

Chương 3

ĐỒNG BỘ TIỀN TRÌNH

(Process Synchronization)

Nội Dung Chương 3

- I. Nhu cầu đồng bộ hóa tiến trình
- II. Giải pháp đồng bộ hóa busy waiting
- III. Giải pháp đồng bộ hóa sleep&wakeup
- IV. Một số bài toán đồng bộ
- V. Tắc nghẽn - Deadlock

I. Nhu cầu đồng bộ tiến trình

Trong hệ thống, các tiến trình có nhu cầu liên lạc với nhau để:

- Chia sẻ thông tin
- Phối hợp thực hiện công việc

Mục tiêu đồng bộ

- Đảm bảo độc quyền truy xuất
- Đảm bảo cơ chế phối hợp giữa các tiến trình.

Bài toán 1

Hai tiến trình P_1 và P_2 cùng truy xuất dữ liệu chung là một tài khoản ngân hàng:

```
if (So_du > Tien_rut)
    So_du = So_du - Tien_rut
else
    Access denied!
```

1. Vấn đề tranh đoạt điều khiển (race condition)

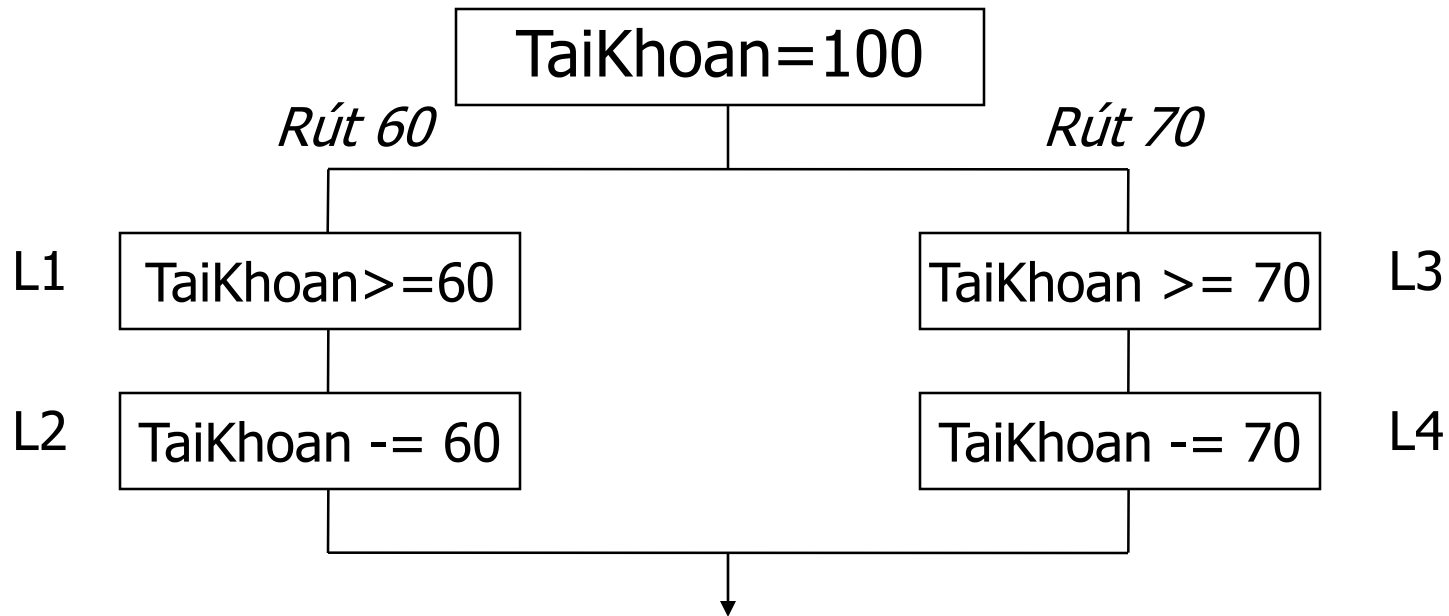
- a) Ví dụ: Rút tiền tài khoản, chỉ được rút khi số tiền rút nhỏ hơn tiền trong tài khoản.

```
static int TaiKhoan;
static void RutTien1()
{
    if (TaiKhoan >= 60)
        TaiKhoan = TaiKhoan - 60;
}

static void RutTien2()
{
    if (TaiKhoan >= 70)
        TaiKhoan = TaiKhoan - 70;
}

static void Main(string[] args)
{
    TaiKhoan = 100;
    Thread t1 = new Thread(new ThreadStart(RutTien1));
    Thread t2 = new Thread(new ThreadStart(RutTien2));
    t1.Start();
    t2.Start();
}
```

Câu hỏi: Với tài khoản là 100, sau khi thực hiện xong hai tiểu trình trên giá trị TaiKhoan là bao nhiêu?



Kịch bản 1: Thứ tự thực hiện

L1 (true)
 L2 (taikhoan=40)
 L3 (false)
 L4 (không thực

hiện)

→ Tài khoản còn 40

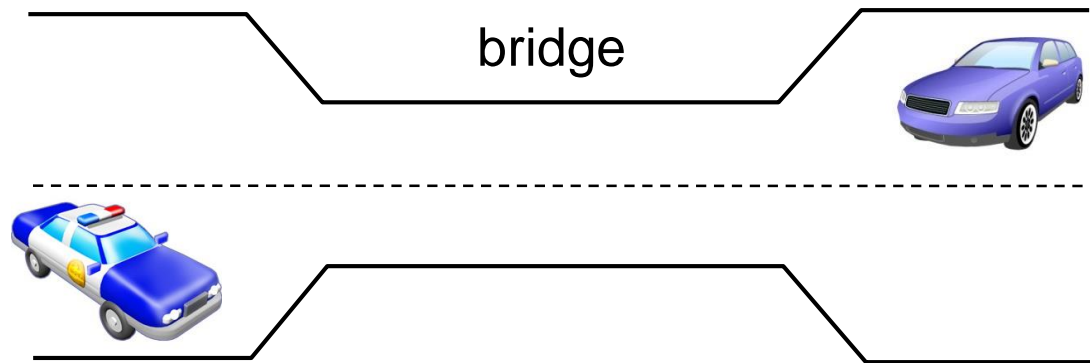
Kịch bản 2: Thứ tự thực hiện

L1 (true)
 L3 (true)
 L2 (taikhoan=40)
 L4 (taikhoan=-30)

→ Tài khoản còn -30

→ Lỗi là do chen ngang L3 giữa L1 và L2

- b) Định nghĩa tranh đoạt điều khiển: *Hai hay nhiều tiến trình truy xuất đồng thời một tài nguyên nào đó dẫn đến kết quả không mong muốn.*



Khái niệm độc quyền truy xuất (mutual exclusion): tại một thời điểm chỉ có duy nhất một tiến trình được quyền truy xuất trên một tài nguyên nào đó.

