# CHƯƠNG 9. ĐỒNG BỘ HÓA TIẾN TRÌNH

Một **tiến trình hợp tác** là một tiến trình có thể gây ảnh hưởng hay bị ảnh hưởng tới tiến trình khác đang thực thi trong hệ thống. Các tiến trình hợp tác có thể chia sẻ trực tiếp không gian địa chỉ lô gic (mã và dữ liệu), hay được phép chia sẻ dữ liệu thông qua các tập tin. Trường hợp đầu đạt được thông qua việc sử dụng các tiến trình có trọng lượng nhẹ hay luồng. Truy xuất đồng hành dữ liệu được chia sẻ có thể dẫn tới việc không đồng nhất dữ liệu. Trong chương này chúng ta sẽ thảo luận các cơ chế đảm bảo việc thực thi có thứ tự của các tiến trình hợp tác chia sẻ không gian địa chỉ để tính đúng đắn của dữ liệu luôn được duy trì.

## I. Tổng quan

Trong chương trước, chúng ta phát triển một mô hình hệ thống chứa số lượng tiến trình hợp tác tuần tự, tất cả chúng chạy bất đồng bộ và có thể chia sẻ dữ liệu. Chúng ta hiển thị mô hình này với cơ chế vùng đệm có kích thước giới hạn, được đại diện cho hệ điều hành.

Chúng ta xét giải pháp bộ nhớ được chia sẻ cho bài toán vùng đệm có kích thước giới hạn. Giải pháp này cho phép có nhiều nhất BUFFER_SIZE –1 sản phẩm trong vùng đệm tại cùng thời điểm. Giả sử rằng chúng ta muốn hiệu chỉnh giải thuật để giải quyết sự thiếu sót này. Một khả năng là thêm một biến đếm số nguyên **counter,** được khởi tạo bằng 0. counter được tăng mỗi khi chúng ta thêm một sản phẩm tới vùng đệm và bị giảm mỗi khi chúng ta lấy một sản phẩm ra khỏi vùng đệm.

Mã cho tiến trình người sản xuất có thể được hiệu chỉnh như sau:

```
while (1)
    {/*tạo sản phẩm trong nextProduced*/
        while (counter==BUFFER_SIZE); /*không làm gì cả*/
        buffer[in] = nextProduced;
        in = ( in + 1 ) % BUFFER_SIZE;
        counter++;
    }
```

Mã cho tiến trình người tiêu dùng có thể được hiệu chỉnh như sau:

```
while (1)
{       while (counter == 0) ; /*không làm gì cả*/
        nextConsumed = buffer[out];
        out = ( out + 1 ) % BUFFER_SIZE;
        counter--; /*tiêu thụ sản phẩm trong nextConsumed*/
}
```

Mặc dù cả hai thủ tục người sản xuất và người tiêu dùng thực thi đúng khi tách biệt nhau nhưng chúng không thực hiện đúng chức năng khi thực thi đồng hành. Như minh hoạ dưới đây, giả sử rằng giá trị của biến counter hiện tại là 5 và thủ tục người

sản xuất và người tiêu dùng thực thi đồng hành câu lệnh "counter++" và "counter--".

Theo sau việc thực thi hai câu lệnh này, giá trị của biến counter có thể là 4, 5 hay 6! Kết quả chỉ đúng khi biến counter==5, được tạo ra đúng nếu tiến trình người sản xuất và người tiêu dùng thực thi riêng biệt.

Chúng ta có thể minh hoạ giá trị của counter có thể không đúng như sau. Chú ý, câu lệnh "counter++" có thể được cài đặt bằng ngôn ngữ máy (trên một máy điển hình) như sau:

$register_1 = counter$

$register_1 = register_1 + 1$

$counter = register_1$

Ở đây $register_1$ là một thanh ghi CPU cục bộ. Tương tự, câu lệnh "counter--" được cài đặt như sau:

$register_2 = counter$

$register_2 = register_2 - 1$

$counter = register_2$

Ở đây $register_2$ là thanh ghi CPU cục bộ. Dù là $register_1$ và $register_2$ có thể dùng cùng thanh ghi vật lý, nhưng nội dung của thanh ghi sẽ được lưu lại và lấy lại bởi bộ quản lý ngắt.

Thực thi đồng hành của "counter++" và "counter--" là tương tự như thực thi tuần tự ở đây các câu lệnh cấp thấp hơn được hiện diện trước bị phủ lắp trong thứ tự bất kỳ (nhưng thứ tự bên trong mỗi câu lệnh cấp cao được lưu giữ). Một sự phủ lắp là:

| | | | |
|---|---|---|---|
| $T_0$: producer | thực thi | $register_1 = counter$ | {$register_1 = 5$} |
| $T_1$: producer | thực thi | $register_1 = register_1 + 1$ | {$register_1 = 6$} |
| $T_2$: consumer | thực thi | $register_2 = counter$ | {$register_2 = 5$} |
| $T_3$: consumer | thực thi | $register_2 = register_2 - 1$ | {$register_2 = 4$} |
| $T_4$: producer | thực thi | $counter = register_1$ | {$counter = 6$} |
| $T_5$: consumer | thực thi | $counter = register_2$ | {$counter = 4$} |

Chú ý rằng, chúng ta xem xét tình trạng không đúng "counter==4" theo đó có 4 vùng đệm đầy, nhưng thực tế khi đó có 5 vùng đệm đầy. Nếu chúng đổi ngược lại thứ tự của câu lệnh $T_4$ và $T_5$, chúng ta sẽ có trạng thái không đúng "counter ==6".

Chúng ta đi đến trạng thái không đúng này vì chúng ta cho phép cả hai tiến trình thao tác đồng thời trên biến counter. Trường hợp tương tự, ở đây nhiều tiến trình truy xuất và thao tác cùng dữ liệu đồng hành và kết quả của việc thực thi phụ thuộc vào thứ tự xác định trong đó việc truy xuất xảy ra, được gọi là **điều kiện cạnh tranh** (race condition). Để ngăn chặn điều kiện cạnh tranh ở trên, chúng ta cần đảm bảo rằng chỉ một tiến trình tại một thời điểm có thể được thao tác biến counter. Để thực hiện việc đảm bảo như thế, chúng ta yêu cầu một vài hình thức đồng bộ hoá tiến trình. Những trường hợp như thế xảy ra thường xuyên trong các hệ điều hành khi các phần khác nhau của hệ thống thao tác các tài nguyên và chúng ta muốn các thay đổi không gây trở ngại một sự thay đổi khác. Phần chính của chương này là tập trung vào vấn đề đồng bộ hoá và cộng tác tiến trình.

## II. Vấn đề vùng tương trục (miền găng)

Xét một hệ thống gồm n tiến trình ($P_0$, $P_1$, … ,$P_{n-1}$ ). Mỗi tiến trình có một phân đoạn mã, được gọi là **vùng tương trục** (miền găng, critical section), trong đó tiến trình này có thể thay đổi những biến dùng chung, cập nhật một bảng, viết đến tập tin,.. Đặc điểm quan trọng của hệ thống là ở chỗ, khi một tiến trình đang thực thi trong vùng tương trục, không có tiến trình nào khác được phép thực thi trong vùng tương trục của nó. Do đó, việc thực thi của các vùng tương trục bởi các tiến trình là sự loại trừ hỗ tương. Vấn đề vùng tương trục là thiết kế một giao thức mà các tiến trình có thể dùng để cộng tác.

Mỗi tiến trình phải yêu cầu quyền để đi vào vùng tương trục của nó. Vùng mã thực hiện yêu cầu này là **phần đi vào** (entry section). Vùng tương trục có thể được theo sau bởi một **phần kết thúc** (exit section). Mã còn lại là **phần còn lại** (remainder

```
do {
        entry section
        critical section
        exit section
        remainder section
}
while (1);
```

**Hình 9.1. Cấu trúc chung của một tiến trình điển hình Pi**

section).

Một giải pháp đối với vấn đề vùng tương trục phải thoả mãn ba yêu cầu sau:

• **Loại trừ hỗ tương** (Mutual Exclusion): Nếu tiến trình $P_i$ đang thực thi trong vùng tương trục của nó thì không tiến trình nào khác đang được thực thi trong vùng tương trục đó.

• **Tiến triển** (Progress): nếu không có tiến trình nào đang thực thi trong vùng tương trục và có vài tiến trình muốn vào vùng tương trục thì chỉ những tiến trình không đang thực thi phần còn lại mới có thể tham gia vào việc quyết định tiến trình nào sẽ đi vào vùng tương trục tiếp theo và chọn lựa này không thể trì hoãn vô hạn định.

• **Chờ đợi có giới hạn** (bounded wait): giới hạn số lần các tiến trình khác được phép đi vào miền tương trục sau khi một tiến trình thực hiện yêu cầu để đi vào miền tương trục của nó và trước khi yêu cầu đó được gán.

Chúng ta giả sử rằng mỗi tiến trình đang thực thi với tốc độ khác 0. Tuy nhiên, chúng ta có thể thực hiện rằng không có giả thuyết nào được quan tâm về tốc tương đối của n tiến trình.

