và báo lại cho đại lý.

Để giải thích cho giả thuyết, đại lý tượng trưng cho một hệ điều hành cấp phát các tài nguyên, người hút thuốc tượng trưng cho các ứng dụng cần tài nguyên. Vấn đề là đảm bảo nếu tài nguyên có sẵn cho phép một ứng dụng khác nữa tiếp tục, các ứng dụng đó có nên được đánh thức. Ngược lại, chúng ta muốn tránh đánh thức một ứng dụng nếu nó không thể tiến hành.

Dựa trên giả thuyết này, có 3 phiên bản của vấn đề này thường xuyên xuất hiện trong các cuốn sách.

- *Phiên bản không khả thi:* Phiên bản của Patil đặt ra những hạn chế cho giải pháp. Đầu tiên, bạn không được phép sửa đổi code của đại lý. Nếu đại lý đại diện cho hệ điều hành, điều này có nghĩa là bạn không muốn sửa đổi nó mỗi khi có một ứng dụng mới xuất hiện. Hạn chế thứ hai là bạn không được sử dụng các câu lệnh điều kiện hoặc một mảng các `semaphore`. Với những ràng buộc này, vấn đề không thể được giải quyết, nhưng Parnas chỉ ra ràng buộc thứ 2 khá nhân tạo. Với những ràng buộc như vậy, rất nhiều bài toán trở thành không thể giải được.

- *Phiên bản thú vị:* Phiên bản này giữ lại hạn chế đầu tiên - bạn không thể thay đổi mã của đại lý - nhưng bỏ qua những hạn chế khác.

- *Phiên bản đơn giản:*: Trong một số sách, vấn đề quy định rằng đại lý nên tín hiệu cho người hút thuốc tiếp theo nên đi tiếp, tùy thuộc vào các thành phần có sẵn. Phiên bản này của vấn đề không thú vị vì nó làm cho toàn bộ giả thuyết, các nguyên liệu và xì gà không liên quan đến nhau. Ngoài ra, ở những vấn dề thực tế, có lẽ không phải là ý kiến hay khi yêu cầu đại lý phải biết về các luồng khác và những gì họ đang chờ đợi. Cuối cùng, phiên bản này của vấn đề quá dễ dàng.

3.1.2 Phương pháp giải quyết

Dương nhiên, chúng ta sẽ tập trung vào phiên bản thú vị. Để hoàn thành bằng mô tả, chúng ta cần chỉ định mã của đại lý. Đại lý sử dụng các `semaphore` sau:

```
std::counting_semaphore<INT_MAX> agentSem{1};
std::counting_semaphore<INT_MAX> tobacco{0};
std::counting_semaphore<INT_MAX> paper{0};
std::counting_semaphore<INT_MAX> match{0};
```

Đại lý thực sự được tạo ra từ 3 luồng đồng thời, đại lý A, đại lý B và đại lý C. Mỗi đại lý chờ đợi ở **agentSem**; mỗi lần **agentSem** được ra hiệu, một trong các đại lý thức dậy và cung cấp nguyên liệu bằng cách ra hiệu cho 2 `semaphore`.

```
// Đại lý A
agentSem.acquire();
tobacco.release();
paper.release();
```

```
// Đại lý B
agentSem.acquire();
paper.release();
match.release();
```

```
// Đại lý C
agentSem.acquire();
tobacco.release();
match.release();
```

Bài toán này khó vì lời giải tự nhiên không hoạt động. Nó sẽ cố thử để viết 1 lời giải như sau:

```
// Người hút thuốc có diêm
tobacco.acquire();
paper.acquire();
agentSem.release();
```

```
// Người hút thuốc có thuốc lá
paper.acquire();
match.acquire();
agentSem.release();
```

```
// Người hút thuốc có giấy
tobacco.acquire();
match.acquire();
agentSem.release();
```

Tuy nhiên lời giải này là sai! Bài toán với lời giải này có khả năng gây ra `deadlock`. Tưởng tượng khi đại lý lấy ra thuốc lá và giấy. Do người hút thuốc có diêm đang đợi ở `tobacco`, nó sẽ được /textttunblocked. Nhưng người thút thuốc có thuốc lá đang đợi ở `paper`, do đó nó cũng có khả năng được unblock. Khi đó người hút thuốc có diêm sẽ block ở `paper` và người thút thuốc có thuốc lá sẽ `block` ở `match`. `Deadlock` xảy ra!

Một lời giải khác được đưa ra bởi Parnas sử dụng 3 luồng hỗ trợ gọi là các "pusher", chúng sẽ phản hồi sự ra hiệu từ đại lý, theo dõi những nguyên liệu được cung cấp, và ra hiệu cho người hút thuốc tương ứng.

Những biến bổ trợ và `semaphore` như sau.

```
bool isTobacco false;
bool isPaper = false;
bool isMatch = false;
std::counting_semaphore<INT_MAX> tobaccoSem{0};
std::counting_semaphore<INT_MAX> paperSem{0};
std::counting_semaphore<INT_MAX> matchSem{0};
std::mutex mutex;
```

Các biến kiểu bool chỉ ra rằng hiện tại nguyên liệu đó có đang sẵn sàng hay không. Các "pusher" sẽ sử dụng `tobaccoSem` để ra hiệu cho người hút thuốc có thuốc lá, và các `semaphore` khác cũng giống như vậy.

Sau đây là code của 1 trong các "pusher":

```
// Pusher A
tobacco.acquire();
mutex.lock();
if (isPaper) {
    isPaper = false;
    matchSem.release();
}
else if (isMatch) {
    isMatch = false;
    paperSem.release();
}
else {
```

```
13      isTocobacco = true;
14  }
15  mutex.unlock();
```

Một "pusher" được đánh thức bất kì khi nào có thuốc lá sẵn sàng. Nếu nó tìm ra `isPaper = true`, nó biết rằng "pusher" B đã thực hiện, do đó nó có thể ra hiệu cho người hút thuốc có diêm. Tương tự nếu nó tìm thấy diêm ở trên bàn, nó có thể ra hiệu cho người hút thuốc có giấy.

Nhưng nếu "pusher" A chạy đầu tiên, nó sẽ tìm ra cả `isPaper` và `isMatch` đều đang bằng `false`. Nó không thể ra hiệu cho bất kì người hút thuốc nào, nên sẽ gán `isTobacco = true`.

Các "pusher" khác cũng tương tự. Bởi vì "pushers" đã làm tất cả công việc thực sự, nên code của người hút thuốc trở nên đơn giản:

```
1  // Người hút thuốc có thuốc lá
2  tobaccoSem.acquire();
3  makeCigarette();
4  agentSem.release();
5  smoke();
```

Parnas biểu diễn một lời giải tương tự bằng cách tổng hợp các biến kiểu bool, `bitwise`, thành một số nguyên, sau đó sử dụng số nguyên đó như một chỉ số trong một mảng các `semaphore`. Như vậy, có thể tránh được sử dụng các câu lệnh điều kiện (một trong những hạn chế nhân tạo). Code kết quả ngắn gọn hơn một chút, nhưng chức năng của nó không rõ ràng.

3.1.3 Bài toán tổng quát

Parnas cho rằng bài toán Cigarret smokers sẽ trở nên khó hơn nếu chúng ta sửa code của đại lý, loại đi yêu cầu rằng đại lý sẽ đợi sau khi lấy ra các nguyên liệu. Trong trường hợp này, sẽ có nhiều hơn 1 cái của từng nguyên liệu đang sẵn sàng.

Nếu đại lý không đợi những người hút thuốc, các loại nguyên liệu sẽ được tích lũy lại. Thay vì sử dụng kiểu dữ liệu `bool` để theo dõi các loại nguyên liệu, chúng ta cần biến `int` để đếm chúng.

```
1  // Gợi ý cho bài toán Cigarret smokers tổng quát
2  int numTobacco = 0;
3  int numPaper = 0;
4  int numMatch = 0;
```

Sau đây là code được chỉnh sửa lại của "pusher" A:

```
// Pusher A
tobacco.acquire();
mutex.lock();
if (numPaper) {
    numPaper -= 1;
    matchSem.release();
}
else if (numMatch) {
    numMatch -= 1;
    paperSem.release();
}
else {
    numTobacco += 1;
}
mutex.unlock();
```

Một cách để hình dung bài toán này là tưởng tượng rằng khi đại lý thực hiện, nó sẽ tạo ra 2 "pusher", cho mỗi "pusher" một nguyên liệu, và đặt chúng trong 1 căn phòng với tất cả các "pusher" khác. Bởi vì có khóa trong (`mutex`), các "pusher" lần lượt đi vào 1 căn phòng nơi mà có 3 người hút thuốc đang ngủ và 1 cái bàn. Tại 1 thời điểm, mỗi "pusher" vào phòng và kiểm tra các nguyên liệu ở trên bàn. Nếu "pusher" có thể tổng hợp thành công tập hợp các nguyên liệu, "pusher" sẽ lấy chúng ra khỏi bàn và đánh thức người hút thuốc thương ứng. Nếu không, "pusher" để lại nguyên liệu mà nó có ở trên bàn và rời đi mà không đánh thức ai.