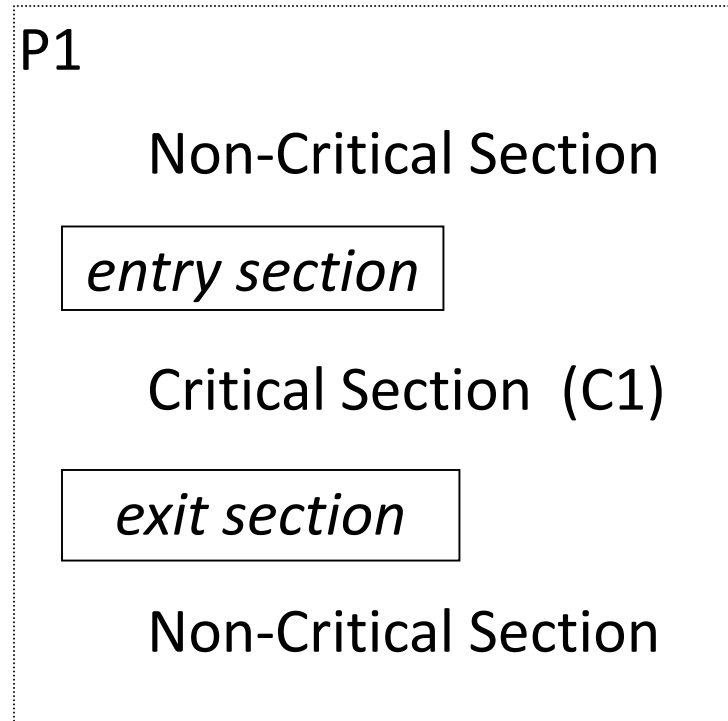
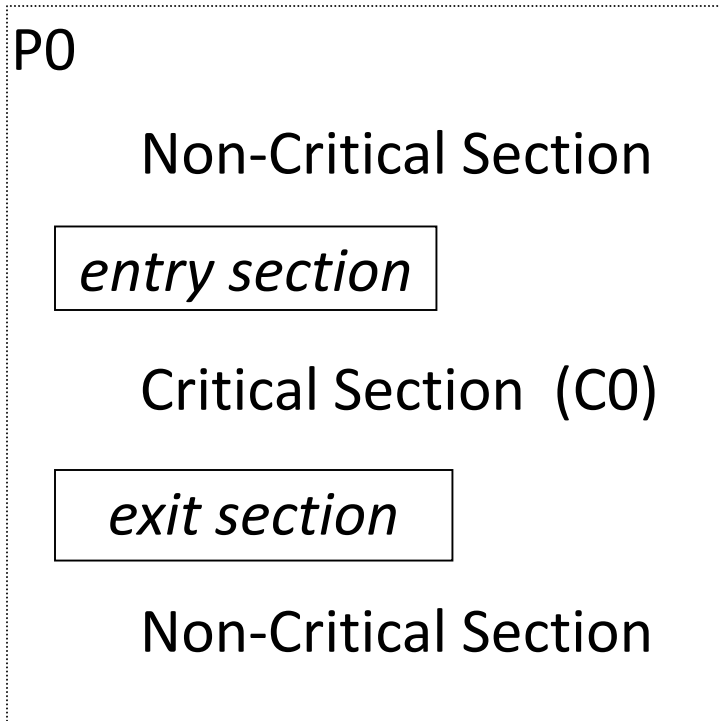
Câu hỏi:

- 1) Trong ví dụ rút tiền, cái gì là tài nguyên dùng chung?
- 2) Làm thế nào để thực hiện độc quyền truy xuất ? →
khái niệm miền găng

2. Miền găng (Critical Section)

a) Định nghĩa:

- Là một đoạn lệnh đặc biệt của tiến trình.
- Khi một miền găng đang thực hiện, các miền găng của các tiến trình khác không được phép thực hiện dù được cấp CPU.



b) Mục đích miền găng :

Đặt các lệnh truy xuất đến tài nguyên dùng chung vào các miền găng thì sẽ không xảy ra vấn đề tranh đoạt điều khiển.

Ví dụ vấn đề rút tiền: Khi đặt L1 và L2 vào miền găng C0, L3 hay L4 vào miền găng C1 thì L3 và L4 không thể chen ngang giữa L1 và L2

3. Khái niệm đồng bộ hóa tiến trình:

Việc sử dụng **miền găng** cho tiến trình hay tiểu trình nhằm đảm bảo **độc quyền truy xuất** được gọi là **đồng bộ hóa tiến trình**.

Các giải pháp đồng bộ

- Nhóm giải pháp “busy and waiting”
 - Giải pháp phần mềm
 - Giải pháp phần cứng
- Nhóm giải pháp “Sleep and Wakeup”
 - Semaphore
 - Monitor
 - Trao đổi bản tin

II. Giải pháp đồng bộ ‘Busy Waiting’

1. Giải pháp dùng cho hai tiến trình

a) Giải pháp luân phiên:

hai tiến trình P0, P1 sẽ ***lần lượt tuần tự*** sử dụng miền găng.

Sử dụng biến turn:

- turn= 0 : lượt vào miền găng của P0
- turn= 1 : lượt vào miền găng của P1

int turn = 0;

P0

while(TRUE)

{

while (turn!=0) ;

C0

turn=1 ;

non-critical section

}

P1

while(TRUE)

{

while (turn!=1) ;

C1

turn=0 ;

non-critical section

}

Thứ tự thực hiện miễn găng: $C0 \rightarrow C1 \rightarrow C0 \rightarrow C1 \rightarrow \dots$

Nhược điểm: tiến trình C1 không vào được miễn găng nếu C0 chưa vào miễn găng và ngược lại.

$(C0 \rightarrow C0 \rightarrow C1 \rightarrow C1 \rightarrow \dots)$

- b) Giải pháp kiểm tra đối phương: Nếu tiến trình kia không ở trong miền găng thì được vào miền găng

bool flag[2] = {FALSE, FALSE};

P0

while(TRUE)

{

flag[0]= TRUE;
while (flag[1]);

C0

flag[0] = FALSE;

non-critical section

}

flag[0] = FALSE: P0 ở ngoài miền găng

TRUE : P0 ở trong miền găng

P1

while(TRUE)

{

flag[1]= TRUE;
while (flag[0]);

C1

flag[1] = FALSE;

non-critical section

}

Ví dụ: Nhân viên đến thăm nhà giám đốc.

Nhược điểm : Khi hai tiến trình cùng thực hiện lệnh `flag=TRUE` rồi mới thực hiện lệnh `while` thì cả P0 và P1 bị treo tại lệnh `while`

b) Giải pháp Peterson: Dựa trên giải pháp kiểm tra đối phương

```
bool flag[2] = {FALSE, FALSE};  
int turn = 0;
```

P0

```
while(TRUE)
```

```
{
```

```
    flag[0]= TRUE;  
    turn = 1;  
    while (turn==1) && (flag[1]) ;
```

C0

```
    flag[0] = FALSE;
```

non-critical section

```
}
```

P1

```
while(TRUE)
```

```
{
```

```
    flag[1]= TRUE;  
    turn = 0;  
    while (turn==0) && (flag[0]) ;
```

C1

```
    flag[1] = FALSE;
```

non-critical section

```
}
```

Nếu flag[0] và flag[1] đều TRUE, tuy nhiên turn không thể cùng lúc vừa 0 vừa 1 → 1 tiến trình vào miền găng

2. Giải pháp khóa (dùng cho nhiều tiến trình)

Dùng một khóa, tiến trình nào lấy khóa được thì vào miền găng. Sau khi ra khỏi miền găng thì trả khóa lại.

```
bool lock;  
while(TRUE)  
{  
    while (lock);  
    lock = TRUE;  
    critical section  
    lock = FALSE;  
    non-critical section  
}
```

- lock = FALSE: không có tiến trình ở trong miền găng
- lock = TRUE : có tiến trình trong miền găng.