Trong phần tiếp theo chúng ta nghiên cứu để nắm được các giải pháp thoả ba yêu cầu này. Những giải pháp này không quan tâm đến các chỉ thị phần cứng hay số lượng bộ xử lý mà phần cứng hỗ trợ. Tuy nhiên chúng ta giả sử rằng những chỉ thị ngôn ngữ máy cơ bản (chỉ thị cơ bản như **load, store** và **test**) được thực hiện mang tính nguyên tử (atomically). Nghĩa là, nếu hai chỉ thị như thế được thực thi đồng hành thì kết quả tương tự như thực thi tuần tự trong thứ tự không xác định. Do đó, nếu chỉ thị **load** và **store** được thực thi đồng hành thì **load** sẽ nhận giá trị cũ hay mới như

không có sự kết hợp vừa cũ vừa mới.

Khi trình bày một giải thuật, chúng ta định chỉ những biến được dùng cho mục đích đồng bộ và mô tả chỉ một tiến trình điển hình $P_i$ mà cấu trúc của nó được hiển thị ở hình 9.1. Phần đi vào và kết thúc được bao trong hình chữ nhật để nhấn mạnh các đoạn mã quan trọng.

## III. Giải pháp

Có nhiều giải pháp để thực hiện việc loại trừ hỗ tương. Các giải pháp này, tuỳ thuộc vào cách tiếp cận trong xử lý của tiến trình bị khoá, được phân biệt thành hai lớp: chờ dợi bận (busy waiting) và nghẽn và đánh thức (sleep and wakeup)

### III.1. Giải pháp "chờ đợi bận"

#### III.1.1. Giải pháp hai tiến trình (two-Process Solution)

Trong phần này, chúng ta giới hạn việc quan tâm tới những giải thuật có thể áp dụng chỉ hai tiến trình cùng một lúc. Những tiến trình này được đánh số $P_0$ và $P_1$. Để thuận lợi, khi trình bày $P_i$, chúng ta dùng $P_j$ để chỉ tiến trình còn lại, nghĩa là j = 1 – i

**a) Giải thuật 1**

Tiếp cận đầu tiên của chúng ta là để hai tiến trình chia sẻ một biến số nguyên chung **turn** được khởi tạo bằng 0 (hay 1). Nếu turn == 0 thì tiến trình $P_i$ được phép thực thi trong vùng tương trục của nó. Cấu trúc của tiến trình $P_i$ được hiển thị trong hình 9.2.

Giải pháp này đảm bảo rằng chỉ một tiến trình tại một thời điểm có thể ở trong vùng tương trục của nó. Tuy nhiên, nó không thoả mãn yêu cầu tiến trình vì nó yêu cầu sự thay đổi nghiêm khắc của các tiến trình trong việc thực thi của vùng tương trục. Thí dụ, nếu turn == 0 và $P_1$ sẵn sàng đi vào vùng tương trục của nó thì $P_1$ không thể đi

```
do {
        while (turn!=i) ;
        critical section
        turn = j;
        remainder section
}
while (1);
```

**Hình 9.2. Cấu trúc của một tiến trình Pi trong giải thuật 1**

vào vùng tương trục thậm chí khi $P_0$ đang ở trong phần còn lại của nó

**b) Giải thuật 2**

Vấn đề với giải thuật 1 là nó không giữ lại đủ thông tin về trạng thái của mỗi tiến trình; nó nhớ chỉ tiến trình nào được phép đi vào miền tương trục. Để giải quyết vấn đề này, chúng ta có thể thay thế biến turn với mảng sau:

```
Boolean flag[2];
```

Các phần tử của mảng được khởi tạo tới flase. Nếu flag[i] là true, giá trị này

```
do {
        flag[i] = true; while (flag[j]);
        critical section
        flag[i] = false;
        remainder section
}
while (1);
```

**Hình 9.3. Cấu trúc của một tiến trình Pi trong giải thuật 2**

hiển thị rằng $P_i$ sẵn sàng đi vào vùng tương trục. Cấu trúc của tiến trình $P_i$ được hiển thị trong hình 9.3 dưới đây:

Trong giải thuật này, tiến trình $P_i$ trước tiên thiết lập flag[i] tới true, hiển thị rằng nó sẵn sàng đi vào miền tương trục. Sau đó, $P_i$ kiểm tra rằng tiến trình tiến trình $P_j$ cũng không sẵn sàng đi vào miền tương trục của nó. Nếu $P_j$ sẵn sàng thì $P_i$ sẽ chờ cho tới khi $P_j$ hiển thị rằng nó không còn cần ở trong vùng tương trục nữa (nghĩa là cho tới khi flag[j] là false). Tại thời điểm này, $P_i$ sẽ đi vào miền tương trục. Thoát ra khỏi miền tương trục, $P_i$ sẽ đặt flag[i] là false, cho phép tiến trình khác (nếu nó đang chờ) đi vào miền tương trục của nó.

Trong giải pháp này, yêu cầu loại trừ hỗ tương sẽ được thoả mãn. Tuy nhiên, yêu cầu tiến trình không được thoả mãn. Để minh hoạ vấn đề này, chúng ta xem xét thứ tự thực thi sau:

$T_0$: $P_0$ thiết lập flag[0] = true;

$T_1$: $P_1$ thiết lập flag[1] = true;

Bây giờ $P_0$ và $P_1$ được lập mãi mãi trong câu lệnh while tương ứng của chúng. Giải thuật này phụ thuộc chủ yếu vào thời gian chính xác của hai tiến trình. Thứ tự này được phát sinh trong môi trường nơi có nhiều bộ xử lý thực thi đồng hành hay nơi một ngắt (chẳng hạn như một ngắt định thời) xảy ra lập tức sau khi bước $T_0$ được thực thi và CPU được chuyển từ một tiến trình này tới một tiến trình khác.

Chú ý rằng chuyển đổi thứ tự của các chỉ thị lệnh để thiết lập flag[i] và kiểm tra giá trị của flag[j] sẽ không giải quyết vấn đề của chúng ta. Hơn nữa chúng ta sẽ có một trường hợp đó là hai tiến trình ở trong vùng tương trục cùng một lúc, vi phạm yêu cầu loại trừ hỗ tương.

**c) Giải thuật 3**

Giải thuật 3 còn gọi là giải pháp Peterson. Bằng cách kết hợp hai ý tưởng quan trọng trong giải thuật 1 và 2, chúng ta đạt được một giải pháp đúng tới với vấn đề vùng tương trục, ở đó hai yêu cầu được thoả. Các tiến trình chia sẻ hai biến:

```
Boolean flag[2]
Int turn;
```

Khởi tạo flag[0] = flag[1] = false và giá trị của turn là không xác định (hoặc là 0 hay 1). Cấu trúc của tiến trình $P_i$ được hiển thị trong hình sau:

```
do{
        flag[i] = true; turn = j; while (flag[j] &&turn ==j);
        critical section
        flag[i] = false;
        remainder section
} while (1);
```

**Hình 9.4. Cấu trúc của một tiến trình Pi trong giải thuật 3**

Để đi vào miền tương trục, tiến trình $P_i$ trước tiên đặt flag[i] là true sau đó đặt turn tới giá trị j, do đó xác định rằng nếu tiến trình khác muốn đi vào miền tương trục nó. Nếu cả hai tiến trình đi vào miền tương trục cùng một lúc turn sẽ đặt cả hai i và j tại xấp xỉ cùng một thời điểm. Chỉ một trong hai phép gán này là kết quả cuối cùng. Giá trị cuối cùng của turn quyết định tiến trình nào trong hai tiến trình được cho phép đi vào miền tương trục trước.

Bây giờ chúng ta chứng minh rằng giải pháp này là đúng. Chúng ta cần hiển thị rằng:

1) Loại trừ hỗ tương được bảo toàn

2) Yêu cầu tiến trình được thoả

3) Yêu cầu chờ đợi có giới hạn cũng được thoả

Chứng minh thuộc tính 1, chúng ta chú ý rằng mỗi $P_i$ đi vào miền tương trục của nó chỉ nếu flag[j] ==false hay turn ==i. Cũng chú ý rằng, nếu cả hai tiến trình có thể đang thực thi trong vùng tương trục của chúng tại cùng thời điểm thì flag[0] == flag[1] ==true. Hai nhận xét này ngụ ý rằng $P_0$ và $P_1$ không thể thực thi thành công trong vòng lặp while của chúng tại cùng một thời điểm vì giá trị turn có thể là 0 hay 1.

Do đó, một trong các tiến trình-$P_j$ phải được thực thi thành công câu lệnh while, ngược lại $P_i$ phải thực thi ít nhất câu lệnh bổ sung ("turn==j"). Tuy nhiên, vì tại thời điểm đó, flag[j] ==true và turn ==j, và điều kiện này sẽ không đổi với điều kiện là $P_j$ ở trong vùng miền tương trục của nó, kết quả sau việc loại trừ hỗ tương được bảo vệ

Để chứng minh thuộc tính 2 và 3, chúng ta chú ý rằng một tiến trình $P_i$ có thể được ngăn chặn từ việc đi vào miền tương trục chỉ nếu nó bị kẹt trong vòng lặp while với điều kiện flag[j] == true và turn == j. Nếu $P_j$ không sẵn sàng đi vào miền tương trục thì flag[j] == false và $P_i$ có thể đi vào miền tương trục của nó. Nếu $P_j$ đặt flag[j] là true và nó cũng đang thực thi trong câu lệnh while của nó thì turn == i hay turn == j.