Đây là một ví dụ về một mô hình chúng ta sẽ thấy nhiều lần, có thể gọi nó là **scoreboard**. Các biến `numPaper, numTobacco` và `numMatch` theo dõi trạng thái của hệ thống. Khi mỗi thread đi qua khóa trong (`mutex`), nó kiểm tra trạng thái như đang nhìn vào bản điểm và phản ứng tương ứng.

3.2 Bài toán Quán rượu Sushi

3.2.1 Mô tả bài toán

Bài toán Quán rượu Sushi (The Sushi Bar Problem) được đề xuất bởi *Kenneth Reek*. Giả sử, một quán rượu sushi có 5 ghế ngồi. Khi bạn vào quán rượu, nếu có ít nhất một ghế trống, bạn có thể ngồi vào chỗ đó. Nếu cả 5 ghế đều đầy, tức là tất cả mọi người đang ăn tối cùng nhau, và bạn sẽ phải chờ tất cả mọi người ăn xong, rời đi trước khi bạn ngồi vào ghế. [1]

3.2.2 Phương pháp giải quyết

```
eating = waiting = 0
mutex = Semaphore(1)
```

```
3  block = Semaphore(0)
4  must_wait = False
```

Trong đó, `eating` và `waiting` chỉ số lượng người đang ngồi và đang chờ, `mutex` đảm bảo cho số lượng khách hàng theo yêu cầu, `must_wait` báo hiệu quán có đang đầy, khách hàng nào đó đến sẽ bị `block`.

Reek sử dụng một phương pháp giải chưa đúng dưới đây để chỉ ra một trong những vấn đề của bài toán của bài toán.

```
1  mutex.wait()
2  if must_wait:
3      waiting +=1;
4      mutex.signal();
5      block.wait();
6
7      mutex.wait(); // yêu cầu lại mutex
8      waiting -=1;
9
10 eating +=1;
11 must_wait = (eating == 5);
12 mutex.signal();
13
14 // ăn sushi
15
16 mutex.wait();
17 eating -=1;
18 if eating ==0:
19     n = min(5, waiting);
20     block.signal(n);
21     must_wait = False;
22 mutex.signal();
```

Vấn đề nằm ở dòng 7 trong phương pháp trên. Nếu một vị khách đến trong khi quán đầy, anh ta phải bỏ qua `mutex` và chờ đến khi các vị khách khác rời đi. Khi mà vị khách cuối cùng rời đi, họ ra tín hiệu `block`, đánh thức những vị khách đang chờ và khởi tạo lại biến `must_wait`.

Nhưng khi vị khách đó thức dậy, họ phải lấy lại `mutex`, điều này đồng nghĩa với việc họ phải cạnh tranh với các vị khách mới đến. Nếu các vị khách mới đến lấy được `mutex` trước, họ có thể ngồi vào các ghế. Vấn đề xảy ra: Nếu có nhiều hơn 5 vị khách mới đến thì vị khách đến ban đầu vẫn phải chờ do vậy vi phạm điều kiện của điều độ tiến trình.

Reek đề xuất hai giải pháp cho vấn đề này.

Thuật toán 1. Lý do duy nhất mà vị khách đang chờ cập nhật lại `mutex` là cập nhật lại trạng thái của `eating` và `waiting`, bởi vậy một cách để giải quyết vấn đề này là vị khách có `mutex` cập nhật.

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

// ăn sushi

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

Khi vị khách ban đầu đưa ra `mutex`, `eating` đã hoàn toàn được cập nhật, bởi vậy vị khách mới đến sẽ xem trạng thái phù hợp và có thể block nếu cần thiết. Reek gọi phương pháp này "Tôi sẽ làm điều đó cho bạn". Về mặt logic, luồng đang thực hiện công việc có vẻ như thuộc về các luồng đang chờ. Một hạn chế của phương pháp này là khó xác nhận rằng trạng thái đang được cập nhật chính xác.

Thuật toán 2. Giải pháp thay thế của Reek dựa trên việc có thể chuyển một `mutex` từ luồng này sang luồng khác. Nói cách khác, một luồng có thể có được một khóa và sau đó một luồng khác có thể giải phóng nó. Miễn là cả luồng hiểu rằng khóa đã được chuyển, không có gì sai với điều này.

```
mutex.wait()
if must_wait:
    waiting +=1
    mutex.signal()
    block.wait() // khi tiếp tục, có mutex
    waitng -=1
```

```
7
8   eating +=1
9   mutex_wait == (eating == 5)
10  if wating and not must_wait:
11      block.signal() // bỏ qua mutex
12  else:
13      mutex.wait()
14
15  // ăn sushi
16
17  mutex.wait()
18  eating -=1
19  if eating == 0: must_wait = False
20  if waiting and not must_wait:
21      block.signal() // bỏ qua mutex
22  else:
23      mutex.signal()
```

Nếu có ít hơn 5 vị khách vào quán, không ai phải chờ, một vị khách mới đến tăng `eating` và ra `mutex`. Vị khách thứ 5 thiết lập `must_wait`.

Nếu `must_wait` được thiết lập, một vị khách mới đến sẽ bị block đến tận khi vị khách cuối cùng trong quán thiết lập lại `must_wait` và đánh thức block. Điều này có nghĩ là, luồng báo hiệu từ bỏ `mutex` và luông chờ đợi nhận `mutex`.

Khi một luồng đang chờ đợi được khởi động lại, nghĩ là nó sở hữu `mutex`. Nếu có các luồng khách đang chờ, nó ra tín hiệu block, chuyển `mutex` cho một luồng đang chờ. Quá trình này tiếp tục, mỗi luồng chuyển `mutex` cho luồng kế tiếp, đến tận khi các ghế đã đầy hoặc không còn luồng chờ. Trong cả hai trường hợp, luồng cuối cùng ;uôn giải phóng `mutex` và ngồi vào ghế.

Reek gọi phương pháp này là "Truyền gậy", bởi vì `mutex` được chuyển từ luồng này sang luồng khác giống như chuyền gậy trong chạy tiếp sức. Ưu điểm của phương pháp này là dễ dàng xác định được trạng thái của `eating` và `waiting`. Một hạn chế là khó xác định `mutex` được sử dụng chính xác.

3.3 Bài toán Chờ xe buýt

3.3.1 Mô tả bài toán

Bài toán này ban đầu dựa trên vấn đề xe buýt Senate ở Wellesley College. Người đi đến một trạm xe buýt và chờ xe buýt. Khi xe buýt tới, tất cả người đang chờ sẽ lên xe, nhưng bất cứ ai đến trong khoảng thời gian này đều phải chờ xe buýt tiếp theo. Sức chứa của xe buýt khoảng 50 người, nếu có hơn 50 người chờ, một vài người sẽ phải chờ cho chuyến xe tiếp theo. Khi tất cả những người đang chờ đã lên

xe, xe buýt có thể khởi hành. Nếu xe buýt đến khi không có người chờ nó đi ngay lập tức. [1]

3.3.2 Phương pháp giải quyết

```
// Khởi tạo
riders = 0
mutex = Semaphore(1)
multiplex = Semaphore(50)
bus = Semaphore (0)
allAboard = Semaphore (0)
```

Trong đó, `mutex` thể hiện cho `riders`, chỉ số lượng người đang chờ xe; `multiplex` đảm bảo không có quá 50 người trên xe.

Người chờ xe buýt được báo hiệu khi xe đến. Xe buýt chờ hành khách lên xe `allAboard`, được báo hiệu bởi hành khách cuối cùng lên xe.

Thuật toán 1. Bài toán có hai đối tượng cần đồng bộ đó là người đi và xe buýt. Dưới đây là đoạn mã giả cho xe buýt, sử dụng phương pháp "Truyền gậy".

```
mutex.wait()
if riders > 0:
    bus.signal() // giải phóng mutex
    allAboard.wait() // nhận lại mutex
mutex.signal()

depart()
```

Khi xe buýt đến, nó lấy `mutex`, điều này ngăn cản những người đến muộn lên xe. Nếu không có người chờ, xe buýt khởi hành ngay lập tức. Mặt khác, nó báo hiệu xe buýt tới và chờ mọi người lên xe.