Nếu lock=false và 2 tiến trình cùng thực hiện lệnh `while` → cả hai vào miền găng → cần thực hiện liên tục lệnh `while` và lệnh gán `lock = TRUE`

Giải pháp “busy and waiting” - Thực hiện bằng phần cứng

- Giải pháp cấm ngắt:

Cho phép tiến trình cấm tất cả các ngắt, kể cả ngắt đồng hồ, trước khi vào miền găng, và phục hồi ngắt khi ra khỏi miền găng.

→ Giải pháp: **dùng phần cứng** để bảo đảm lệnh while và gán lock không bị xâm phạm

```
bool TestAndSet(bool *lock)
{
    bool v= *lock;
    *lock = TRUE;
    return v;
}
```

```
while(TRUE)
```

```
{
```

```
    while (TestAndSet(&lock)) ;
```

critical section

```
    lock = FALSE;
```

non-critical section

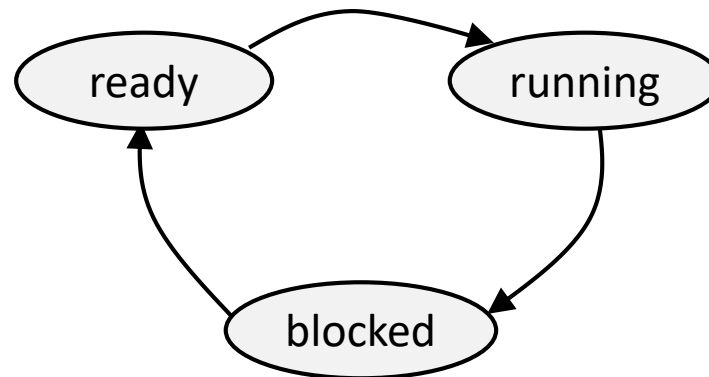
```
}
```

III. Giải pháp đồng bộ ‘Sleep&Wakeup’

Nhận xét: đối với ‘busy waiting’, CPU phải thực hiện lệnh ‘chờ’ vào miền găng (lệnh while) → hao phí CPU.

Giải pháp ‘Sleep&Wakeup’:

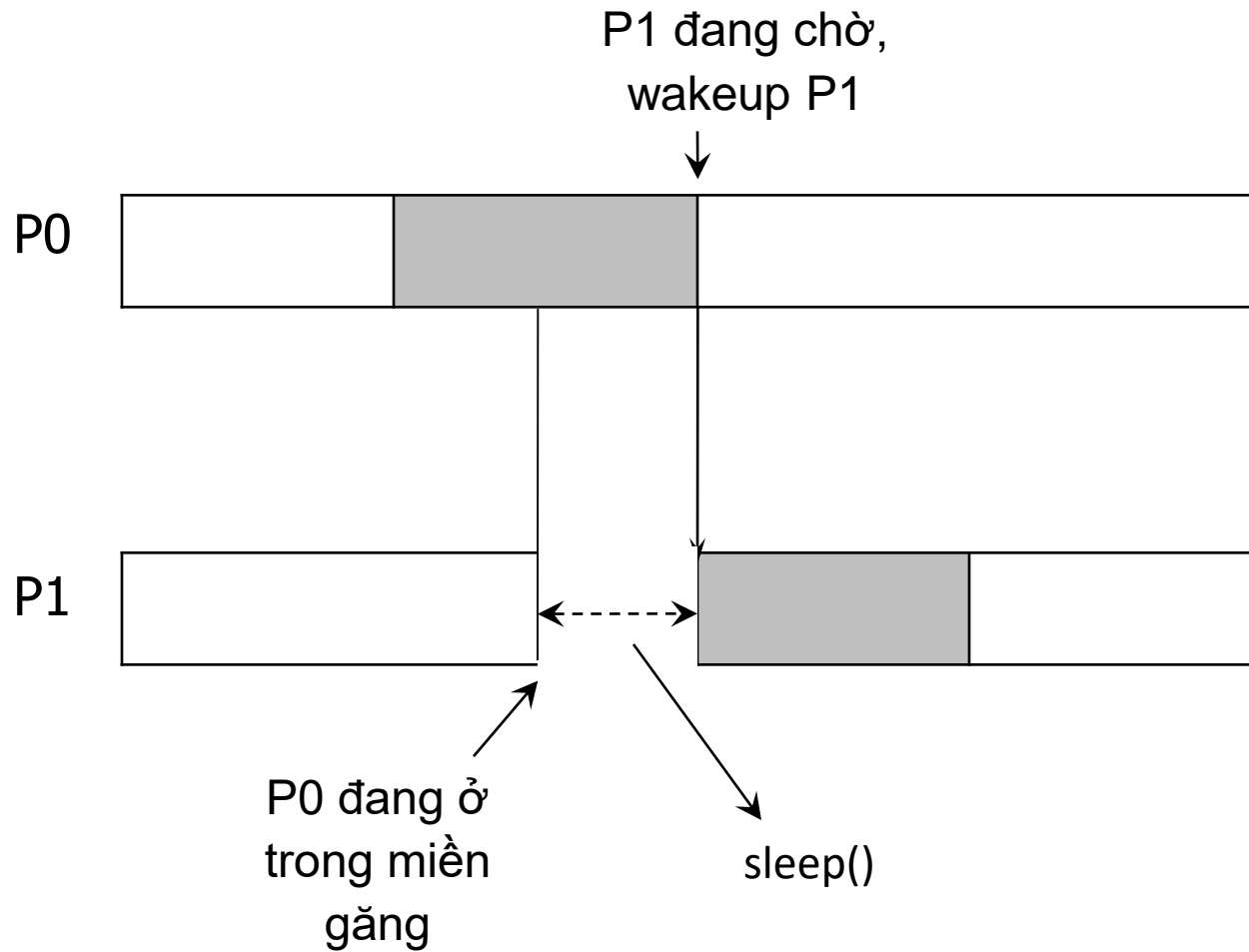
- Chuyển tiến trình vào trạng thái blocked nếu phải chờ vào miền găng (sleep).
- Khi đủ điều kiện vào miền găng, tiến trình được chuyển qua trạng thái ready sẵn sàng dùng CPU (wakeup).