Nếu turn == i thì $P_i$ sẽ đi vào miền tương trục. Nếu turn ==j thì $P_j$ sẽ đi vào miền tương trục. Tuy nhiên, một khi $P_j$ ở trong vùng tương trục của nó thì nó sẽ đặt lại flag[j] tới false, cho phép $P_i$ đi vào miền tương trục của nó. Nếu $P_j$ đặt lại flag[j] tới true, nó cũng phải đặt turn tới i. Do đó, vì $P_i$ không thay đổi giá trị của biến turn trong khi thực thi câu lệnh while, nên $P_i$ sẽ đi vào miền tương trục (tiến trình) sau khi nhiều nhất chỉ $P_j$ đi vào (chờ có giới hạn).

### III.1.2. Giải pháp nhiều tiến trình

Giải thuật 3 giải quyết vấn đề miền tương trục cho hai tiến trình. Bây giờ chúng ta phát triển một giải thuật để giải quyết vấn đề miền tương trục cho n tiến trình. Giải thuật này được gọi là giải thuật Bakery và nó dựa trên cơ sở của giải thuật định thời thường được dùng trong cửa hiệu bánh mì, cửa hàng kem,..nơi mà thứ tự rất hỗn độn. Giải thuật này được phát triển cho môi trường phân tán, nhưng tại thời điểm này chúng ta tập trung chỉ những khía cạnh của giải thuật liên quan tới môi trường tập trung.

Đi vào một cửa hàng, mỗi khách hàng nhận một số. Khách hàng với số thấp nhất được phục vụ tiếp theo. Tuy nhiên, giải thuật Bakery không thể đảm bảo hai quá trình (khách hàng) không nhận cùng số. Trong trường hợp ràng buộc, một tiến trình với tên thấp được phục vụ trước. Nghĩa là, nếu $P_i$ và $P_j$ nhận cùng một số và nếu (i < j) thì $P_i$ được phục vụ trước. Vì tên tiến trình là duy nhất và được xếp thứ tự nên giải thuật là hoàn toàn mang tính "may rủi" (deterministic). Cấu trúc dữ liệu chung là:

```
boolean  choosing[n];
int      number[n];
```

Đầu tiên, các cấu trúc dữ liệu này được khởi tạo tới false và 0 tương ứng. Để tiện dụng, chúng ta định nghĩa các ký hiệu sau:

- (a, b) < (c, d) nếu a< c hay nếu a==c và b< d
- max($a_0$,…,$a_{n-1}$) là số k ε $a_i$ với i = 0,…,n-1

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[i],…,number[n-1]) + 1;
    choosing[i] = false;
    for (j=0; j < n; j++){
        while (choosing[j]);
        while ((number[j]!=0)&&((number[ j ], j ) <(number[i], i)));
    }
    Critical section
    Number[i] = 0;
}While (1);
```

**Hình 9.5. Cấu trúc của một tiến trình Pi trong giải thuật Bakery**

Cấu trúc của tiến trình $P_i$ được dùng trong giải thuật Bakery, được hiển thị trong hình dưới đây.

Kết quả được cho này thể hiện rằng loại trừ hỗ tương được tuân theo. Thật vậy, xét $P_i$ trong vùng tương trục của nó và $P_k$ cố gắng đi vào vùng tương trục $P_k$. Khi quá trình $P_k$ thực thi câu lệnh while thứ hai cho j==i, nhận thấy rằng

- number[ i ] != 0
- (number[ i ], i ) < (number[k], k).

Do đó, nó tiếp tục vòng lặp trong câu lệnh while cho đến khi $P_i$ rời khỏi vùng tương trục $P_i$.

Giải thuật trên đảm bảo rằng yêu cầu về tiến trình, chờ đợi có giới hạn và đảm bảo sự công bằng, vì các tiến trình đi vào miền tương trục dựa trên cơ sở tới trước được phục vụ trước.

### III.1.3. Phần cứng đồng bộ hoá

Như các khía cạnh khác của phần mềm, các đặc điểm phần cứng có thể làm các tác vụ lập trình dễ hơn và cải tiến tính hiệu quả của hệ thống. Trong phần này, chúng

```
Boolean TestAndSet(boolean &target)
    {
        boolean rv = target;
                target = true;
                return rv;
    }
```

**Hình 9.6. Định nghĩa của chỉ thị TestAndSet**

ta trình bày một số chỉ thị phần cứng đơn giản sẵn dùng trên nhiều hệ thống và trình bày cách chúng được dùng hiệu quả trong việc giải quyết vấn đề miền tương trục.

Vấn đề miền tương trục có thể được giải quyết đơn giản trong môi trường chỉ có một bộ xử lý nếu chúng ta cấm các ngắt xảy ra khi một biến chia sẻ đang được thay đổi. Trong cách này, chúng ta đảm bảo rằng chuỗi chỉ thị hiện hành có thể được cho phép thực thi trong thứ tự không trưng dụng. Không có chỉ thị nào khác có thể chạy vì thế không có bất cứ sự thay đổi nào có thể được thực hiện trên các biến được chia sẻ.

Tuy nhiên, giải pháp này là không khả thi trong một môi trường có nhiều bộ xử lý. Vô hiệu hoá các ngắt trên đa bộ xử lý có thể mất nhiều thời gian khi một thông điệp muốn truyền qua tất cả bộ xử lý. Việc truyền thông điệp này bị trì hoãn khi đi vào miền tương trục và tính hiệu quả của hệ thống bị giảm.

Do đó nhiều máy cung cấp các chỉ thị phần cứng cho phép chúng ta kiểm tra hay thay đổi nội dung của một từ (word) hay để thay đổi nội dung của hai từ tuần theo tính **nguyên tử** (atomically)-như là một đơn vị không thể ngắt. Chúng ta có thể sử dụng các chỉ thị đặc biệt này để giải quyết vấn đề miền tương trục trong một cách tương đối đơn giản.

Chỉ thị TestAndSet có thể được định nghĩa như trong hình 9.6. Đặc điểm quan trọng của chỉ thị này là việc thực thi có tính nguyên tử. Do đó, nếu hai chỉ thị TestAndSet được thực thi cùng một lúc (mỗi chỉ thị trên một CPU khác nhau), thì chúng sẽ được thực thi tuần tự trong thứ tự bất kỳ.

Nếu một máy hỗ trợ chỉ thị TestAndSet thì chúng ta có thể loại trừ hỗ tương bằng cách khai báo một biến khoá kiểu lô gic và được khởi tạo tới false. Cấu trúc của tiến trình $P_i$ được hiển thị trong hình 9.7. ở trên.

Chỉ thị Swap được định như hình 9.8. dưới đây, thao tác trên nội dung của hai từ; như chỉ thị TestAndSet, nó được thực thi theo tính nguyên tử.

```
Do      {
                while (TestAndSet(lock));
                Critical section
                lock:= false
                remainder section
        } while (1);
```

**Hình 9.7.Cài đặt loại trừ hỗ tương với TestAndSet**

Nếu một máy hỗ trợ chỉ thị Swap, thì việc loại trừ hỗ tương có thể được cung cấp như sau. Một biến lô gic toàn cục lock được khai báo và được khởi tạo tới false. Ngoài ra, mỗi tiến trình cũng có một biến lôgic cục bộ key. Cấu trúc của tiến trình $P_i$ được hiển thị trong hình 9.9 dưới dây.

```
void Swap(boolean &a, boolean &b)
{
        boolean temp = a;
        a = b;
        b = temp;
}
```

**Hình 9.8. Định nghĩa chỉ thị Swap**

```
do{
        key = true; while (key == true) Swap(lock, key);
        Critical section
        lock = false;
        Remainder section
} while(1);
```

**Hình 9.9. Cài đặt loại trừ hỗ tương với chỉ thị Swap**

Các giải thuật này không thoả mãn yêu cầu chờ đợi có giới hạn. Chúng ta hiển thị giải thuật sử dụng chỉ thị TestAndSet trong hình 9.10. Giải thuật này thoả mãn tất cả các yêu cầu miền tương trục.

Cấu trúc dữ liệu thông thường là:

```
boolean    waiting[n];
boolean    lock;
```

Cấu trúc dữ liệu này được khởi tạo tới false. Để chứng minh rằng loại trừ hỗ tương được thoả, chúng ta chú ý rằng tiến trình $P_i$ có thể đưa vào miền tương trục chỉ nếu hoặc waiting[i] ==false hay key == false. Giá trị của key có thể trở thành false chỉ nếu TestAndSet được thực thi. Đối với tiến trình đầu tiên, để thực thi TestAndSet sẽ tìm key == false; tất cả tiến trình khác phải chờ. Biến waiting[i] có thể trở thành false chỉ nếu tiến trình khác rời khỏi miền tương trục của nó; chỉ một waiting[i] được đặt false, duy trì yêu cầu loại trừ hỗ tương.