Dưới đây là đoạn mã giả cho người đi.

```
multiplex.wait()
    mutex.wait()
        riders +=1
    mutex.signal()

    bus.wait() //lấy mutex
multiplex.signal()

boardBus()
```

```
10
11  riders -=1
12  if riders == 0:
13      allAboard.signal()
14  else:
15      bus.signal() //giải phóng mutex
```

Biến `mutiplex` kiểm soát số lượng người trong khu vực chờ, chính xác là một người không thể vào khu vực chờ đến tận khi họ tăng `riders`.

Người đi chờ đến tận khi xe buýt tới. Khi người đi được "đánh thức", họ có `mutex`. Sau khi lên xe, mỗi người giảm `riders`. Nếu có nhiều người đang chờ, người đang lên xe ra hiệu co xe buýt và trao lại `mutex` cho người tiếp theo. Người lên cuối cùng báo hiệu `allAboard` và trao lại `mutex` cho xe buýt.

Cuối cùng, xe buýt giải phóng `mutex` và khởi hành.

Thuật toán 2. Grant Hutchins đề cập tới phương pháp dưới đây, sử dụng ít biến hơn so với phương pháp trên và không liên quan đến việc trao lại đèn báo.

```
1  mutex.wait()
2  waiting = 0
3  mutex = new Semaphore (1)
4  bus = new Semaphore (0)
5  boarded = new Semaphore (0)
```

Trong đó, `waiting` là số lượng người trong khu vực lên xe, được báo hiệu bởi `mutex`, xe buýt báo hiệu khi đến; `boarded` báo hiệu rằng một người đã lên xe.

Dưới đây là đoạn mã giả cho xe buýt:

```
1  mutex.wait()
2  n = min(waiting, 50)
3  for i in range (n):
4      bus.signal()
5      boarded.wait()
6
7  wating = max(waiting-50, 0)
8  mutex.signal()
9
10 depart()
```

Xe buýt nhận `mutex` và giữ nó trong suốt quá trình lên xe. Vòng lặp báo hiệu cho mỗi người và chờ họ lên xe. Bằng việc kiểm soát số lượng báo hiệu, xe buýt

ngăn không cho quá 50 người lên xe.

Khi mà tất cả hành khách lên xe, xe buýt cập nhật lại `waiting`, "Tôi sẽ làm điều đó cho bạn". Dưới đây là đoạn mã giả cho hành khách sử dụng một đèn báo và một địa điểm.

```
1 mutex.wait()
2     waiting +=1
3 mutex.signal()
4
5 bus.wait()
6 board()
7 boarded.signal()
```

"Nếu hành khách đến trong khi xe buýt đang có người lên, họ có thể tức giận khi phải chờ đến chuyến tiếp theo."

3.4 Bài toán Người nhập cư

3.4.1 Mô tả bài toán

Bài toán này được viết bởi Grant Hutchins. [1]

Có 3 loại luồng: người nhập cư, người dân và một thẩm phán. Người nhập cư phải xếp hàng đợi, làm thủ tục và sau đó được ngồi xuống.

Tại một thời điểm nào đó, thẩm phán bước vào tòa nhà. Khi thẩm phán đang ở trong tòa nhà, không ai được vào và người nhập cư không được rời đi. Người dân có thể rời đi.

Khi người nhập cư làm thủ tục, thẩm phán có thể xác nhận việc nhập tịch. Sau khi được xác nhận người nhập cư được nhận giấy quốc tịch Hoa Kỳ. Vào thời điểm nào đó, thẩm phán rời đi sau khi xác nhận. Bây giờ, người dân có thể vào như trước. Sau khi người nhập cư nhận giấy chứng nhận, họ có thể rời đi.

Để cụ thể hóa bài toán, tác giả đưa ra một số ràng buộc với từng đối tượng:

- Người nhập cư phải gọi các hàm `enter`, `checkIn`, `sitDown`, `swear`, `getCertificate`, và `leave`.
- Thẩm phán gọi hàm `enter`, `confirm`, và `leave`.
- Người dân gọi các hàm `enter`, `spectate` và `leave`.
- Khi mà thẩm phán đang ở trong tòa nhà, không ai có thể `enter` và người nhập cư không thể `leave`.

- Người thẩm phán không thể `confirm` đến tận khi tất cả người nhập cư `enter` và `checkIn`.
- Người nhập cư không thể `getCertificate` đến tận khi thẩm phán tiến hành `confirm`.

3.4.2 Phương pháp giải quyết

```
// Khởi tạo
noJudge = Semaphore (1)
entered = 0
checked = 0
mutex = Semaphore (1)
confirmed = Semaphore (0)
```

Trong đó, `noJudge` cho phép người nhập cư và người dân đến; nó cũng liên quan đến `entered`, đếm số lượng người nhập cư đang ở trong phòng; `checked` chỉ số lượng người nhập cư đã làm thủ tục, được bảo vệ bởi `mutex`; `confirmed` thông báo thẩm phán tiến hành `confirm`.

Đoạn mã giả cho người nhập cư:

```
noJudge.wait()
enter()
entered ++
noJudge.signal()

mutex.wait()
checkIn()
checked++

if judge = 1 and entered == checked:
    allSignedIn.signal()
else:
    mutex.signal()

sitDown()
confirmed.wait()

swear()
getCertificate()

noJudge.wait()
```

```
22  leave()
23  noJudge.signal()
```

Người nhập cư khi vào đi qua một cái cửa; trong khi thẩm phán đang ở trong phòng cánh của bị khóa lại.

Sau khi vào, người nhập cư phải lấy được `mutex` để làm thủ tục và cập nhật `checked`. Nếu có một thẩm phán đang chờ, người nhập cư cuối cùng sau khi làm thủ tục báo hiệu `allSignedIn` và chuyển `mutex` cho thẩm phán.

Đoạn mã giả cho thẩm phán:

```
1   noJudge.wait()
2   mutex.wait()
3
4   enter()
5   judge = 1
6
7   if entered > checked:
8       mutex.signal()
9       allSignedIn.wait() //Lấy lại mutex
10
11  confirm()
12  confirmed.signal(checked)
13  entered = checked = 0
14
15  leave()
16  judge = 0
17
18  mutex.signal()
19  noJudge.signal()
```

Thẩm phán giữ `noJudge` để ngăn cản người nhập cư và người dân đi vào, ông ấy cũng giữ `mutex` bởi vậy được quyền truy cập vào `entered` và `checked`.

Khi mà thẩm phán đến mà người nhập cư đã vào và làm thủ tục xong, ông ấy sẽ tiến hành xác nhận ngay lập tức. Trái lại, thẩm phán phải giải phóng `mutex` và chờ. Khi mà người nhập cư cuối cùng xác nhận thut tục và ra báo hiệu `allSignedIn`, lúc này người thẩm phán lấy lại `mutex`.

Sau khi gọi hàm `confirm`, thẩm phán báo hiệu `confirmed` cho mỗi người nhập cư đã hoàn thành thủ tục, và sau đó cập nhập lại số lượng. Sau đó, thẩm phán giải phóng `mutex` và `noJudge`.

Sau khi thẩm phán báo hiệ `confirmed`, người nhập cư gọi hàm `swear` và `getCertificate`, và sau đó chở cửa `noJudge` mở trước khi rời đi.

Đối với người dân chỉ có ràng buộc duy nhất, họ phải tuân theo cửa `noJudge`.

```
noJudge.wait()
enter()
noJudge.signal()

spectate()

leave()
```

Chú ý, trong giải pháp trên, người nhập cư có thể gặp khó khăn, sau khi họ nhận được giấy chứng nhận một vị thẩm phán khác lại đến xác nhận cho các người nhập cư khác. Nếu điều này xảy ra, họ sẽ tiếp tục phải chờ trước khi rời đi.

Giải pháp đưa ra là, cải thiện thuật toán trên bằng việc thêm ràng buộc, sau khi thẩm phán rời đi, tất cả người nhập cư đã được xác nhận phải được rời đi trước khi thẩm phán quay trở lại.

3.4.3 Bài toán mở rộng

Sử dụng thêm một vài biến: sử dụng hai đèn báo.

```
exit = Semaphore (0)
allGone = Semaphore (0)
```

Đối với người nhập cư: Người nhập cư chờ cho đến khi thẩm phán rời đi.

```
noJudge.wait()
enter()
entered ++
noJudge.signal()

mutex.wait()
checkIn()
checked++

if judge = 1 and entered == checked:
    allSignedIn.signal()
else:
    mutex.signal()

```

```
sitDown()
confirmed.wait()

swear()
getCertificate()

exit.wait()  // lấy mutex
leave()
checked--
if checked == 0:
    allGone.signal() //trao lại mutex
else:
    exit.signal() // trao lại mutex
```

Đối với thẩm phán: Khi thẩm phản chuẩn bị rời đi, ông ấy sẽ không thể giải phóng `noJudge`, bởi vì điều này có thể cho phép nhiều người nhập cư hoặc một thẩm phán vào. Thay vì vậy, thẩm phán sẽ báo hiệu `exit`, cho phép một người nhập cư rời đi và trao lại `mutex`.