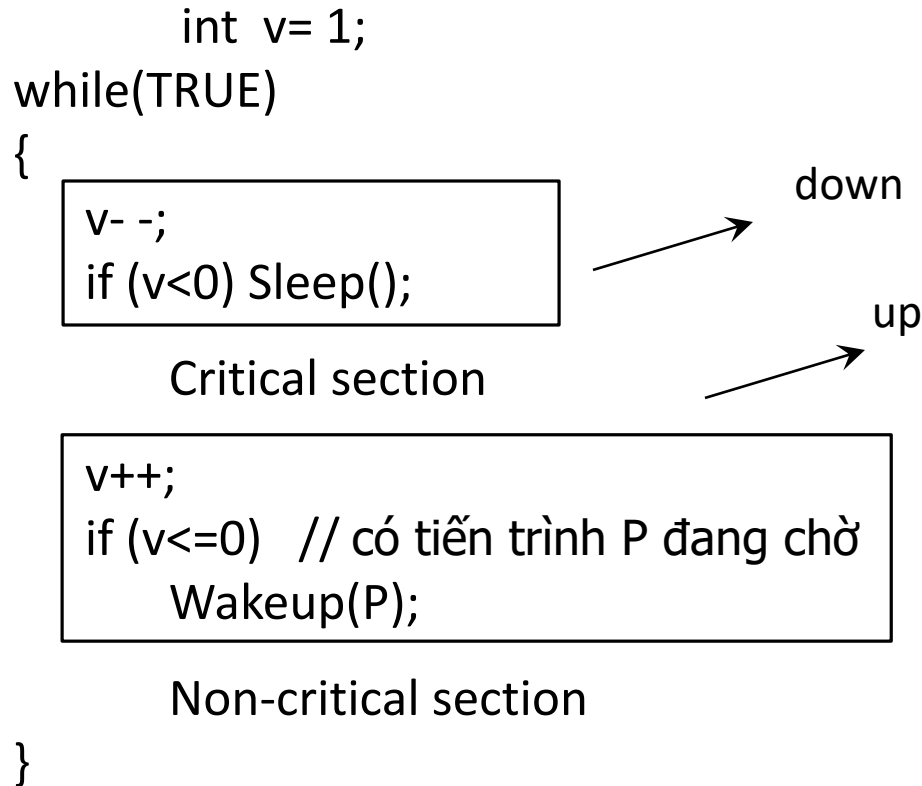
Miền găng



Ngoài miền găng



Sơ đồ ý tưởng:



- $v=1$: không có tiến trình ở trong miền găng
- $v<1$: có tiến trình trong miền găng.

- Nếu có nhiều tiến trình: sau khi thực hiện bước down ($-v$) cho biết số tiến trình đang bị blocked chờ miền găng.

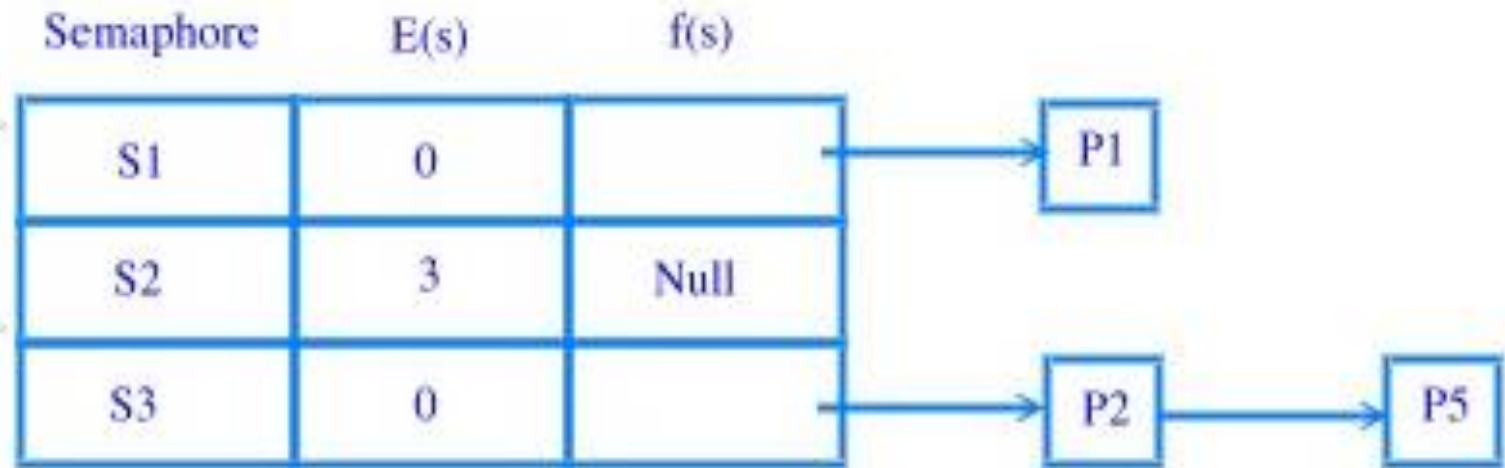
Giải pháp “sleep and wakeup”

- Tồn tại: Tiến trình vẫn có thể bị chặn không cho vào miền găng do:
 - Thao tác kiểm tra điều kiện và thao tác sleep có thể bị ngắt.
 - Tín hiệu wakeup có thể bị “thất lạc”
- Giải pháp:
 - Dùng semaphore
 - Dùng monitor
 - Dùng message

Semaphore

- Semaphore: Là một cấu trúc đặc biệt với các thuộc tính:
 - Một số nguyên dương e
 - Một hàng đợi f lưu danh sách các tiến trình đang bị khóa
 - Hai thao tác được định nghĩa trên semaphore:
 - Down(s):*** giảm giá trị e đi 1. Nếu $e \geq 0$ thì tiếp tục xử lý. Ngược lại, nếu $e < 0$, tiến trình phải chờ.
 - Up(s):*** tăng giá trị của e lên 1. Nếu có tiến trình đang chờ thì chọn một tiến trình để đánh thức.

Semaphore



Yêu cầu của semaphore: Khi tiến trình đang xử lý Semaphore thì không được ngắt!!!! (Giống như giải pháp phần cứng)

Hai thao tác của semaphore

Down(s):

```
e = e - 1;  
if e < 0 {  
    status(P)= blocked;  
    enter(P,f(s));  
}
```

Up(s):

```
e = e + 1;  
if (e) ≤ 0 {  
    exit(Q,f(s));  
    status (Q) = ready;  
    enter(Q,ready-list);  
}
```

Sử dụng semaphore

- Giải quyết điều kiện 1 của miền găng:
Có n tiến trình dùng chung một semaphore để đồng bộ, semaphore được khởi tạo = 1.

```
while (TRUE) {
```

```
    Down(s)
```

```
    critical-section ();
```

```
    Up(s)
```

```
    Noncritical-section ();
```

```
}
```

Tiến trình đầu tiên vào được miền găng (được truy xuất tài nguyên).

Các tiến trình sau phải chờ vì $e(s) < 0$.