```
do{
        Waiting[i] = true;
        key = true;
        while (waiting[i] && key)
                key = TestAndSet(lock);
        waiting[i] = false;
        Critical section
        j = (i + 1) % n;
        while ((j != i ) && !waiting[j])
                j = (j + 1 ) % n;
        if (j == i)
                lock = false;
        else
                waiting[j] = false;
        Remainder section
} while(1);
```

**Hình 9.10. Loại trừ hỗ tương chờ đợi có giới hạn với TestAndSet**

Để chứng minh yêu cầu tiến trình được thoả, chúng ta chú ý rằng các đối số được hiện diện cho việc loại trừ hỗ tương cũng áp dụng được ở đây, vì thế một quá trình thoát khỏi miền tương trục hoặc đặt lock bằng false hay đặt waiting[j] bằng false. Cả hai trường hợp đều cho phép một tiến trình đang chờ để đi vào miền tương trục được xử lý.

Để chứng minh yêu cầu chờ đợi được giới hạn được thoả, chúng ta chú ý rằng khi một tiến trình rời miền tương trục của nó, nó duyệt qua mảng waiting trong thứ tự tuần hoàn (i + 1, i + 2, …, n – 1, 0, …, i - 1). Nó định rõ tiến trình đầu tiên trong thứ tự này mà thứ tự đó ở trong phần đi vào (waiting[j] == true) khi tiến trình tiếp theo đi vào miền tương trục. Bất cứ tiến trình nào đang chờ để đi vào miền tương trục sẽ thực hiện n – 1 lần. Tuy nhiên, đối với người thiết kế phần cứng, cài đặt các chỉ thị nguyên tử TestAndSet trên bộ đa xử lý không là tác vụ thử nghiệm.

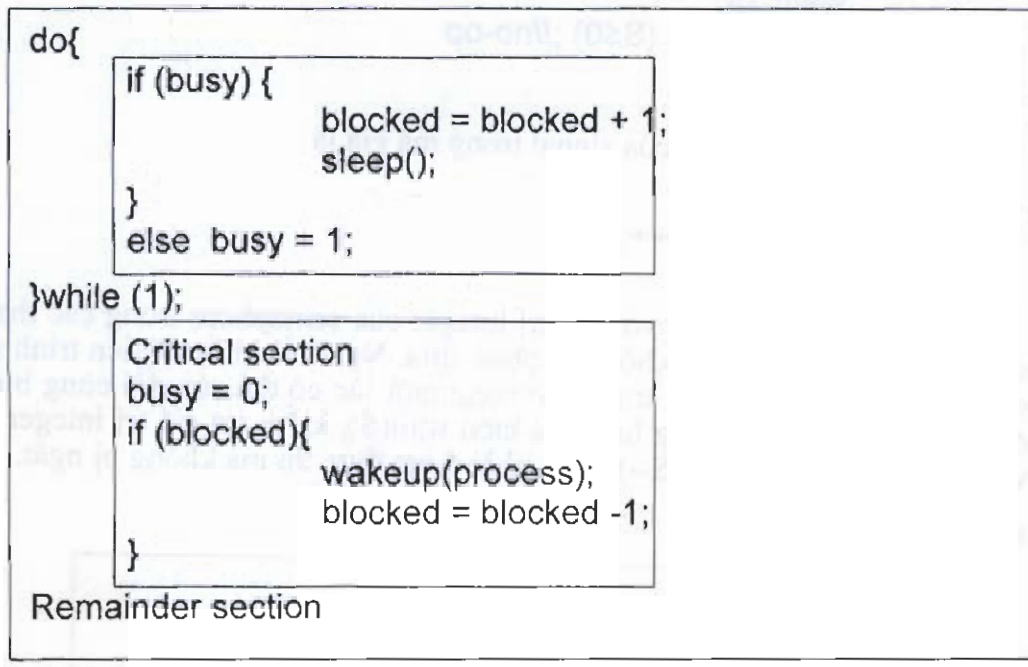
Những giải pháp trên đều phải thực hiện một vòng lặp để kiểm tra liệu nó có được phép vào miền tương trục hay không. Nếu điều kiện chưa thoả, tiến trình phải chờ tiếp tục trong vòng lặp kiểm tra này. Các giải pháp buộc tiến trình phải liên tục kiểm tra điều kiện để phát hiện thời điểm thích hợp được vào miền tương trục như thế được gọi là các ***giải pháp chờ đợi bận*** "busy waiting". Lưu ý, việc kiểm tra như thế tiêu thụ rất nhiều thời gian sử dụng CPU, do vậy tiến trình đang chờ vẫn chiếm dụng CPU. Xu hướng giải quyết vấn đề đồng bộ hoá là nên tránh các giải pháp chờ đợi bận.

### III.2. Các giải pháp "SLEEP and WAKEUP"

Để loại bỏ các bất tiện của của giải pháp chờ đợi bận, chúng ta có thể tiếp cận theo hướng cho một tiến trình chưa đủ điều kiện vào miền tương trục chuyển sang trạng thái nghẽn, từ bỏ quyền sử dụng CPU. Để thực hiện điều này, cần phải sử dụng các thủ tục do hệ điều hành cung cấp để thay đổi trạng thái tiến trình. Hai thủ tục cơ bản SLEEP và WAKEUP thường được sử dụng cho mục đích này. SLEEP là một lời gọi hệ thống có tác dụng làm "nghẽn" (blocked) hoạt động của tiến trình gọi nó và chờ

đến khi được một tiến trình khác "đánh ***thức***". Lời gọi hệ thống WAKEUP nhận một tham số duy nhất: tiến trình sẽ được kích hoạt trở lại (đặt về trạng thái sẵn sàng).

Ý tưởng sử dụng SLEEP và WAKEUP như sau: khi một tiến trình chưa đủ điều kiện vào miền tương trục, nó gọi SLEEP để tự khoá đến khi có một tiến trình khác gọi WAKEUP để giải phóng nó. Một tiến trình gọi WAKEUP khi ra khỏi miền tương trục




**Hình 9.11. Cấu trúc chương trình trong giải pháp SLEEP and WAKEUP**

để đánh thức một tiến trình đang chờ, tạo cơ hội cho tiến trình này vào miền tương trục.

```
int     busy; // 1 nếu miền tương trục đang bị chiếm
int     blocked; // đếm số lượng tiến trình đang bị khoá
```

Khi sử dụng SLEEP và WAKEUP cần hết sức cẩn thận, nếu không muốn xảy ra tình trạng mâu thuẫn truy xuất trong một vài tình huống như sau: giả sử tiến trình A vào miền tương trục, và trước khi nó rời miền tương trục thì tiến trình B được kích hoạt. Tiến trình B thử vào miền tương trục nhưng nó nhận thấy A đang ở trong đó, do vậy B tăng giá trị biến blocked lên 1 và chuẩn bị gọi SLEEP để tự nghẽn. Tuy nhiên, trước khi B có thể thực hiện SLEEP, tiến trình A được kích hoạt trở lại và ra khỏi miền tương trục. Khi ra khỏi miền tương trục, tiến trình A nhận thấy có một tiến trình đang chờ (blocked=1) nên gọi WAKEUP và giảm giá trị blocked xuống 1. Khi đó tín hiệu WAKEUP sẽ lạc mất do tiến trình B chưa thật sự "ngủ" để nhận tín hiệu đánh thức! Khi tiến trình B được tiếp tục xử lý, nó mới gọi SLEEP và tự nghẽn vĩnh viễn!

Vấn đề ghi nhận được là tình trạng lỗi này xảy ra do việc kiểm tra trạng thái miền tương trục và việc gọi SLEEP hay WAKEUP là những hành động tách biệt, có thể bị ngắt nửa chừng trong tiến trình xử lý, do đó có khi tín hiệu WAKEUP gởi đến một tiến trình chưa bị nghẽn sẽ lạc mất. Để tránh những tình huống tương tự, hệ điều hành cung cấp những cơ chế đồng bộ hoá dựa trên ý tưởng của chiến lược "SLEEP and WAKEUP" nhưng chưa được xây dựng bao gồm cả phương tiện kiểm tra điều kiện vào miền tương trục giúp sử dụng an toàn.

### III.2.1. Semaphore

Tiếp cận Semaphore được Dijkstra đề xuất vào năm 1965. Một semaphore S là một biến số nguyên (integer) được truy xuất chỉ thông qua hai thao tác nguyên tử: **wait** và **signal.** Các thao tác này được đặt tên **P** (cho wait - chờ để kiểm tra) và **V** (cho signal- báo hiệu để tăng). Định nghĩa cơ bản của wait trong mã giả là:

```
wait(S){
        while (S≤0) ;//no-op
        S--;
        }
```

Định nghĩa cơ bản của signal trong mã giả là

```
signal(S){
          S++;
        }
```

Những sửa đổi đối với giá trị integer của semaphore trong các thao tác wait và signal phải được thực thi không bị phân chia. Nghĩa là khi một tiến trình sửa đổi giá trị semaphore, không có tiến trình nào cùng một lúc có thể sửa đổi cùng biến semaphore đó. Ngoài ra, trong trường hợp của biến wait(S), kiểm tra giá trị integer của S (S δ 0) và sửa đổi có thể của nó (S--) cũng phải được thực thi mà không bị ngắt.

**a) Cách dùng**