Người nhập cư nhận báo hiệu giảm `checked` và trao lại `mutex` cho người nhập cư tiếp theo. Người nhập cư cuối cùng rời đi sẽ ra báo hiệu `allGone` và trao lại `mutex` cho thẩm phán. Việc trao lại không thực sự cần thiết, nhưng thẩm phán có thể giải phóng `mutex` và `noJudge` cùng lúc, trước khi rời đi.

```
noJudge.wait()
mutex.wait()

enter()
judge = 1

if entered > checked:
    mutex.signal()
    allSignedIn.wait() // lấy lại mutex

confirm()
confirmed.signal(checked)
entered = checked = 0

leave()
judge = 0

mutex.signal() // trao lại mutex
allGone.wait() // lấy lại mutex
mutex.signal()
```

```
21  noJudge.signal()
```

Đoạn mã giả của người dân không thay đổi.

3.5 Bài toán Bữa tối của tộc người nguyên thủy

3.5.1 Mô tả bài toán

Một bộ tộc người nguyên thủy ăn tối chung với nhau. Đồ ăn ở trong 1 cái mâm lớn có thể chứa được M phần đồ ăn, mỗi phần cho 1 người. Khi một người muốn ăn, hắn tự mình lấy đồ ăn từ nồi nếu thức ăn vẫn chưa hết. Còn nếu đồ ăn đã hết,người đó gọi người nấu nướng và đợi đến khi mâm chứa đầy đồ ăn. [1]

3.5.2 Phương pháp giải quyết

Đoạn code dưới đây mô tả hoạt động của một người ăn bất kỳ:

```
1  // Khởi tạo
2  while True
3      getServingFromPot()
4      eat()
```

Và một người nấu ăn có hoạt động:

```
1  while True
2      putServingInPot(M)
```

Bài toán có các ràng buộc đồng bộ sau:

- Người ăn không thể thực hiện lệnh `getServingFromPot` nếu trong mâm không còn đồ ăn.
- Người nấu có thể thực hiện `putServingsInPot` chỉ khi mà trong mâm hết sạch phần ăn.

Vấn đề đặt ra là thêm các code cho người ăn và người nấu ăn sao cho thỏa mãn ràng buộc đồng bộ.

Đầu tiên,ta thử dùng một đèn báo để theo dõi số lượng phần ăn,giống như bài toán `Producer-Consumer`.

Nhưng để báo cho người nấu ăn khi mâm hết thức ăn, một luồng sẽ phải biết rằng, trước khi giảm đèn báo,liệu nó có phải đợi hay không.và chúng ta không thể làm được vậy.

Một cách thay thế là sử dụng một báng tính cũng để giám sát lượng đồ ăn. Nếu một người thấy bộ đếm bằng 0, hắn sẽ gọi người nấu ăn và đợi đến khi đầy đồ ăn.

Dưới đây là các biến mà ta sẽ dùng:

```
servings = 0
mutex = Semaphore(1)
emptyPot = Semaphore(0)
emptyPot = Semaphore(0)
```

Dễ thấy: `emptyPot` biểu thị cho mâm trống và `fullPot` biểu thị cho mâm đầy.

Giải pháp được nêu ra là 1 phép so sánh bảng tính với một mốc thời gian .Sau đây là code cho người nấu ăn:

```
while True
    emptyPot.wait()
    putServingsInPot(M)
    fullPot.signal()
```

Đoạn code cho người ăn chỉ tinh vi hơn 1 chút.Khi mà một người ăn dùng textttmutex, hắn kiểm tra mâm.

- Nếu mâm trống không, hắn ra hiệu cho người nấu và chờ đợi.
- Nếu không, hắn giảm `servings` đi 1 và lấy đồ ăn trong mâm ra để ăn.

```
while True
    mutex.wait()
        If servings == 0
        emptyPot.signal()
        fullPot.wait()
        servings = M
        servings -= 1
        getServingsFromPot()
    mutex.signal()

    eat()
```

Trông có vẻ kỳ lạ khi người ăn,thay vì người nấu,cho giá trị `servings = 0`. Điều đó không cần thiết; khi người nấu thực hiện lệnh `putServingsInPot`, chúng ta biết rằng người ăn đang giữ `mutex` đang đợi `fullPot`. Vì vậy người nấu có thể truy cập `servings` một cách an toàn. Nhưng trong trường hợp này,tác giả cho người ăn làm điều đó để thấy rõ ràng tất cả các truy cập vào `servings` đều phải thông qua `mutex`.

Giải pháp này tránh gặp tình trạng **deadlock**. Trường hợp duy nhất xảy ra deadlock khi mà người ăn giữ `mutex` đợi đến khi `fullPot`. Khi hắn đợi, các người khác đang xếp hàng đợi `mutex`. Nhưng sau khi gười nấu ăn thực hiện và ra hiệu cho `fullPot`, những người ăn đang đợi có thể tiếp tục và sử dụng `mutex`.

Liệu cách giải quyết này có đảm bảo được mâm là luồng an toàn,hoặc có đảm bảo được `putServingsInPot` và `getServingFromPot` thực thi riêng biệt?

3.6 Bài toán Sự hình thành phân tử nước

3.6.1 Mô tả bài toán

Bài toán này được đề xuất bởi 1 lớp Hệ điều hành tại U.C Barkeyley ít nhất 1 thập kỷ. Bài toán dựa trên 1 bài tập trong Andrew's Concurrent Programming. [1]

Có 2 loại nguyên tử: **oxy** và **hydro**. Để kết hợp những nguyên tử này thành các phân tử nước, ta phải tạo ra một **barrier** (vách ngăn) để khiến mỗi nguyên tử đợi đến khi 1 phân tử hoàn chỉnh sẵn sàng.

Khi mỗi nguyên tử vượt qua `barrier`, nó sẽ gọi ra 1 lời gọi trước khi một nguyên tử nào đó từ phân tử kế tiếp thực thi.

Nói cách khác:

- Nếu một nguyên tử oxy đến `barrier` khi ở đây không có nguyên tử hydro nào, nó sẽ phải đợi 2 nguyên tử hydro.
- Nếu 1 nguyên tử hydro đến `barrier` khi ở đây không có nguyên tử nào, nó phải đợi 1 nguyên tử oxy và 1 nguyên tử hydro khác.

Ta không cần lo về việc ghép các nguyên tử với nhau, các nguyên tử không cần phải biết nguyên tử nào sẽ ghép với nó. Quan trọng là các nguyên tử vượt qua barrier trong cùng 1 cặp; do đó, nếu chúng ta giám sát chuỗi các nguyên tử gọi liên kết và chia chúng thành các nhóm gồm 3 nguyên tử, mỗi nhóm gồm 1 nguyên tử oxy và 2 nguyên tử hydro.

Vấn đề đặt ra: Tạo ra 1 mã lệnh đồng bộ cho nguyên tử tử oxy và hydro sao cho thỏa mãn ràng buộc trên.

3.6.2 Phương pháp giải quyết

Dưới đây là các biến tác giả sử dụng trong lời giải:

```
1 mutex = Semaphore(1)
2 oxygen = 0
3 hydrogen = 0
```

```
barrier = Barrier(3)
oxyQueue = Semaphore(0)
hydroQueue = Semaphore(0)
```

Trong đó:

- `oxygen` và `hydrogen` là các biến đếm,được quản lý bởi `mutex`. `barrier` là nơi mà mỗi tập hợp 3 nguyên tử gặp nhau sau khi gọi liên kết và trước khi cho phép tập tiếp theo thực hiện.
- `oxyQueue` là `semaphore`: nguyên tử `oxygen` đợi, tương tự với `hydroQueue`. Tác giả sử dụng các tên quy ước cho hàng đợi, vì vậy `oxyQueue` nghĩa là "đi vào hàng oxy" và `oxyQueue.singal()` nghĩa là "giải phóng một nguyên tử oxy từ hàng".

Ban đầu, `hydroQueue` và `oxyQueue` bị khóa. Khi một nguyên tử oxy đến, nó báo hiệu **hydroQueue** hai lần, cho phép hai hydro tiếp tục. Sau đó nguyên tử oxy chờ các nguyên tử hydro đến.