Sử dụng semaphore

- Đồng bộ tiến trình

Dùng chung 1 semaphore với giá trị khởi tạo =0

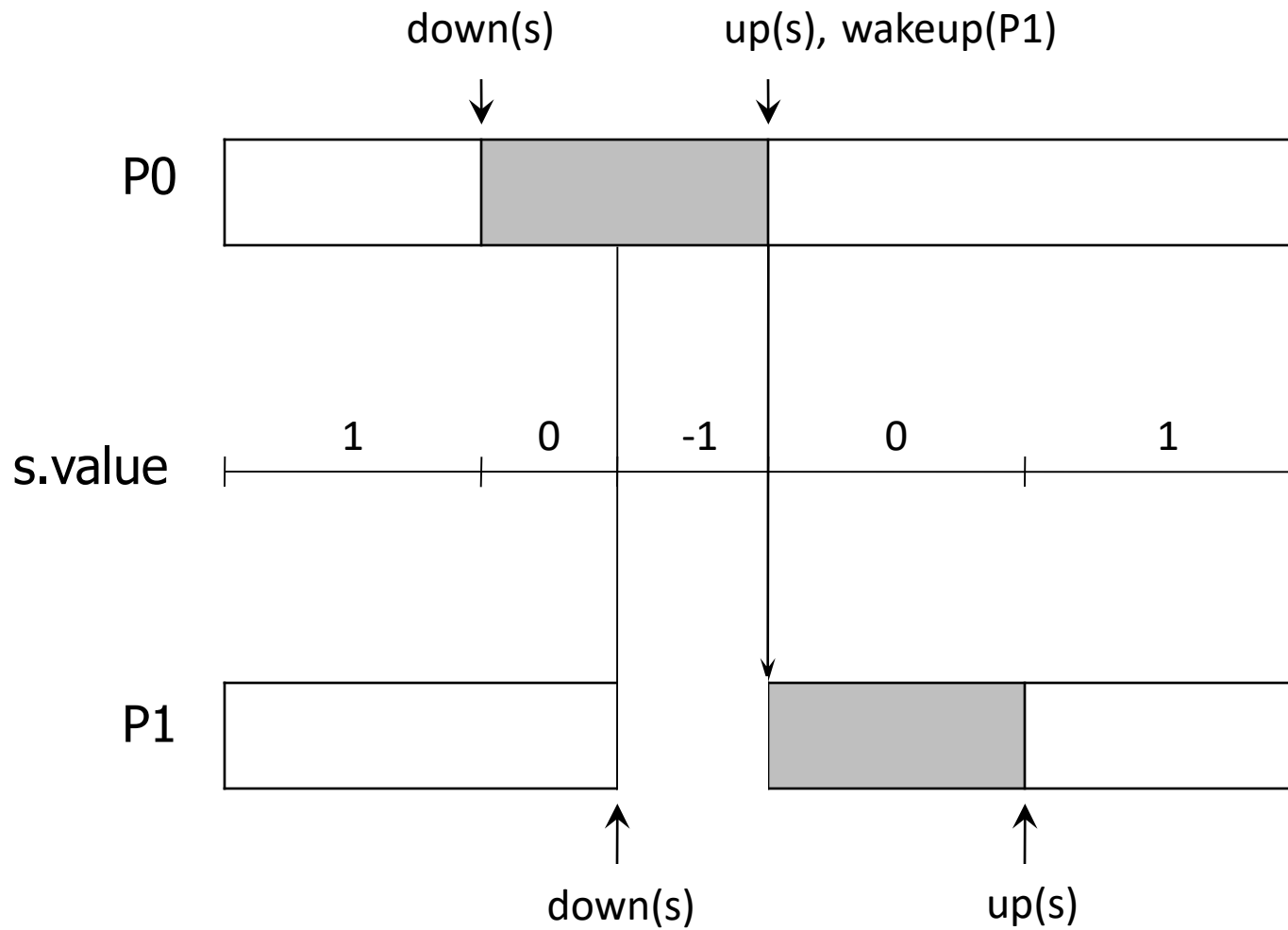
P1:

```
while (TRUE) {  
    job1();  
    Up(s); //đánh thức P2  
}
```

P2:

```
while (TRUE) {  
    Down(s); // chờ P1  
    job2();  
}
```

Ngữ cảnh đồng bộ: có hai tiến trình tương tranh, và tiến trình này phải chờ tiến trình kia kết thúc thì mới xử lý được.



muốn vào miền găng nhưng
 $s.value < 0 \rightarrow \text{sleep}()$

Công dụng mở rộng của Semaphore: cho phép tối đa n tiến trình vào miền găng cùng lúc

```
semaphore s;
```

```
s.value = 2;
```

```
while(TRUE)
```

```
{
```

```
    down (s) ;
```

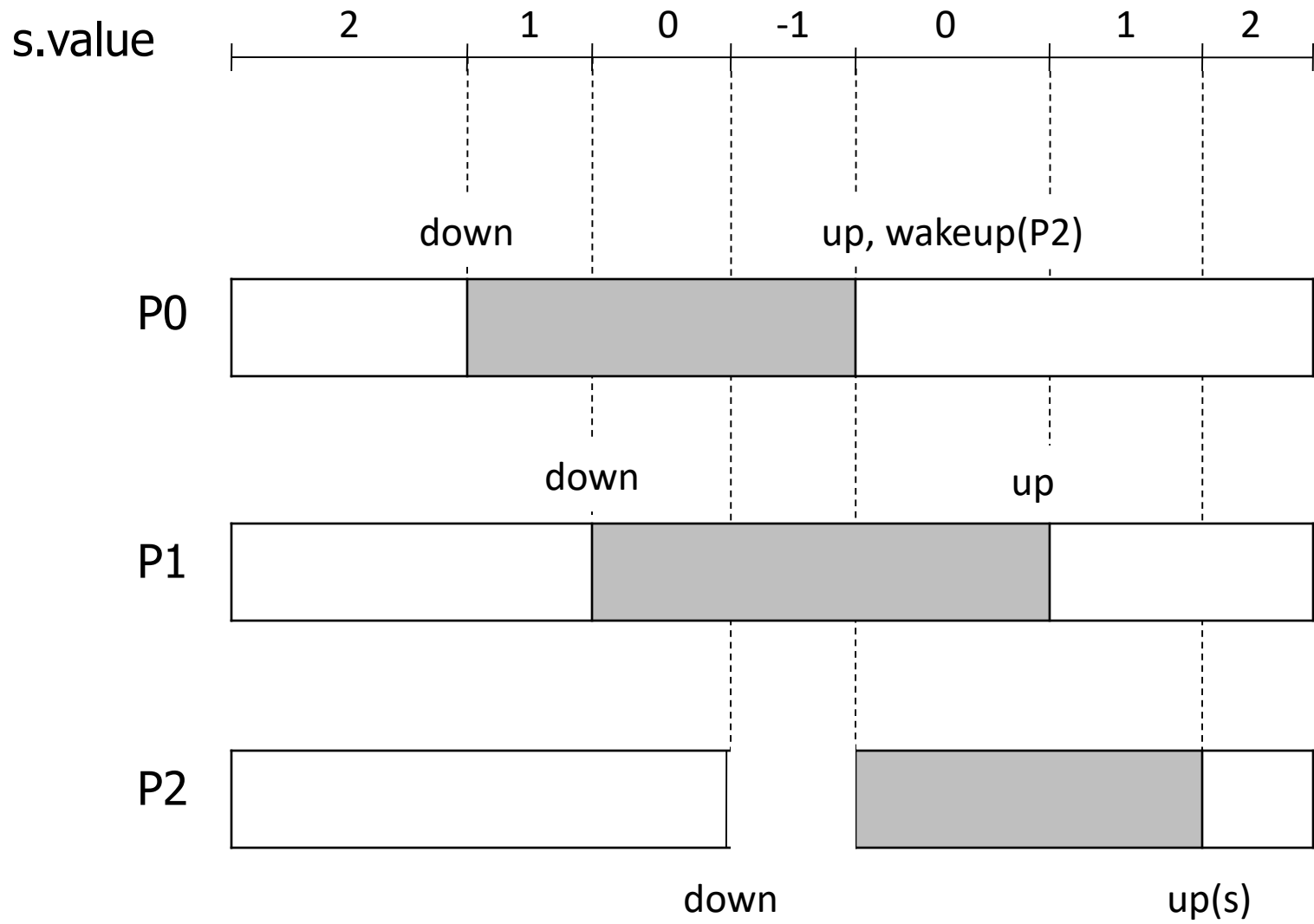
critical section

```
    up (s) ;
```

non-critical section

```
}
```

2 tiến trình có thể vào
miền găng cùng lúc



Vấn đề khi sử dụng semaphore

```
while (TRUE) {  
    Down(s)  
    critical-section ();  
    Noncritical-section ();  
}
```

Người lập trình quên gọi Up(s), hậu quả là từ đó về sau không có tiến trình nào vào được miền găng!