```
do{
        wait(mutex)
        critical section
        Signal(mutex)
        remainder section
}while(1);
```

**Hình 9-13. Cài đặt loại trừ hỗ tương với semaphores**

Chúng ta có thể sử dụng semaphores để giải quyết vấn đề miền tương trục với n tiến trình. N tiến trình chia sẻ một biến semaphore, mutex (viết tắt từ mutual exclusion) được khởi tạo 1. Mỗi tiến trình $P_i$ được tổ chức như được hiển thị trong hình dưới đây.

Chúng ta cũng sử dụng semaphores để giải quyết các vấn đề đồng bộ khác nhau. Thí dụ, để xem xét hai tiến trình đang thực thi đồng hành: $P_1$ với câu lệnh $S_1$ và $P_2$ với câu lệnh $S_2$. Giả sử chúng ta yêu cầu rằng $S_2$ được thực thi chỉ sau khi $S_1$ hoàn thành. Chúng ta có thể hoàn thành cơ chế này một cách dễ dàng bằng cách $P_1$ và $P_2$ chia sẻ một semaphore chung **synch,** được khởi tạo 0 và bằng cách chèn các câu lệnh:

$S_1$;
**signal(sync);**

vào tiến trình $P_1$ và các câu lệnh:

**wait(synch);**
$S_2$;

vào trong tiến trình $P_2$. Vì synch được khởi tạo 0. $P_2$ sẽ thực thi $S_2$ chỉ sau khi $P_1$ nạp signal(synch) mà sau đó là $S_1$;

**b) Cài đặt**

Nhược điểm chính của các giải pháp loại trừ hỗ tương trong phần III.2.1 và của semaphore được cho ở đây là tất cả chúng đều đòi hỏi sự chờ đợi bận. Để giải quyết yêu cầu cho việc chờ đợi bận, chúng ta có thể hiệu chỉnh định nghĩa của các thao tác wait và signal của semaphore. Khi một tiến trình thực thi thao tác wait và nhận thấy rằng nếu giá trị của semaphore không đương, nó phải chờ. Tuy nhiên, thay vì chờ đợi bận, tiến trình có thể nghẽn chính nó. Thao tác nghẽn đặt tiến trình vào một hàng đợi gắn liền với semaphore và trạng thái tiến trình được chuyển tới trạng thái chờ. Sau đó, điều khiển được chuyển tới bộ định thời biểu và bộ định thời biểu chọn một tiến trình khác để thực thi.

Một tiến trình bị nghẽn chờ trên biến semaphore nên được khởi động lại khi tiến trình khác thực thi thao tác signal. Tiến trình được khởi động lại bởi thao tác wakeup và chuyển tiến trình từ trạng thái chờ sang trạng thái sẵn sàng. Sau đó, quá trình này được đặt vào hàng đợi sẵn sàng. (CPU có thể hay không thể được chuyển từ tiến trình đang chạy tới tiến trình sẵn sàng mới nhất phụ thuộc vào giải thuật định thời biểu CPU).

Để cài đặt semaphore dưới định nghĩa này, chúng ta định nghĩa một semaphore như một cấu trúc được viết bằng C như sau:

```
typedef struct{
        int value;
        struct process *L;
        } semaphore;
```

Mỗi semaphore có một số integer **value** và một danh sách các tiến trình **L**. Khi một tiến trình phải chờ trên một semaphore, nó được thêm vào danh sách các tiến trình L. Một thao tác signal xoá một tiến trình ra khỏi danh sách các tiến trình đang chờ và đánh thức tiến trình đó.

Thao tác semaphore wait bây giờ được định nghĩa như sau:

```
void wait(semaphore S)
    {
    S.value--;
    If (S.value < 0){
        Thêm tiến trình này tới danh sách các tiến trình S.L;
    block();
                }
    }
```

Thao tác semaphore signal bây giờ có thể được định nghĩa như sau:

```
void signal(semaphore S)
    {
        S.value++;
        if(S.value <= 0){
           xoá một tiến trình ra khỏi hàng đợi S.L;
```

```
                wakeup(P);
                }
        }
```

Thao tác **block()** tạm dừng tiến trình gọi thao tác đó. Thao tác **wakeup(P)** tiếp tục thực thi tiến trình bị nghẽn P. Hai thao tác này được cung cấp bởi hệ điều hành như những lời gọi hệ thống cơ bản.

Chú ý rằng, mặc dù dưới sự định nghĩa kinh điển của semaphores với sự chờ đợi bận là giá trị semaphore không bao giờ âm. Cài đặt này có thể có giá trị semaphore âm. Nếu giá trị semaphore âm thì tính chất trọng yếu của nó là số lượng tiến trình chờ trên semaphore đó. Sự thật này là kết quả của việc chuyển thứ tự của việc giảm và kiểm tra trong việc cài đặt thao tác **wait.** Danh sách các tiến trình đang chờ có thể được cài đặt dễ dàng bởi một trường liên kết trong mỗi khối điều khiển quá trình (PCB). Mỗi cách thêm và xoá các tiến trình từ danh sách, đảm bảo việc chờ đợi có giới hạn sẽ sử dụng hàng đợi FIFO, ở đó semaphore chứa hai con trỏ đầu (head) và đuôi (tail) chỉ tới hàng đợi. Tuy nhiên, danh sách có thể dùng bất cứ chiến lược hàng đợi nào. Sử dụng đúng semaphores không phụ thuộc vào chiến lược hàng đợi cho danh sách semaphore.

Khía cạnh quyết định của semaphores là chúng được thực thi theo tính nguyên tử. Chúng ta phải đảm bảo rằng không có hai tiến trình có thể thực thi các thao tác wait và signal trên cùng một semaphore tại cùng một thời điểm. Trường hợp này là vấn đề vùng tương trục và có thể giải quyết bằng một trong hai cách.

Trong môi trường đơn xử lý (nghĩa là chỉ có một CPU tồn tại), đơn giản là chúng ta có thể ngăn chặn các ngắt trong thời gian các thao tác wait và signal xảy ra. Cơ chế này làm việc trong một môi trường đơn xử lý vì một khi ngắt bị ngăn chặn, các chỉ thị từ các tiến trình khác không thể được chen vào. Chỉ tiến trình đang chạy hiện tại thực thi cho tới khi các ngắt được cho phép sử dụng trở lại và bộ định thời có thể thu hồi quyền điều khiển.

Trong môi trường đa xử lý, ngăn chặn ngắt không thể thực hiện được. Các chỉ thị từ các tiến trình khác nhau (chạy trên các bộ xử lý khác nhau) có thể được chen vào trong cách bất kỳ. Nếu phần cứng không cung cấp bất cứ các chỉ thị đặc biệt nào, chúng ta có thể tận dụng các giải pháp phần cứng phù hợp cho vấn đề vùng tương trục (phần III.4), ở đó các vùng tương trục chứa cả thủ tục wait và signal.

Vấn đề quan trọng là chúng ta không xoá hoàn toàn chờ đợi bận, với định nghĩa này cho các thao tác wait và signal. Dĩ nhiên là chúng ta xoá chờ đợi bận từ việc đi vào vùng tương trục của chương trình ứng dụng. Ngoài ra, chúng ta hạn chế việc chờ đợi bận chỉ các miền tương trục với thao tác wait và signal và các vùng này là ngắn (nếu được mã hợp lý, chúng nên không quá 10 chỉ thị). Do đó, miền tương trục hầu như không bao giờ bị chiếm và sự chờ đợi bận rất hiếm khi xảy ra và sau đó chỉ cho thời gian ngắn. Một trường hợp hoàn toàn khác xảy ra với những chương trình ứng dụng có miền tương trục dài (vài phút hay thậm chí vài giờ) hay có thể hầu như luôn bị chiếm. Trong trường hợp này, chờ đợi bận là cực kỳ kém hiệu quả.

### c) Sự khoá chết (deadlocks) và đói tài nguyên

Cài đặt semaphore với một hàng đợi có thể dẫn đến trường hợp hai hay nhiều

tiến trình đang chờ không hạn định một sự kiện mà có thể được gây ra chỉ bởi một trong những tiến trình đang chờ. Sự kiện đặt ra là sự thực thi của thao tác signal. Khi một trạng thái như thế xảy ra, những tiến trình này được nói là bị khoá chết.

Để hiển thị điều này, chúng ta xét một hệ thống chứa hai tiến trình $P_0$ và $P_1$, mỗi truy xuất hai semaphore, S và Q, được đặt giá trị 1.

```
P0              P1
wait(S);        wait(Q);
wait(Q);        wait(S);
.               .
.               .
signal(S);      signal(Q);
Signal(Q);      signal(S);
```

Giả sử rằng $P_0$ thực thi wait(S) và sau đó $P_1$ thực thi wait(Q). Khi $P_0$ thực thi wait(Q), nó phải chờ cho đến khi $P_1$ thực thi signal(Q). Tương tự, khi $P_1$ thực thi wait(S), nó phải chờ cho tới khi $P_0$ thực thi signal(S). Vì các thao tác signal này không thể được thực thi nên $P_0$ và $P_1$ bị khoá chết.

Chúng ta nói rằng một tập hợp các tiến trình trong trạng thái khoá chết khi mọi tiến trình trong tập hợp đang chờ một sự kiện được gây ra chỉ bởi một tiến trình khác trong tập hợp. Những sự kiện mà chúng ta quan tâm chủ yếu ở đây là việc chiếm tài nguyên và giải phóng tài nguyên. Tuy nhiên, các loại sự kiện khác cũng có thể dẫn đến việc khoá chết. Chúng ta sẽ xem trong chương 10. Trong chương đó, chúng ta sẽ mô tả các cơ chế khác nhau để giải quyết vấn đề khoá chết.

Một vấn đề khoá chết liên quan tới khoá chết là **nghẽn hay đói tài nguyên không hạn định** (indefinite blocking or starvation), ở đó các tiến trình chờ đợi không hạn định trong semaphore. Nghẽn không hạn định có thể xảy ra nếu chúng ta thêm vào và lấy ra các tiến trình từ danh sách được nối kết với một semaphore trong thứ tự vào sau ra trước (LIFO).

**d) Semaphore nhị phân**

Xây dựng semaphore được mô tả trong phần trước được gọi là **semaphore đếm** (counting semaphore) vì giá trị nguyên có thể trải dài một phạm vi không giới hạn. Một **semaphore nhị phân** (binary semaphore) là một semaphore với một giá trị nguyên mà trải dài từ 0 và 1. Semaphore nhị phân có thể đơn giản hơn trong cài đặt so với semaphore đếm và phụ thuộc vào kiến trúc phần cứng nằm bên dưới. Chúng sẽ hiển thị cách một semaphore đếm có thể được cài đặt sử dụng semaphore nhị phân dưới đây:

Giả sử S là một semaphore đếm. Để cài đặt nó trong dạng semaphore nhị phân chúng ta cần các cấu trúc dữ liệu như sau:

```
Binary-semaphore S1, S2;
int C;
```

Khởi tạo S1 = 1, S2 = 0 và giá trị nguyên C được đặt tới giá trị khởi tạo của semaphore đếm S. Thao tác wait trên semaphore đếm S có thể được cài đặt như sau:

```
wait(S);
C--;
```

```
If (C<0) {
            signal(S1);
            wait(S2);
        }
signal(S1);
```

Thao tác signal trên semaphore đếm S có thể được cài đặt như sau:

```
wait(S1);
C++;
if (C<=0)
        signal(S2);
else
        signal(S1);
```

### III.2.2. Monitors

Để có thể dễ viết đúng các chương trình đồng bộ hoá hơn, Hoare (1974) và Brinch & Hansen (1975) đề nghị một cơ chế đồng bộ hoá cấp cao hơn được cung cấp bởi ngôn ngữ lập trình là **monitor.** Một monitor được mô tả bởi một tập hợp của các toán tử được định nghĩa bởi người lập trình. Biểu diễn kiểu của một monitor bao gồm việc khai báo các biến mà giá trị của nó xác định trạng thái của một thể hiện kiểu, cũng như thân của thủ tục hay hàm mà cài đặt các thao tác trên kiểu đó. Cú pháp của monitor được hiển thị trong hình dưới đây:

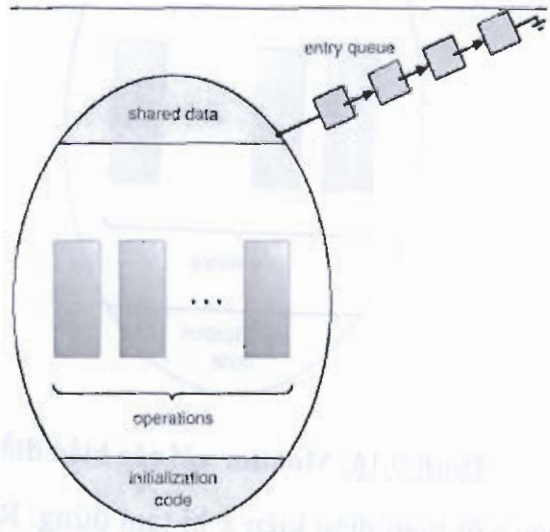
```
Monitor <tên monitor>
        {
                khai báo các biến được chia sẻ
                procedure P1 (...){
        ...
        }

        procedure P2 (...){
        ...
        }
        .
        .
        .
        procedure Pn (...){
        ...
        }
        {
        mã khởi tạo
        }
```

}

**Hình 9.14 Cú pháp của monitor**

Biểu diễn kiểu monitor không thể được dùng trực tiếp bởi các tiến trình khác nhau. Do đó, một thủ tục được định nghĩa bên trong một monitor chỉ có thể truy xuất những biến được khai báo cục bộ bên trong monitor đó và các tham số chính thức của




**Hình 9.15. Hình ảnh dưới dạng biểu đồ của monitor**

nó. Tương tự, những biến cục bộ của monitor có thể được truy xuất chỉ bởi những thủ tục cục bộ.

Xây dựng monitor đảm bảo rằng chỉ một tiến trình tại một thời điểm có thể được kích hoạt trong monitor. Do đó, người lập trình không cần viết mã ràng buộc đồng bộ hoá như hình 9.15 dưới đây:

Tuy nhiên, xây dựng monitor như được định nghĩa là không đủ mạnh để mô hình hoá các cơ chế đồng bộ. Cho mục đích này, chúng ta cần định nghĩa các cơ chế đồng bộ hoá bổ sung. Những cơ chế này được cung cấp bởi construct ***condition.*** Người lập trình có thể định nghĩa một hay nhiều biến của kiểu ***condition:***

```
condition x, y;
```

Chỉ những thao tác có thể gọi lên trên các biến điều kiện là wait và signal. Thao tác

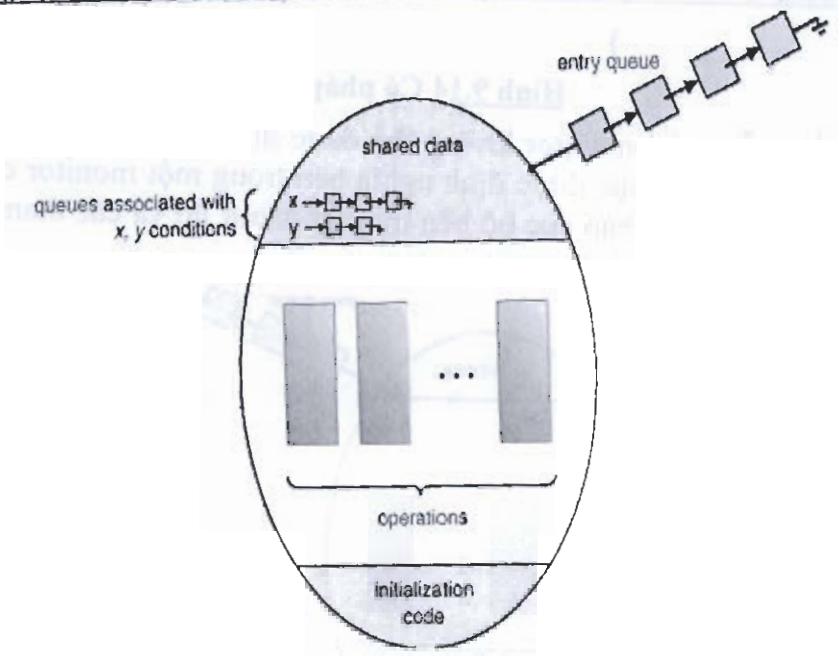
```
x.wait();
```

có nghĩa là tiến trình gọi trên thao tác này được tạm dừng cho đến khi tiến trình khác gọi

```
x.signal();
```

thao tác x.signal() thực thi tiếp một cách chính xác một tiến trình tạm dừng. Nếu không có tiến trình tạm dừng thì thao tác signal không bị ảnh hưởng gì cả; nghĩa là trạng thái x như thể thao tác chưa bao giờ được thực thi (như hình 9.16). Ngược lại, với thao tác signal được gán cùng với semaphores luôn ảnh hưởng tới trạng thái của semaphore.

Bây giờ giả sử rằng, khi thao tác x.signal() được gọi bởi một tiến trình P thì có




**Hình 9.16. Monitor với các biến điều kiện**

một tiến trình Q gán với biến diều kiện x bị tạm dừng. Rõ ràng, nếu tiến trình Q được phép thực thi tiếp thì tiến trình P phải dừng. Nếu không thì cả hai tiến trình P và Q hoạt động cùng một lúc trong monitor. Tuy nhiên, về khái niệm hai tiến trình có thể tiếp tục việc thực thi của chúng. Hai khả năng có thể xảy ra: P chờ cho đến khi Q rời khỏi monitor hoặc chờ diều kiện khác.

Q chờ cho đến khi P rời monitor hoặc chờ diều kiện khác.

Có các luận cứ hợp lý trong việc chấp nhận khả năng I hay 2. Vì P đã thực thi trong monitor rồi, nên chọn khả năng 2 có vẻ hợp lý hơn. Tuy nhiên, nếu chúng ta cho phép tiến trình P tiếp tục, biến điều kiện "lô gic" mà Q đang chờ có thể không còn quản lý thời gian Q được tiếp tục. Chọn khả năng 1 được tán thành bởi Hoare vì tham số đầu tiên của nó chuyển trực tiếp tới các qui tắc chứng minh đơn giản hơn. Thoả hiệp giữa hai khả năng này được chấp nhận trong ngôn ngữ đồng hành C. Khi quá trình P thực thi thao tác signal thì tiến trình Q lập tức được tiếp tục. Mô hình này không mạnh hơn mô hình của Hoare vì một tiến trình không thể báo hiệu nhiều lần trong một lời gọi thủ tục đơn.

Bây giờ chúng ta xem xét cài đặt cơ chế monitor dùng semaphores. Đối với mỗi monitor, một biến semaphore ***mutex*** (được khởi tạo 1) được cung cấp. Một quá trình phải thực thi ***wait(mutex)*** trước khi đi vào monitor và phải thực thi ***signal(mutex)*** sau khi rời monitor.

Vì tiến trình đang báo hiệu phải chờ cho đến khi tiến trình được bắt đầu lại rời hay chờ, một biến semaphore bổ sung ***next*** được giới thiệu, được khởi tạo 0 trên quá trình báo hiệu có thể tự tạm dừng. Một biến số nguyên ***next_count*** cũng sẽ được cung cấp để đếm số lượng tiến trình bị tạm dừng trên ***next***. Do đó, mỗi thủ tục bên ngoài F sẽ được thay thế bởi