Mã lệnh của oxy như sau:

```
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 nguyên tử oxy tiến vào, nó nhân `mutex` và kiểm tra bảng đếm.

- Nếu có ít nhất 2 nguyên tử hydro đang đợi, nó ra hiệu cho 2 nguyên tử đó và tự liên kết với nhau.
- Nếu không, nó giải phóng `mutex` và đợi.

Sau khi liên kết, các nguyên tử đợi ở barrier cho đến khi cả 3 nguyên tử liên kết xong, và sau đó oxy giải phóng `mutex`. Khi có duy nhất 1 nguyên tử oxy trong

tập 3 nguyên tử, ta có thể đảm bảo việc ra hiệu cho `mutex` 1 lần.

Mã lệnh cho hydro cũng tương tự:

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

hydroQueue.wait()
bond()

barrier.wait()
```

Một điểm mới lạ của lời giải này là điểm dừng của `mutex` là không rõ ràng. Trong vài trường hợp, các nguyên tử giữ `mutex`, cập nhật biến đếm, và thoát`mutex`. Nhưng khi 1 nguyên tử tiến vào trạng thái hình thành phân tử, nó phải giữ `mutex` để ngăn các nguyên tử khác cho đến khi tập hợp gọi 1 liên kết.

Sau khi gọi liên kết, 3 nguyên tử đợi tại **barrier**. Khi 1 `barrier` mở, ta biết rằng 3 nguyên tử đó đã gọi liên kết và 1 trong số chúng giữ `mutex`. Ta biết được nguyên tử nào gọi liên kết và 1 trong số chúng giữ `mutex`. Ta không biết chính xác được nguyên tử nào đang giữ `mutex`, nhưng không sao cả miễn là nó giải phỏng `mutex`. Khi ra biết được chỉ có 1 nguyên tử oxy, nó sẽ thực hiện công việc này.

Lời giải này trông có vẻ không đúng, vì cho đến này,nó đa phần đúng khi mà 1 nguyên tử giữ khóa để giải phóng. Nhưng không có luật nào nói rằng điều đó phải đúng. Đây là một trong những trường hợp có thể gây nhầm lẫn tơi suy nghĩ 1 `mutex` như là 1 dấu hiệu mà các nguyên tử đạt được và giải phóng.

3.7 Bài toán Qua sông

3.7.1 Mô tả bài toán

Bài toán này được lấy từ tập bài viết bởi *Anthony Joseph ở U.C.Berkeley*, nhưng tác giả không biết liệu đó có phải là người sáng tác gốc hay không. Bài toán gần giống với bài toán ở phần 6 trong việc có một `barrier` cho phép 1 thread đi qua phải có sự liên kết với các thread khác. [1]

Ở một nơi nào đó gần Redmond, Washington có một chiếc thuyền chèo được sử dụng bởi **Hacker Linux** (hacker) và **nhân viên Microsoft** (serf) để vượt qua

một con sông. Thuyền có thể chứa chính xác bốn người; nó sẽ không rời khỏi bờ với nhiều hơn hoặc ít hơn.Để đảm bảo an toàn của khách, không được phép có 1 hacker cùng với 3 serf. Các trường hợp khác được cho phép.

Khi 1 người lên thuyền, thuyền sẽ gọi 1 hàm tên là `board`. Bạn phải đảm bảo rằng 4 người trong 1 chuyến phải gọi board trước khi những người trong chuyến khác gọi.

Cuối cùng, 4 người đều đã gọi board, duy nhất 1 trong số họ sẽ gọi 1 hàm tên là `rowBoat`, nghĩa là người đó sẽ thực hiện việc chèo thuyền. Không quan trọng ai gọi `rowBoat`, miễn là có ai đó gọi. Không cần phải lo lắng về hướng di chuyển. Giả sử mọi người đều chỉ đi một hướng duy nhất.

3.7.2 Phương pháp giải quyết

Sau đây là các biến tác giả sử dụng trong lời giải:

```
barrier = Barrier(4)
mutex = Semaphore(1)
hackers = 0
serfs = 0
hackerQueue = Semaphore(0)
serfQueue = Semaphore(0)
local isCaptain = False
```

Các biến `hackers` và `serfs` biểu thị số hacker và serf đợi board. Vì họ đều được kiểm soát bởi `mutex`, ta có thể kiểm tra tình trạng của cả hai mà không phải lo không cập nhật kịp thời.

Trong đó, `hackerQueue` và `serfQueue` cho phép chúng ta kiểm soát được lượng hacker và seft dùng thuyền.

- `barrier` đảm bảo cả 3 người gọi board trước khi 1 người gọi `rowBoat`.
- `isCaptain` là 1 biến địa phương ,biểu thị cho người gọi `row`.

Ý tưởng của thuật toán là với mỗi lần cập nhật một trong các bộ đếm và sau đó kiểm tra xem có đầy đủ không.

Tác giả sẽ trình bày mã lệnh của hackers.

```
mutex.wait()
    hackers += 1
    if  hackers == 4:
        hackerQueue.signal(4)
```

```
5           hackers = 0
6           isCaptain = True
7       elif hackers == 2 and serfs >= 2:
8           hackerQueue.signal(2)
9           serfQueue.signal(2)
10          serfs -= 2
11          hackers = 0
12          isCaptain = True
13      else:
14          mutex.signal()
15
16  hackerQueue.wait()
17
18  board()
19  barrier.wait()
20
21  if isCaptain:
22      rowBoat()
23      mutex.signal()
```

Khi một người thông qua *mutual exclusion section*, người đó kiểm tra xem đã sẵn sàng lên thuyền chưa. Nếu có, người đó ra hiệu cho mọi người, nhận mình là `captain` (người lái thuyền) và giữ `mutex` để chặn những người khác cho đến khi thuyễn xuất phát. `barrier` sẽ kiểm tra xem có bao nhiêu người đã lên thuyền.Khi người cuối cùng lên thuyền, `captain` gọi hàng và giải phóng `mutex`.

3.8 Bài toán Cầu đông - tây

3.8.1 Mô tả bài toán

Có một cây cầu hướng theo trục đông - tây.

Cầu này quá hẹp để cho phép các xe đi qua theo cả hai hướng cùng một lúc. Do đó, các xe phải luân phiên đi qua cầu. Cầu cũng không đủ mạnh để chịu tải hơn ba xe cùng một lúc.

Nhiệm vụ là tìm một giải pháp để các xe có thể qua cầu mà không gây ra tình trạng chết đói (starvation).

Điều này có nghĩa là các xe muốn qua cầu phải được qua cầu cuối cùng. Tuy nhiên, chúng ta cũng muốn tối đa hóa việc sử dụng cầu, tức là ba xe nên qua cầu cùng một lúc nếu có thể. Nếu một xe rời khỏi cầu đi về phía đông và không có xe nào đi về phía tây, thì xe tiếp theo đi về phía đông nên được phép qua cầu. Chúng ta không muốn giải pháp chỉ cho phép ba xe đi qua cầu một lần mà phải đợi cho đến khi cả ba xe đó qua xong trước khi cho phép xe khác đi.

3.8.2 Phương pháp giải quyết

Bài toán này có thể được xem như là một biến thể của bài toán nhiều độc giả - nhiều người viết (multiple readers-writers problem). Cách giải quyết như sau:

- Nếu bạn là một độc giả (xe đi về phía đông), và bạn thấy người viết (xe đi về phía tây) đang trong phòng (trên cầu), bạn ngồi chờ (đợi).
- Nếu bạn là một độc giả và có ít hơn ba độc giả trong phòng, và không có người viết nào đang chờ, bạn vào phòng.
- Nếu bạn là một độc giả và đã có ba độc giả trong phòng, bạn chờ.
- Nếu bạn là một độc giả và đang rời khỏi phòng, và không có người viết nào đang chờ, nhưng có độc giả đang chờ, bạn cho một độc giả vào phòng.
- Nếu có người viết đang chờ, và bạn không phải là độc giả cuối cùng rời khỏi phòng, bạn chỉ rời đi.
- Nếu bạn là độc giả cuối cùng rời khỏi phòng, và có người viết đang chờ, bạn cho tối đa ba người viết vào phòng.