2. Sử dụng Semaphore để phối hợp hoạt động:

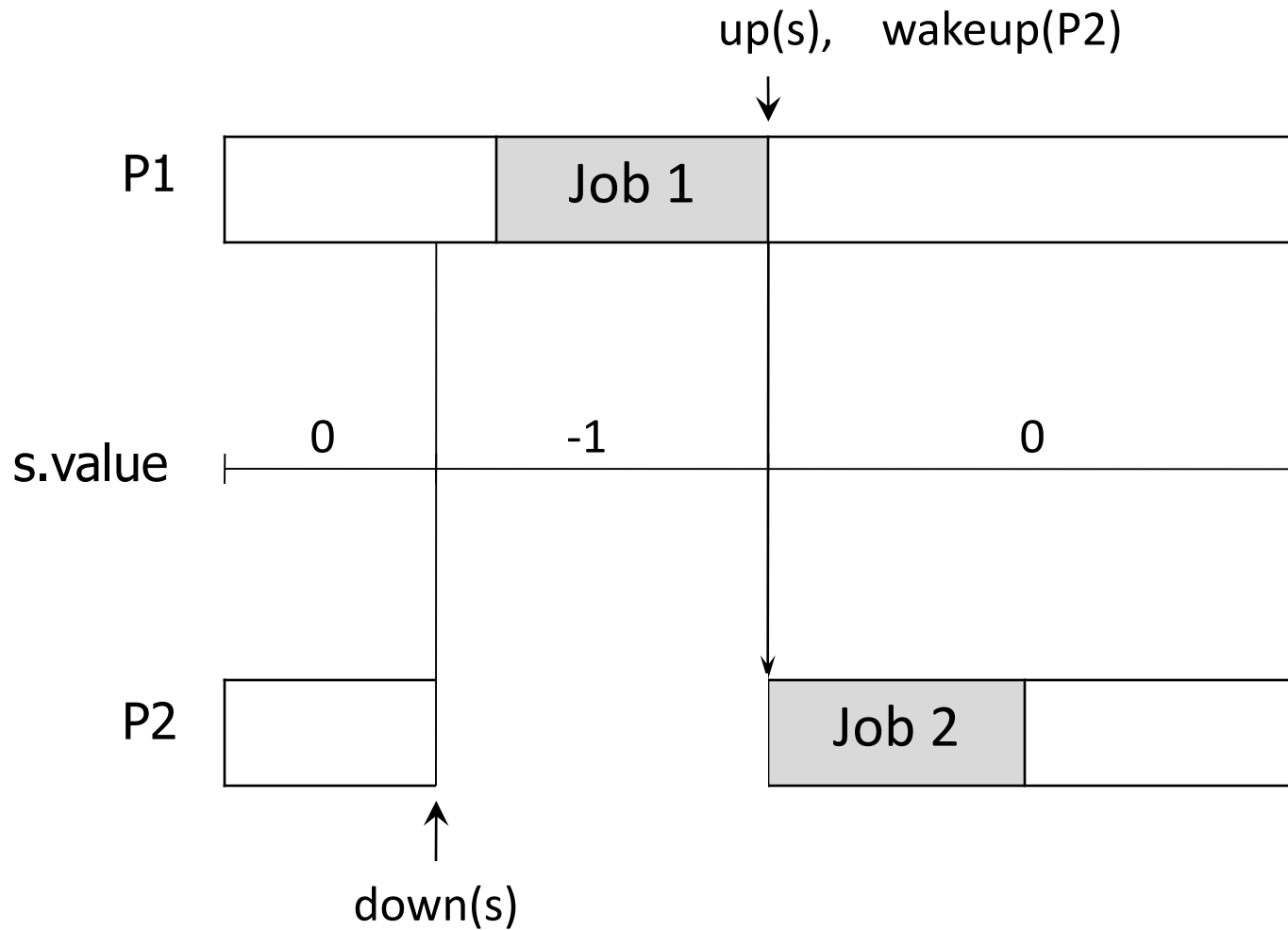
Bên cạnh bảo đảm vấn đề độc quyền truy xuất, phối hợp hoạt động là mục tiêu thứ hai của đồng bộ tiến trình.

Yêu cầu: Job2 của P1 phải thực hiện sau Job1 của P0

```
semaphore s;  
s.value = 0;
```

```
P0  
while(TRUE)  
{  
    Job1();  
    up (s) ;  
}
```

```
P1  
while(TRUE)  
{  
    down (s) ;  
    Job2 () ;  
}
```



muốn thực hiện Job2 nhưng
 $s.value < 0 \rightarrow \text{sleep}()$

Miền găng trong C#:

```
static int TaiKhoan;
static Mutex mutex;

static void RutTien1()
{
    mutex.WaitOne();
    if (TaiKhoan >= 60) TaiKhoan = TaiKhoan - 60;
    mutex.ReleaseMutex();
}

static void RutTien2()
{
    mutex.WaitOne();
    if (TaiKhoan >= 70) TaiKhoan = TaiKhoan - 70;
    mutex.ReleaseMutex();
}

static void Main(string[] args)
{
    mutex = new Mutex();
    TaiKhoan = 100;
    Thread t1 = new Thread(new ThreadStart(RutTien1));
    Thread t2 = new Thread(new ThreadStart(RutTien2));
    t1.Start();
    t2.Start();
}
```

Semaphore trong C#:

```
static int TaiKhoan;
static Semaphore s;

static void RutTien1()
{
    s.WaitOne();
    if (TaiKhoan >= 60) TaiKhoan = TaiKhoan - 60;
    s.Release ();
}

static void RutTien2()
{
    s.WaitOne();
    if (TaiKhoan >= 70) TaiKhoan = TaiKhoan - 70;
    s.Release ();
}

static void Main(string[] args)
{
    s = new Semaphore(1, 1);
    TaiKhoan = 100;
    Thread t1 = new Thread(new ThreadStart(RutTien1));
    Thread t2 = new Thread(new ThreadStart(RutTien2));
    t1.Start();
    t2.Start();
}
```

Monitor

Monitor là một cấu trúc với các thuộc tính:

- Các *biến điều kiện* (c), hàng đợi chứa các tiến trình bị khóa $f(c)$ và hai thao tác kèm theo là Wait và Signal:
 - Wait(c): chuyển trạng thái tiến trình gọi sang blocked , và đặt tiến trình này vào hàng đợi của c .
 - Signal(c): nếu có một tiến trình đang bị khóa trong hàng đợi của c , tái kích hoạt tiến trình đó, và tiến trình gọi sẽ rời khỏi monitor.