```
wait(mutex);
...
```

```
thân của F
if (next_count > 0)
        signal(next);
else
        signal(mutex);
```

Loại trừ hỗ tương trong monitor được đảm bảo. Bây giờ chúng ta mô tả các biến điều kiện được cài đặt như thế nào. Đối với mỗi biến điều kiện ***x,*** chúng ta giới thiệu một biến semaphore ***x_sem*** và biến số nguyên ***x_count,*** cả hai được khởi tạo tới 0. Thao tác ***x.wait*** có thể được cài đặt như sau:

```
x_count++;
if ( next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x_count--;
```

Thao tác ***x.signal()*** có thể được cài đặt như sau:

```
if ( x_count > 0){
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
}
```

Cài đặt này có thể áp dụng để định nghĩa của monitor bởi cả hai Hoare và Brinch Hansen. Tuy nhiên, trong một số trường hợp tính tổng quát của việc cài đặt là không cần thiết và yêu cầu có một cải tiến hiệu quả hơn.

Bây giờ chúng ta sẽ trở lại chủ đề thứ tự bắt đầu lại của tiến trình trong monitor. Nếu nhiều tiến trình bị trì hoãn trên biến điều kiện ***x*** và thao tác ***x.signal*** được thực thi bởi một vài tiến trình thì thứ tự các tiến trình bị trì hoãn được thực thi trở lại như thế nào? Một giải pháp đơn giản là dùng thứ tự FCFS vì thế tiến trình chờ lâu nhất sẽ được thực thi tiếp trước. Tuy nhiên, trong nhiều trường hợp, cơ chế định thời biểu như thế là không đủ. Cho mục đích này cấu trúc ***conditional-wait*** có thể được dùng: nó có dạng

```
x.wait(c);
```

ở đây ***c*** là một biểu thức số nguyên được định giá khi thao tác ***wait*** được thực thi. Giá trị ***c,*** được gọi là ***số ưu tiên,*** được lưu với tên tiến trình được tạm dừng. Khi ***x.signal*** được thực thi, tiến trình với số ưu tiên nhỏ nhất được thực thi tiếp.

Để hiển thị cơ chế mới này, chúng ta xem xét monitor được hiển thị như hình dưới đây, điều khiển việc cấp phát của một tài nguyên đơn giữa các tiến trình cạnh tranh. Mỗi tiến trình khi yêu cầu cấp phát tài nguyên của nó, xác định thời gian tối đa nó hoạch định để sử dụng tài nguyên. Monitor cấp phát tài nguyên tới tiến trình có yêu cầu thời gian cấp phát ngắn nhất.

```
Monitor ResourceAllocation
    {
      Boolean      busy;
      Condition    x;
      void acquire (int time){
      if (busy)       x.wait(time);
                      busy = true;
          }
          void release(){
                      busy = false;
                      x.signal();
          }
          void init(){
                      busy = false;
          }
    }
```

**Hình 9.17. Một monitor cấp phát tới một tài nguyên**

Một tiến trình cần truy xuất tài nguyên phải chú ý thứ tự sau:

```
R.acquire(t);
...
        truy xuất tài nguyên
...
R.release();
```

ở đây ***R*** là thể hiện của kiểu ***ResourceAllocation.*** Tuy nhiên, khái niệm monitor không đảm bảo rằng các thứ tự truy xuất trước sẽ được chú ý. Đặc biệt,

• Một tiến trình có thể truy xuất tài nguyên mà không đạt được quyền truy xuất trước đó.

• Một tiến trình sẽ không bao giờ giải phóng tài nguyên một khi nó được gán truy xuất tới tài nguyên đó.

• Một tiến trình có thể cố gắng giải phóng tài nguyên mà nó không bao giờ yêu cầu.

• Một tiến trình có thể yêu cầu cùng tài nguyên hai lần (không giải phóng tài nguyên đó trong lần đầu)

Việc sử dụng monitor cũng gặp cùng những khó khăn như xây dựng miền tương trục. Trong phần trước, chúng ta lo lắng về việc sử dụng đúng semaphore. Bây giờ, chúng ta lo lắng về việc sử dụng đúng các thao tác được định nghĩa của người lập trình cấp cao mà các trình biên dịch không còn hỗ trợ chúng ta.

Một giải pháp có thể đối với vấn đề trên là chứa các thao tác truy xuất tài nguyên trong monitor ***ResourceAllocation.*** Tuy nhiên, giải pháp này sẽ dẫn đến việc định thời được thực hiện dựa theo giải thuật định thời monitor được xây dựng sẵn hơn là được viết bởi người lập trình.

Để đảm bảo rằng các tiến trình chú ý đến thứ tự hợp lý, chúng ta phải xem xét kỹ tất cả chương trình thực hiện việc dùng monitor ResourceAllocation và những tài

nguyên được quản lý của chúng. Có hai điều kiện mà chúng ta phải kiểm tra để thiết lập tính đúng đắn của hệ thống. Đầu tiên, các tiến trình người dùng phải luôn luôn thực hiện các lời gọi của chúng trên monitor trong thứ tự đúng. Thứ hai, chúng ta phải đảm bảo rằng một tiến trình không hợp tác không đơn giản bỏ qua cổng (gateway) loại trừ hỗ tương được cung cấp bởi monitor và cố gắng truy xuất trực tiếp tài nguyên được chia sẻ mà không sử dụng giao thức truy xuất. Chỉ nếu hai điều kiện này có thể được đảm bảo có thể chúng ta đảm bảo rằng không có lỗi ràng buộc thời gian nào xảy ra và giải thuật định thời sẽ không bị thất bại.

Mặc dù việc xem xét này có thể cho hệ thống nhỏ, tĩnh nhưng nó không phù hợp cho một hệ thống lớn hay động. Vấn đề kiểm soát truy xuất có thể được giải quyết chỉ bởi một cơ chế bổ sung khác.

## I.V. Các bài toán đồng bộ hoá nguyên thuỷ

Trong phần này, chúng ta trình bày một số bài toán đồng bộ hoá như những thí dụ về sự phân cấp lớn các vấn đề điều khiển đồng hành. Các vấn đề này được dùng cho việc kiểm tra mọi cơ chế đồng bộ hoá được đề nghị gần đây. Semaphore được dùng cho việc đồng bộ hoá trong các giải pháp dưới đây.

### IV.1 Bài toán người sản xuất-người tiêu thụ

Bài toán người sản xuất-người tiêu thụ (Producer-Consumer) thường được dùng để hiển thị sức mạnh của các hàm cơ sở đồng bộ hoá. Hai tiến trình cùng chia sẻ một vùng đệm có kích thước giới hạn n. Biến semaphore ***mutex*** cung cấp sự loại trừ hỗ tương để truy xuất vùng đệm và được khởi tạo với giá trị ***1.*** Các biến semaphore ***empty*** và ***full*** đếm số khe trống và đầy tương ứng. Biến semaphore ***empty*** được khởi tạo tới giá trị ***n;*** biến semaphore ***full*** được khởi tạo tới giá trị ***0.***

Mã cho người tiến trình sản xuất được hiển thị trong hình 9.18:

```
do{
        ...
        sản xuất sản phẩm trong nextp
        ...
        wait(empty);
        wait(mutex);
        ...
        thêm nextp tới vùng đệm
        ...
        signal(mutex);
        signal(full);
} while (1);
```

**Hình 9.18 Cấu trúc của tiến trình người sản xuất**

Mã cho tiến trình người tiêu thụ được hiển thị trong hình dưới đây:

```
do{
        wait(full);
        wait(mutex);
        ...
        lấy một sản phẩm từ vùng đệm tới nextc
        ...
```