Dưới đây là đoạn mã cho quá trình của độc giả. Quá trình của người viết tương tự, chỉ cần thay `r` bằng `w` và ngược lại. Nhắc lại, `r_active` là số độc giả đang hoạt động (trong vùng quan trọng), `r_waiting` là số độc giả đang chờ vào vùng quan trọng. `r_sem` là `semaphore` mà các độc giả đang chờ sẽ ngủ trên đó. Người viết có các biến tương tự.`lock` là một `semaphore mutex` dùng cho cả độc giả và người viết.

```
1  P(lock); // lock là một mutex và được khởi tạo là 1.
2  if ((nw_active + nw_waiting == 0) && (nr_active < 3))
3  {
4     nr_active++; // Thông báo chúng ta đang hoạt động
5     V(r_sem);  // Cho phép chúng ta đi qua
6  }
7  else
8     nr_waiting++;  // Chúng ta đang chờ
9  V(lock);
10 P(r_sem);  // Độc giả sẽ chờ ở đây nếu họ phải chờ.
11
12 ĐỌC...
13
14 P(lock);
15 nr_active--;
16 if ((nr_active == 0) && (nw_waiting > 0))
17 { // Nếu chúng ta là độc giả cuối cùng
18    count = 0;  // Đảm bảo chỉ có tối đa 3 người viết vào
```

```
19      while ((nw_waiting > 0) && (count < 3))
20      {
21          V(w_sem);  // Đánh thức một người viết;
22          w_active++;  // Thêm một người viết hoạt động
23          w_waiting--; // Bớt một người viết đang chờ.
24          count++;
25      }
26  }
27  else if ((nw_waiting == 0) && (nr_waiting > 0))
28  {  // Cho phép một độc giả đang chờ khác vào, nếu không có người viết đang chờ
29      V(r_sem);
30      nr_active++;  // Thêm một độc giả hoạt động
31      nr_waiting--; // Bớt một độc giả đang chờ.
32  }
33  V(lock);
```

Điều kiện an toàn: Độc giả và người viết không thể hoạt động cùng một lúc.

- `nr_active <= 3`
- `nw_active <= 3`
- `nr_active > 0 -> nw_active == 0`
- `nw_active > 0 -> nr_active == 0`
- `nr_active > 0 & nr_waiting > 0 -> nw_waiting > 0`
- `nw_active > 0 & nw_waiting > 0 -> nr_waiting > 0`

3.9 Bài toán Ông già Noel

3.9.1 Mô tả bài toán

Vấn đề này xuất phát từ cuốn Operating Systems của William Stallings, nhưng ông gán nó cho John Trono của St. Michael's College ở Vermont.

Ông già Noel ngủ trong cửa hàng của mình tại Bắc Cực và chỉ có thể bị đánh thức bởi một trong hai tình huống: (1) tất cả chín con tuần lộc trở về từ kỳ nghỉ ở Nam Thái Bình Dương, hoặc (2) một số yêu tinh gặp khó khăn trong việc làm đồ chơi; để cho phép ông già Noel ngủ, các yêu tinh chỉ có thể đánh thức ông khi có ba yêu tinh gặp vấn đề.

Khi ba yêu tinh đang được giải quyết vấn đề, bất kỳ yêu tinh nào khác muốn gặp ông già Noel phải chờ cho đến khi những yêu tinh kia trở lại. Nếu ông già Noel thức dậy và thấy ba yêu tinh đang chờ tại cửa hàng của mình, cùng với con tuần lộc cuối cùng trở về từ vùng nhiệt đới, ông già Noel quyết định rằng các yêu tinh có

thể chờ đến sau Giáng Sinh, vì điều quan trọng hơn là chuẩn bị cho chiếc xe trượt tuyết. (Người ta giả định rằng các con tuần lộc không muốn rời khỏi vùng nhiệt đới, và do đó chúng ở lại đó cho đến thời điểm cuối cùng có thể.) Con tuần lộc cuối cùng đến phải gọi ông già Noel trong khi những con khác chờ trong một túp lều ấm áp trước khi được cài vào xe trượt tuyết.

Dưới đây là một số yêu cầu bổ sung:

- Sau khi con tuần lộc thứ chín đến, ông già Noel phải gọi hàm `prepareSleigh`, và sau đó tất cả chín con tuần lộc phải gọi hàm `getHitched`.
- Sau khi yêu tinh thứ ba đến, ông già Noel phải gọi hàm `helpElves`. Đồng thời, cả ba yêu tinh phải gọi hàm `getHelp`.
- Tất cả ba yêu tinh phải gọi hàm `getHelp` trước khi bất kỳ yêu tinh nào khác vào (tăng bộ đếm yêu tinh).
- Ông già Noel nên chạy trong một vòng lặp để có thể giúp nhiều nhóm yêu tinh. Chúng ta có thể giả định rằng có đúng 9 con tuần lộc, nhưng có thể có bất kỳ số lượng yêu tinh nào.

3.9.2 Phương pháp giải quyết

Đầu tiên, chúng ta khởi tạo các biến đếm và `semaphore` cần thiết để đồng bộ hóa các luồng. Biến `elves` và `reindeer` được dùng để đếm số yêu tinh và tuần lộc. Các`semaphore santaSem`, `reindeerSem`, `elfTex`, và `mutex` được sử dụng để điều khiển truy cập vào các đoạn mã quan trọng.

```
Initialize elves = 0
Initialize reindeer = 0
Initialize semaphores santaSem, reindeerSem, elfTex, mutex
```

Hàm `SantaClaus` mô tả hành vi của ông già Noel. Ông già Noel sẽ ngủ cho đến khi được đánh thức bởi một trong hai tình huống: tất cả 9 tuần lộc đã trở về hoặc có 3 yêu tinh cần giúp đỡ. Nếu tất cả tuần lộc đã trở về, ông già Noel sẽ chuẩn bị xe trượt tuyết và đánh thức tất cả tuần lộc. Nếu có 3 yêu tinh cần giúp đỡ, ông già Noel sẽ giúp các yêu tinh.

```
Function SantaClaus()
    Print "Santa Claus: Hoho, here I am"
    While true
        Wait(santaSem) // Ông già Noel ngủ cho đến khi được đánh thức
                       // bởi semaphore santaSem
        Wait(mutex) // Dùng mutex để đảm bảo đoạn mã dưới đây là vùng găng
        If reindeer == 9  // Nếu tất cả 9 tuần lộc đã trở về
            Print "Santa Claus: preparing sleigh"
```

```
            For r = 0 to 8
                Post(reindeerSem) // Đánh thức tất cả tuần lộc để chuẩn bị xe
            Print "Santa Claus: make all kids in the world happy"
            reindeer = 0 // Đặt lại biến đếm tuần lộc
        Else If elves == 3 // Nếu có 3 yêu tinh gặp vấn đề
            Print "Santa Claus: helping elves"
        Post(mutex)
    End While
```

Hàm `Reindeer` mô tả hành vi của tuần lộc. Tuần lộc ngủ và sau đó tăng biến đếm `reindeer`. Nếu tuần lộc cuối cùng (thứ 9) trở về, nó sẽ đánh thức ông già Noel. Tuần lộc đợi cho đến khi ông già Noel chuẩn bị xong xe trượt tuyết, sau đó tuần lộc sẽ được cài vào xe.

```
Function Reindeer(id)
    Print "This is reindeer " + id
    While true
        Wait(mutex)
        reindeer++
        If reindeer == 9 // Nếu tuần lộc cuối cùng trở về
            Post(santaSem) // Đánh thức ông già Noel
        Post(mutex)
        Wait(reindeerSem) // Đợi ông già Noel chuẩn bị xong xe trượt tuyết
        Print "Reindeer " + id + " getting hitched"
        Sleep(20)
```

Hàm `Elve` mô tả hành vi của yêu tinh. Yêu tinh làm việc, thỉnh thoảng một yêu tinh cần sự giúp đỡ. Nếu yêu tinh cần giúp đỡ, nó sẽ kiểm tra `semaphore elfTex` để đảm bảo không có hơn 3 yêu tinh vào cùng một lúc. Sau khi được giúp đỡ, yêu tinh giảm biến đếm elves và giải phóng `semaphore elfTex` nếu không còn yêu tinh nào cần giúp đỡ.

```
Function Elve(id)
    Print "This is elve " + id
    While true
        need_help = Random < 10%
        If need_help // Nếu yêu tinh cần giúp đỡ
            Wait(elfTex) // Kiểm tra semaphore elfTex để đảm bảo
                         // không có hơn 3 yêu tinh vào cùng một lúc
            Wait(mutex)
            elves++
            If elves == 3 // Nếu có 3 yêu tinh cần giúp đỡ
                Post(santaSem) // Đánh thức ông già Noel
            Else
                Post(elfTex)
```

```
            Post(mutex)
            Print "Elve " + id + " will get help from Santa Claus"
            Sleep(10)
            Wait(mutex)
            elves--
            If elves == 0
                Post(elfTex) // Giải phóng semaphore elfTex nếu
                             // không còn yêu tinh nào cần giúp đỡ
            Post(mutex)
        Print "Elve " + id + " at work"
        Sleep(random_time)
```

Trong hàm `main`, chúng ta khởi tạo các biến đếm và `semaphore`. Tiếp theo, chúng ta tạo luồng cho ông già Noel, 9 tuần lộc và 10 yêu tinh. Cuối cùng, chúng ta chờ cho luồng của ông già Noel kết thúc.

```
Main()
    Initialize variables and semaphores
    Create SantaClaus thread
    For r = 0 to 8
        Create Reindeer thread with id r+1
    For e = 0 to 9
        Create Elve thread with id e+1
    Join SantaClaus thread
```

Phương pháp này đảm bảo rằng ông già Noel chỉ bị đánh thức khi cần thiết, và không có xung đột giữa các yêu tinh và tuần lộc. Các `semaphore` đảm bảo rằng các yêu tinh và tuần lộc hoạt động đúng thứ tự và không vi phạm các quy tắc đã đề ra.