Thao tác Wait và Signal

```
Wait(c){  
    status(P)= blocked;  
    enter(P,f(c));  
}
```

```
Signal(c){  
    if (f(c) != NULL){  
        exit(Q,f(c));           //Q là tiến trình chờ trên c  
        status-Q = ready;  
        enter(Q,ready-list);  
    }  
}
```

Cài đặt monitor

```
monitor <tên monitor > {  
    <các biến chung>  
    <các biến điều kiện c>;  
    //Danh sách các thủ tục  
    Action-1(){}  
    ....  
    Action-n(){}  
}
```

Monitor dùng để điều khiển truy xuất độc quyền đối với tài nguyên. Mỗi tài nguyên có một monitor riêng và tiến trình truy xuất thông qua monitor đó

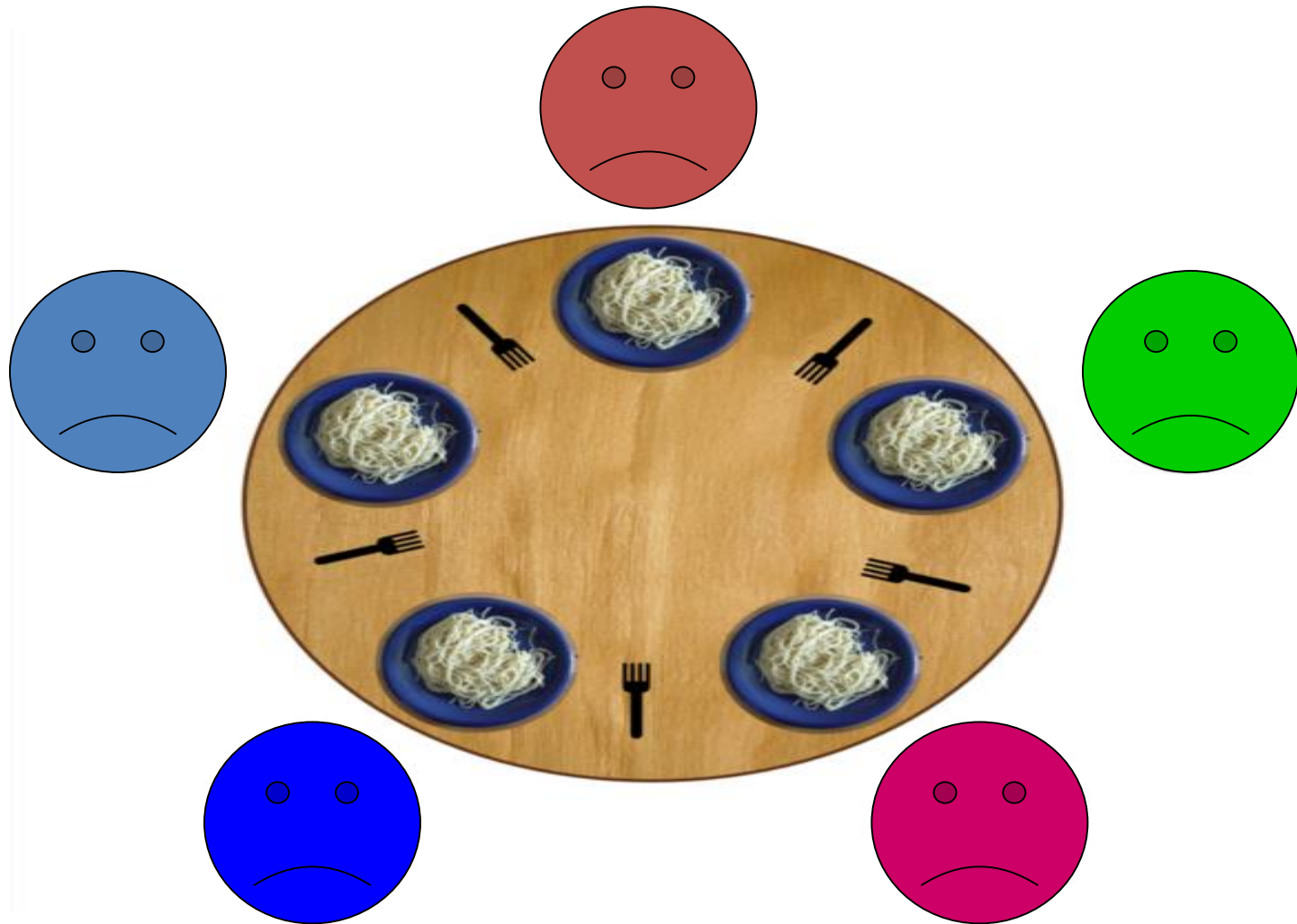
Tiến trình sử dụng monitor

```
while (TRUE) {  
    Noncritical-section ();  
    <monitor>.Action-i; //miền găng  
    Noncritical-section ();  
}
```

Nhận xét: -Thao tác trên monitor do trình biên dịch thực hiện -> giảm sai sót.

-Trình biên dịch phải hỗ trợ monitor

Bài toán triết gia ăn tối



Xây dựng monitor

```
monitor philosopher{  
    enum (thinking, hungry, eating) state[5];  
    condition self[5];  
    void init();  
    void test(int i);  
    void pickup( int i);  
    void putdown (int i);  
}
```

Khởi tạo monitor

- Tất cả các triết gia đang suy nghĩ...

```
void init (){  
    for(int i=0; i < 5; i++)  
        state[i] = thinking;  
}
```

Kiểm tra điều kiện trước khi ăn

- Nếu TGi đang đói và cả hai TG bên cạnh đều không ăn thì TGi ăn.

```
void test (int i) {  
    if ((state[i] == hungry) && state[(i + 4)%5] != eating)  
        && state[(i + 1)%5] != eating)  
        self[i].signal(); //Đánh thức TGi  
        state[i] = eating;  
}
```

Lấy một chiếc nĩa

```
void pickup(int i) {  
    state[i] = hungry;  
    test(i);  
    if (state[i] != eating)  
        self[i].wait(); //TGì chờ đến lượt mình  
}
```

Trả một chiếc nữa

```
void putdown(int i){  
    state[i] = thinking;  
    test((i + 4) % 5);  
    test((i + 1) % 5);  
}
```

Cài đặt tiến trình

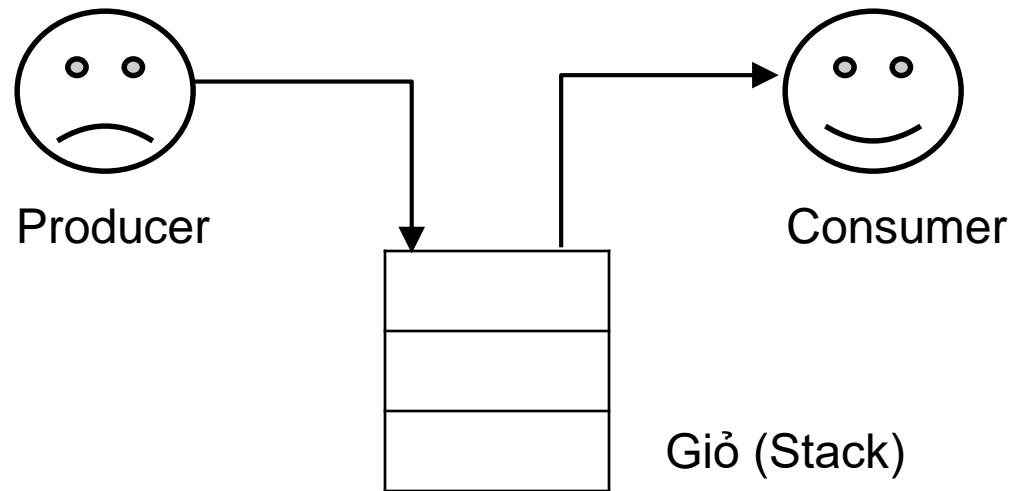
Philosophers pp;