```
            signal(mutex);
            signal(empty);
        } while (1);
```

**Hình 9.19. Cấu trúc của tiến trình người tiêu thụ**

### IV.2. Bài toán bộ đọc-bộ ghi

Bộ đọc-bộ ghi (Readers-Writers) là một đối tượng dữ liệu (như một tập tin hay mẫu tin) được chia sẻ giữa nhiều tiến trình đồng hành. Một số trong các tiến trình có thể chỉ cần đọc nội dung của đối tượng được chia sẻ, ngược lại một vài tiến trình khác cần cập nhật (nghĩa là đọc và ghi ) trên đối tượng được chia sẻ. Chúng ta phân biệt sự khác nhau giữa hai loại tiến trình này bằng cách gọi các tiến trình chỉ đọc là bộ đọc và các tiến trình cần cập nhật là bộ ghi. Chú ý, nếu hai bộ đọc truy xuất đối tượng được chia sẻ cùng một lúc sẽ không có ảnh hưởng gì. Tuy nhiên, nếu một bộ ghi và vài quá trình khác (có thể là bộ đọc hay bộ ghi) truy xuất cùng một lúc có thể dẫn đến sự hỗn độn.

Để đảm bảo những khó khăn này không phát sinh, chúng ta yêu cầu các bộ ghi có truy xuất loại trừ lẫn nhau tới đối tượng chia sẻ. Việc đồng bộ hoá này được gọi là bài toán bộ đọc-bộ ghi. Bài toán bộ đọc-bộ ghi có một số biến dạng liên quan đến độ ưu tiên. Dạng đơn giản nhất là bài toán ***bộ đọc trước-bộ ghi*** (first reader-writer). Trong dạng này yêu cầu không có bộ đọc nào phải chờ ngoại trừ có một bộ ghi đã được cấp quyền sử dụng đối tượng chia sẻ. Nói cách khác, không có bộ đọc nào phải chờ các bộ đọc khác để hoàn thành đơn giản vì một bộ ghi đang chờ. Bài toán ***bộ đọc sau-bộ ghi*** (second readers-writers) yêu cầu một khi bộ ghi đang sẵn sàng, bộ ghi đó thực hiện việc ghi của nó sớm nhất có thể. Nói một cách khác, nếu bộ ghi đang chờ truy xuất đối tượng, không có bộ đọc nào có thể bắt đầu việc đọc.

Giải pháp cho bài toán này có thể dẫn đến việc đói tài nguyên. Trong trường hợp đầu, các bộ ghi có thể bị đói; trong trường hợp thứ hai các bộ đọc có thể bị đói. Trong giải pháp cho bài toán bộ đọc trước-bộ ghi, các tiến trình bộ đọc chia sẻ các cấu trúc dữ liệu sau:

```
semaphore mutex, wrt;
int readcount;
```

Biến semaphore ***mutex*** và ***wrt*** được khởi tạo **1;** biến ***readcount*** được khởi tạo **0.** Biến semaphore ***wrt*** dùng chung cho cả hai tiến trình bộ đọc và bộ ghi. Biến semaphore ***mutex*** được dùng để đảm bảo loại trừ hỗ tương khi biến ***readcount*** được cập nhật. Biến ***readcount*** ghi vết có bao nhiêu tiến trình hiện hành đang đọc đối tượng. Biến semaphore ***wrt*** thực hiện chức năng như một biến semaphore loại trừ hỗ tương cho các bộ đọc. Nó cũng được dùng bởi bộ đọc đầu tiên hay bộ đọc cuối cùng mà nó đi vào hay thoát khỏi miền tương trục. Nó cũng không được dùng bởi các bộ đọc mà nó đi vào hay thoát trong khi các bộ đọc khác đang ở trong miền tương trục. Mã cho tiến trình bộ viết được hiển thị như hình 9.20:

```
wait(wrt);
...
Thao tác viết được thực hiện
signal(wrt);
```

**Hình 9.20. Cấu trúc của tiến trình viết**

Mã của tiến trình dọc được hiển thị như hình 9.21:

```
wait(mutex);
readcount++;
if (readcount == 1)
        wait(wrt);
signal(mutex);
...
Thao tác đọc được thực hiện
wait(mutex);
readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex);
```

**Hình 9.21 Cấu trúc của bộ đọc**

Chú ý rằng, nếu bộ viết đang ở trong miền tương trục và n bộ đọc đang chờ thì một bộ đọc được xếp hàng trên wrt, và n-1 được xếp hàng trên mutex. Cũng cần chú ý thêm, khi một bộ viết thực thi signal(wrt) thì chúng ta có thể thực thi tiếp việc thực thi

**Hình 9.22. Tình huống các triết gia ăn tối**

của các tiến trình đọc đang chờ hay một tiến trình viết đang chờ. Việc chọn lựa này có thể được thực hiện bởi bộ định thời biểu.

### IV.3. Bài toán các triết gia ăn tối

Có năm nhà triết gia, vừa suy nghĩ vừa ăn tối. Các triết gia ngồi trên cùng một bàn tròn xung quanh có năm chiếc ghế, mỗi chiếc ghế được ngồi bởi một triết gia. Chính giữa bàn là một bát cơm và năm chiếc đũa được hiển thị như hình 9.22:

Khi một triết gia suy nghĩ, ông ta không giao tiếp với các triết gia khác. Thỉnh

thoảng, một triết gia cảm thấy đói và cố gắng chọn hai chiếc đũa gần nhất (hai chiếc đũa nằm giữa ông ta với hai láng giềng trái và phải). Một triết gia có thể lấy chỉ một chiếc đũa tại một thời điểm. Chú ý, ông ta không thể lấy chiếc đũa mà nó đang được dùng bởi người láng giềng. Khi một triết gia đói và có hai chiếc đũa cùng một lúc, ông ta ăn mà không đặt đũa xuống. Khi triết gia ăn xong, ông ta đặt đũa xuống và bắt đầu suy nghĩ tiếp.

Bài toán các triết gia ăn tối được xem như một bài toán đồng bộ hoá kinh điển. Nó trình bày yêu cầu cấp phát nhiều tài nguyên giữa các tiến trình trong cách tránh việc khoá chết và đói tài nguyên. Một giải pháp đơn giản là thể hiện mỗi chiếc đũa bởi một biến semaphore.

Một triết gia cố gắng chiếm lấy một chiếc đũa bằng cách thực thi thao tác wait trên biến semaphore đó; triết gia đặt hai chiếc đũa xuống bằng cách thực thi thao tác signal trên các biến semaphore tương ứng. Do đó, dữ liệu được chia sẻ là:

```
semaphore chopstick[5];
```

ở đây tất cả các phần tử của chopstick được khởi tạo 1. Cấu trúc của philosopher i được hiển thị như hình dưới đây:

```
do{
        wait(chopstick[ i ]);
        wait(chopstick[ ( i + 1 ) % 5 ]);
        …
        ăn
        …
        signal(chopstick[ i ]);
        signal(chopstick[ ( i + 1 ) % 5 ]);
        …
        suy nghĩ
        …
} while (1);
```

**Hình 9.23 Cấu trúc của triết gia thứ i**

Mặc dù giải pháp này đảm bảo rằng không có hai láng giềng nào đang ăn cùng một lúc nhưng nó có khả năng gây ra khoá chết. Giả sử rằng năm triết gia bị đói cùng một lúc và mỗi triết gia chiếm lấy chiếc đũa bên trái của ông ta. Bây giờ tất cả các phần tử chopstick sẽ là 0. Khi mỗi triết gia cố gắng dành lấy chiếc đũa bên phải, triết gia sẽ bị chờ mãi mãi.

Nhiều giải pháp khả thi đối với vấn đề khoá chết được liệt kê tiếp theo. Giải pháp cho vấn đề các triết gia ăn tối mà nó đảm bảo không bị khoá chết.

- Cho phép nhiều nhất bốn triết gia đang ngồi cùng một lúc trên bàn
- Cho phép một triết gia lấy chiếc đũa của ông ta chỉ nếu cả hai chiếc đũa là sẵn dùng (để làm điều này ông ta phải lấy chúng trong miền tương trục).

• Dùng một giải pháp bất đối xứng; nghĩa là một triết gia lẽ chọn dũa bên trái đầu tiên của ông ta và sau đó dũa bên phải, trái lại một triết gia chẵn chọn chiếc dũa bên phải và sau đó chiếc dũa bên phải của ông ta.

Tóm lại, bất cứ một giải pháp nào thoả mãn đối với bài toán các triết gia ăn tối phải đảm bảo dựa trên khả năng một trong những triết gia sẽ đói chết. Giải pháp giải quyết việc khoá chết không cần thiết xoá đi khả năng đói tài nguyên.

## V. Tóm tắt

Một tập hợp các tiến trình tuần tự cộng tác chia sẻ dữ liệu, loại trừ hỗ tương phải được cung cấp. Một giải pháp đảm bảo rằng vùng tương trục của mã dang sử dụng chỉ bởi một tiến trình hay một luồng tại một thời điểm. Các giải thuật khác tồn tại để giải quyết vấn đề miền tương trục, với giả thuyết rằng chỉ khoá bên trong việc lưu trữ là sẵn dùng.

Sự bất lợi chủ yếu của các giải pháp được mã hoá bởi người dùng là tất cả chúng đều yêu cầu sự chờ đợi bận. Semaphore khắc phục sự bất lợi này. Semaphores có thể được dùng để giải quyết các vấn đề đồng bộ khác nhau và có thể được cài đặt hiệu quả, đặc biệt nếu phần cứng hỗ trợ các thao tác nguyên tử.

Các bài toán đồng bộ khác (chẳng hạn như bài toán người sản xuất-người tiêu dùng, bài toán bộ đọc, bộ ghi và bài toán các triết gia ăn tối) là cực kỳ quan trọng vì chúng là thí dụ của phân lớp lớn các vấn đề diều khiển đồng hành. Vấn đề này được dùng để kiểm tra gần như mọi cơ chế đồng bộ được đề nghị gần đây.

Hệ điều hành phải cung cấp phương tiện để đảm bảo chống lại lỗi thời gian. Nhiều cấu trúc dữ liệu được đề nghị để giải quyết các vấn đề này. Các vùng tương trục có thể được dùng để cài đặt loại trừ hỗ tương và các vấn đề đồng bộ toàn và hiệu quả. Monitors cung cấp cơ chế đồng bộ cho việc chia sẻ các loại dữ liệu trừu tượng. Một biến diều kiện cung cấp một phương thức cho một thủ tục monitor khoá việc thực thi của nó cho đến khi nó được báo hiệu tiếp tục.