3.10 Bài toán Bưu điện

3.10.1 Mô tả bài toán

Chương trình mô phỏng hoạt động của một bưu điện, nơi có 50 khách hàng đến và yêu cầu 3 nhân viên bưu điện thực hiện các nhiệm vụ khác nhau như mua tem, gửi thư, hoặc gửi bưu kiện. Mỗi nhiệm vụ có thời gian hoàn thành khác nhau tùy thuộc vào từng loại công việc.

Một nhân viên bưu điện chỉ có thể bắt đầu phục vụ khách hàng tiếp theo khi đã hoàn thành công việc với khách hàng hiện tại. Trong thời gian chờ đợi, các khách hàng khác cần phải đợi đến lượt của mình. Tất cả các nhiệm vụ phải được thực hiện theo một thứ tự nhất định.

Ví dụ, nhân viên bưu điện phải được phân công cho một khách hàng trước khi khách hàng đó có thể yêu cầu nhân viên thực hiện công việc của mình. Do đó,

cần có sự loại trừ lẫn nhau giữa các sự kiện khác nhau và điều này được kiểm soát bằng cách sử dụng các semaphore.

Các semaphore được sử dụng để thực hiện các hoạt động như chỉ cho phép một số lượng khách hàng nhất định vào bưu điện, hoặc cho phép một nhiệm vụ được thực hiện chỉ sau khi một nhiệm vụ khác đã hoàn thành. Có một số giả định đã được đưa ra, ví dụ như 60 giây được coi là tương đương với 1 giây để chương trình chạy nhanh hơn. Theo cách này, tất cả 50 khách hàng sẽ lần lượt vào bưu điện, chọn nhiệm vụ, được phân công cho một nhân viên, yêu cầu nhân viên thực hiện nhiệm vụ của họ và cuối cùng rời đi để nhường chỗ cho các khách hàng khác.

3.10.2 Phương pháp giải quyết

Một trong những thách thức chính là đảm bảo rằng mỗi luồng (thread) thực hiện công việc của mình một cách riêng biệt. Việc in ra thông tin về nhân viên nào đang phục vụ khách hàng nào, cũng như khách hàng nào yêu cầu nhân viên nào thực hiện nhiệm vụ, khá khó khăn do có các luồng riêng biệt cho cả khách hàng và nhân viên.

Cần có một cơ chế để luồng của nhân viên biết được số của khách hàng mà nó đang phục vụ. Tương tự, cần có cơ chế để mỗi luồng khách hàng biết nhân viên bưu điện nào đã được phân công cho nó.

Chúng tôi sử dụng một hàng đợi để lưu trữ số của khách hàng mỗi khi khách hàng vào bưu điện và đến bàn làm việc của nhân viên. Bất kỳ luồng nhân viên bưu điện nào rảnh sẽ lấy số khách hàng từ hàng đợi và được phân công cho khách hàng đó. Ngoài ra, trong mỗi đối tượng khách hàng có một trường lưu trữ nhân viên nào đã được phân công cho nó. Trường này trở nên hữu ích khi khách hàng yêu cầu nhân viên tương ứng thực hiện một nhiệm vụ trong luồng riêng của mình.

Đầu tiên, chúng ta khởi tạo các biến và `semaphore` cần thiết để điều phối các hoạt động trong bưu điện. Các `semaphore` được sử dụng để giới hạn số lượng khách hàng trong bưu điện, quản lý việc sử dụng các bàn làm việc của nhân viên, và điều phối việc hoàn thành các nhiệm vụ.

```
1 Initialize semaphores maxCapacity(10), custReady(0), workerDesk(3), mutex(1),
2                       coord(0), leave_workerDesk(0), coord2(0), scale(1)
3 Initialize global variables count = 0, queue = empty queue
4 Initialize arrays finished[50] with semaphores(0), objCust[50],
5                   objWork[3], t1[50], t2[3]
```

Hàm khởi tạo và hoạt động của nhân viên bưu điện. Mỗi nhân viên bưu điện được khởi tạo với một ID duy nhất. Trong hàm run của nhân viên, các nhân viên sẽ liên tục chờ khách hàng và phục vụ họ theo thứ tự.

```
Function Worker.run()
    Print "Postal Worker " + workerNum + " created"
    While true
        Wait(custReady) // Chờ khách hàng sẵn sàng
        Wait(mutex) // Đảm bảo truy cập đồng thời an toàn vào hàng đợi
        next_cust_number = queue.remove() // Lấy số khách hàng từ hàng đợi
        Print "Postal worker " + workerNum + " serving Customer " +
        next_cust_number
        objCust[next_cust_number].worker_assigned = workerNum
        Signal(mutex) // Giải phóng mutex
        Signal(coord) // Thông báo rằng khách hàng đã được phục vụ
        Wait(coord2) // Chờ tín hiệu từ khách hàng
        serve_cust() // Phục vụ khách hàng
        Signal(finished[next_cust_number]) // Thông báo hoàn thành nhiệm vụ
        Wait(leave_workerDesk) // Chờ khách hàng rời bàn làm việc
        Signal(workerDesk) // Giải phóng bàn làm việc
```

Hàm `serve_cust` của nhân viên bưu điện phục vụ khách hàng theo nhiệm vụ mà họ yêu cầu. Mỗi nhiệm vụ có thời gian thực hiện khác nhau.

```
Function Worker.serve_cust()
    task = workerQueue.remove() // Lấy nhiệm vụ từ hàng đợi
    If task == 1
        Sleep(1 second) // Mua tem
        Print "Postal worker " + workerNum + " finished serving customer " +
        next_cust_number
    Else If task == 2
        Sleep(1.5 seconds) // Gửi thư
        Print "Postal worker " + workerNum + " finished serving customer " +
        next_cust_number
    Else If task == 3
        Wait(scale) // Sử dụng cân
        Print "Scales in use by postal worker " + workerNum
        Sleep(2 seconds) // Gửi bưu kiện
        Print "Scales released by postal worker " + workerNum
        Signal(scale) // Giải phóng cân
        Print "Postal worker " + workerNum + " finished serving customer " +
        next_cust_number
```

Hàm khởi tạo và hoạt động của khách hàng. Mỗi khách hàng được khởi tạo với một ID duy nhất và một nhiệm vụ ngẫu nhiên. Trong hàm run của khách hàng, mỗi khách hàng sẽ thực hiện các bước để yêu cầu và nhận dịch vụ từ nhân viên bưu điện.

```
Function Customer.run()
    Print "Customer " + custNum + " created"
    task = Random task between 1 and 3 // Gán nhiệm vụ ngẫu nhiên cho khách hàng
    Wait(maxCapacity) // Chờ để vào bưu điện nếu đã đủ sức chứa
    Print "Customer " + custNum + " enters shop"
    Wait(workerDesk) // Chờ bàn làm việc của nhân viên bưu điện
    Wait(mutex) // Đảm bảo truy cập đồng thời an toàn vào hàng đợi
    queue.add(custNum) // Thêm số khách hàng vào hàng đợi
    Signal(custReady) // Thông báo rằng khách hàng đã sẵn sàng
    Signal(mutex) // Giải phóng mutex
    Wait(coord) // Chờ tín hiệu từ nhân viên bưu điện
    Print "Customer " + custNum + " asks postal worker "    +
           worker_assigned + " to " + taskname(task)
    PostOffice.objWork[worker_assigned].workerQueue.add(task)
                          // Thêm nhiệm vụ vào hàng đợi của nhân viên bưu điện
    Signal(coord2) // Thông báo rằng yêu cầu đã được gửi đi
    Wait(finished[custNum]) // Chờ nhiệm vụ hoàn thành
    Print "Customer " + custNum + " finished " + taskname2(task)
    Print "Customer " + custNum + " leaves post office"
    Signal(leave_workerDesk) // Giải phóng bàn làm việc của nhân viên bưu điện
    Signal(maxCapacity) // Giải phóng sức chứa của bưu điện
    count++
```

Trong hàm `main`, chúng ta khởi tạo các đối tượng khách hàng và nhân viên bưu điện, sau đó tạo và khởi chạy các luồng cho mỗi khách hàng và nhân viên.

```
Function Main()
    Initialize PostOffice
    For i = 0 to 49
        objCust[i] = new Customer(i)
        t1[i] = new Thread(objCust[i])
        t1[i].start() // Khởi chạy luồng khách hàng
    For i = 0 to 2
        objWork[i] = new Worker(i)
        t2[i] = new Thread(objWork[i])
        t2[i].start() // Khởi chạy luồng nhân viên bưu điện
    For i = 0 to 49
        t1[i].join() // Chờ các luồng khách hàng kết thúc
        Print "Joined customer " + i
    System.exit(0) // Kết thúc chương trình
```

Phương pháp này đảm bảo rằng mỗi khách hàng được phục vụ theo thứ tự họ đến, và mỗi nhân viên chỉ phục vụ một khách hàng tại một thời điểm. Các `semaphore` được sử dụng để kiểm soát truy cập đồng thời và đảm bảo rằng các nhiệm vụ được thực hiện theo đúng thứ tự.

4 Thảo luận

Quá trình nghiên cứu và thực hiện bài tập lớn này đã mang lại nhiều kiến thức và kinh nghiệm quý báu về các cơ chế và thuật toán điều độ tiến trình trong quản lý đoạn găng của hệ điều hành. Trong phần này, chúng tôi sẽ thảo luận về những kết quả đạt được, những thách thức gặp phải và các hướng nghiên cứu tiếp theo.

Một trong những kết quả quan trọng của nghiên cứu là việc hiểu rõ và áp dụng các cơ chế đồng bộ hóa như Mutex, Semaphore và Monitor. Các cơ chế này không chỉ giúp giải quyết các vấn đề tương tranh mà còn đảm bảo tính nhất quán và hiệu suất của hệ thống. Trong quá trình nghiên cứu, chúng tôi đã nhận thấy rằng mỗi cơ chế có ưu điểm và nhược điểm riêng, và việc lựa chọn cơ chế phù hợp phụ thuộc vào ngữ cảnh cụ thể của vấn đề cần giải quyết.

Mutex là một công cụ hiệu quả để đảm bảo tính loại trừ lẫn nhau, ngăn chặn nhiều tiến trình truy cập đồng thời vào cùng một tài nguyên. Tuy nhiên, Mutex có thể dẫn đến vấn đề khóa sống nếu không được quản lý đúng cách. Semaphore, với khả năng đếm và quản lý số lượng tiến trình truy cập đồng thời, cung cấp một giải pháp linh hoạt hơn, nhưng cũng phức tạp hơn trong việc triển khai. Monitor, với các biến điều kiện và khả năng quản lý đồng bộ hóa ở mức cao hơn, giúp đơn giản hóa việc lập trình và quản lý tài nguyên, nhưng đòi hỏi hiểu biết sâu hơn về cấu trúc dữ liệu và cơ chế đồng bộ hóa.

Các bài toán kinh điển như Người hút thuốc lá, Chờ xe buýt, Người nhập cư và Quán rượu Sushi đã minh họa rõ ràng cách thức áp dụng các cơ chế đồng bộ hóa để giải quyết các vấn đề cụ thể trong quản lý đoạn găng. Mỗi bài toán mang đến những thách thức và yêu cầu đặc thù, đòi hỏi sự kết hợp của nhiều phương pháp và kỹ thuật khác nhau. Quá trình phân tích và giải quyết các bài toán này đã giúp chúng tôi hiểu rõ hơn về cách các hệ điều hành hiện đại duy trì tính nhất quán và hiệu suất của hệ thống.

Một thách thức lớn trong quá trình nghiên cứu là việc xử lý tình trạng bế tắc và khóa sống. Bế tắc xảy ra khi một nhóm các tiến trình bị kẹt chờ đợi lẫn nhau, không thể tiến hành. Khóa sống xảy ra khi các tiến trình vẫn hoạt động nhưng không tiến triển được trong công việc thực tế. Việc phát hiện và giải quyết các tình trạng này đòi hỏi sự hiểu biết sâu sắc về cơ chế đồng bộ hóa và khả năng phân tích, đánh giá tình huống một cách chính xác.

5 Kết luận

Trong quá trình thực hiện bài tập lớn này, chúng tôi đã tiến hành nghiên cứu và phân tích chi tiết các cơ chế, thuật toán điều độ tiến trình, đặc biệt là trong bối cảnh quản lý đoạn găng của hệ điều hành. Các vấn đề như tương tranh, bế tắc, chết

đói và khóa sống là những thách thức lớn trong quản lý tài nguyên hệ điều hành, đòi hỏi các phương pháp tiếp cận đồng bộ hóa tiến trình hiệu quả. Qua quá trình nghiên cứu, chúng tôi đã làm rõ tầm quan trọng của việc áp dụng các giải pháp như Mutex, Semaphore, và Monitor, đồng thời minh họa những ứng dụng này thông qua các bài toán kinh điển như Người hút thuốc lá, Chờ xe buýt, Người nhập cư và Quán rượu Sushi.

Quá trình điều độ tiến trình là một trong những nhiệm vụ quan trọng nhất của hệ điều hành, vì nó đảm bảo rằng các tiến trình được thực thi một cách hiệu quả và công bằng, đồng thời tối ưu hóa việc sử dụng tài nguyên hệ thống. Một trong những khía cạnh chính của điều độ tiến trình là việc quản lý đoạn găng, nơi mà nhiều tiến trình hoặc luồng có thể cạnh tranh để truy cập vào các tài nguyên chung. Điều này đòi hỏi các giải pháp đồng bộ hóa để ngăn chặn các vấn đề như tương tranh, nơi mà kết quả của một tiến trình có thể bị ảnh hưởng bởi sự thực thi đồng thời của các tiến trình khác.

Trong quá trình nghiên cứu, chúng tôi đã xem xét các phương pháp đồng bộ hóa khác nhau. Mutex (Mutual Exclusion) là một công cụ phổ biến để quản lý quyền truy cập vào các tài nguyên chia sẻ. Nó đảm bảo rằng chỉ một tiến trình có thể truy cập tài nguyên tại một thời điểm, giúp ngăn chặn tình trạng tương tranh. Semaphore, một khái niệm mạnh mẽ hơn, không chỉ giới hạn quyền truy cập mà còn có thể đếm và quản lý số lượng tiến trình có thể truy cập vào tài nguyên đồng thời. Monitor, một cơ chế đồng bộ hóa cấp cao hơn, cho phép các tiến trình chờ và thông báo cho nhau khi một tài nguyên trở nên khả dụng, giúp tối ưu hóa việc quản lý tài nguyên và tránh tình trạng bế tắc.

Một trong những bài toán kinh điển mà chúng tôi đã nghiên cứu là bài toán Người hút thuốc lá. Đây là một ví dụ điển hình về vấn đề đồng bộ hóa, nơi mà ba người hút thuốc chia sẻ ba loại tài nguyên khác nhau: thuốc lá, giấy và diêm. Bài toán yêu cầu thiết kế một hệ thống sao cho mỗi người hút thuốc có thể làm ra một điếu thuốc khi có đủ ba loại tài nguyên. Thông qua việc áp dụng các semaphore và cơ chế đồng bộ hóa, chúng tôi đã phát triển được giải pháp hiệu quả cho bài toán này, minh họa cách các tiến trình có thể phối hợp với nhau để tránh tình trạng tương tranh và bế tắc.

6 Lời cảm ơn

Xin bày tỏ lòng biết ơn sâu sắc đến thầy Phạm Đăng Hải, giảng viên bộ môn Nguyên lý hệ điều hành tại Trường Công nghệ Thông tin và Truyền thông, Đại học Bách khoa Hà Nội. Sự nhiệt tình và kiến thức sâu rộng của thầy đã không chỉ truyền cảm hứng cho chúng em trong quá trình học tập mà còn hỗ trợ chúng em rất nhiều trong việc hoàn thành báo cáo này. Những bài giảng quý báu của thầy đã giúp chúng em hiểu sâu sắc hơn về các khái niệm và vấn đề phức tạp liên quan đến hệ điều hành, từ đó cải thiện đáng kể chất lượng nghiên cứu. Xin chân thành cảm ơn thầy vì đã luôn kiên nhẫn và hỗ trợ chúng em trong học tập!

Tài liệu

[1] A. Downey. *The Little Book of Semaphores (2nd Edition): The Ins and Outs of Concurrency Control and Common Mistakes*. CreateSpace Independent Publishing Platform, 2009.

[2] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[3] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson, 3rd edition, 2008.

[4] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, January 2006.