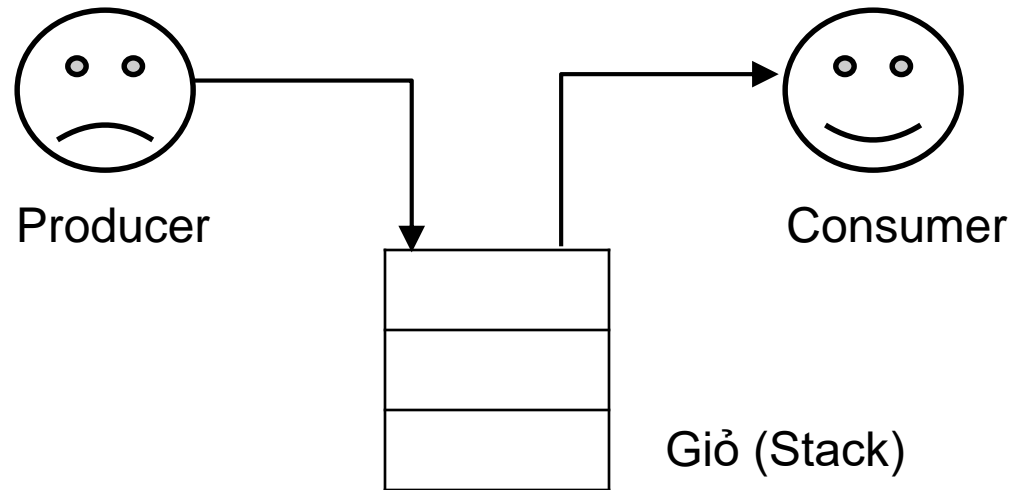
Tiến trình i:

```
while(1) {  
    Noncritical-section();  
    pp.pickup(i);  
    eat();  
    pp.putdown(i);  
    Noncritical-section();  
}
```


IV. Một số ví dụ về đồng bộ

1. Bài toán nhà sản xuất, người tiêu thụ (Producer-Consumer)
 - Nhà sản xuất cung cấp hàng bỏ vào giỏ, người tiêu thụ lấy hàng ra khỏi giỏ.





Yêu cầu độc quyền truy xuất: Nhà sản xuất và người tiêu thụ không cùng lúc truy xuất giỏ.

Yêu cầu phối hợp:

- Nhà sản xuất không được bỏ hàng vào giỏ đầy.
- Người tiêu thụ không được lấy hàng từ giỏ trống.

```
#define STACK_SIZE 10
int index= -1;
int stack[STACK_SIZE];
```

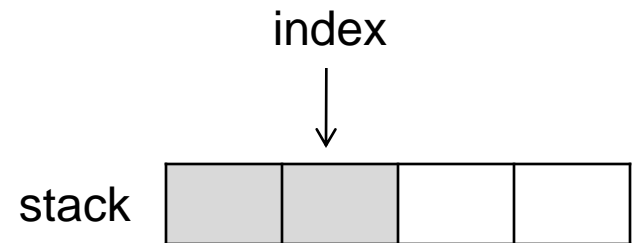
```
void Producer()
{
    while(TRUE) {
        int item= random();
        while (index==STACK_SIZE-1);

        index++;
        stack[index]= item;
    }
}
```

```
void Consumer()
{
    while(TRUE) {
        while (index== -1);

        int item = stack[index];
        index--;
    }
}
```

Cách thực hiện bình thường:
→Không bảo đảm độc quyền truy xuất



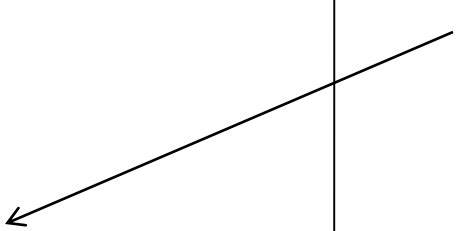
```
#define STACK_SIZE 10
int index= -1;
int stack[STACK_SIZE];
```

```
semaphore mutex= 1;
```

```
void Producer()
{
    while(TRUE) {
        int item= random();
        while (index==STACK_SIZE-1);

        down(mutex);
        index++;
        stack[index]= item;
        up(mutex);
    }
}
```

Mục đích độc quyền truy xuất



```
void Consumer()
{
    while(TRUE) {
        while (index== -1);

        down(mutex);
        int item=stack[index];
        index--;
        up(mutex);
    }
}
```

```
#define STACK_SIZE 10
int index= -1;
int stack[STACK_SIZE];
```

```
semaphore mutex= 1;
semaphore empty= STACK_SIZE;
semaphore full= 0;
```

```
void Producer()
{
    while(TRUE) {
        int item= random();

        down(empty) ;

        down(mutex);
        index++;
        stack[index]= item;
        up(mutex);

        up(full) ;
    }
}
```

Mục đích độc quyền truy xuất

Mục đích phối hợp hoạt động

```
void Consumer()
{
    while(TRUE) {

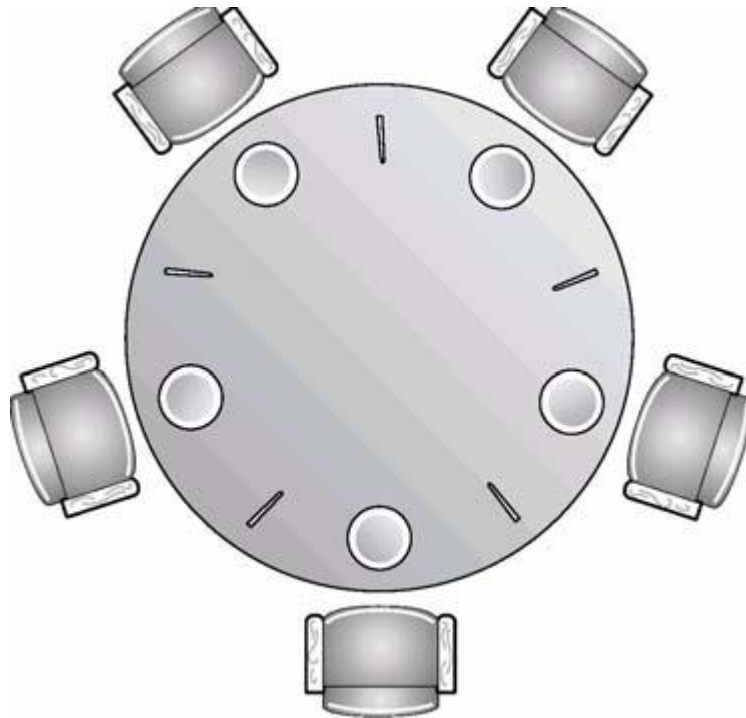
        down(full) ;

        down(mutex);
        int item=stack[index];
        index--;
        up(mutex);

        up(empty) ;
    }
}
```

2. Bài toán Triết gia ăn tối (Dining Philosophers)

- Năm triết gia và 5 cây đũa.
- Để ăn tối một triết gia cần có đủ 2 cây đũa 2 bên



Độc quyền truy xuất cho
từng cây đũa

```
semaphore chopstick[5]= {1,1,1,1,1};
```

Philosopher thứ i

```
void Dining(int i)  
{
```

```
    while(TRUE) {
```

```
        down(chopstick[i]);  
        down(chopstick[(i+1)%5]);
```

Cầm đũa bên phải

```
        // eating ...
```

Cầm đũa bên trái

```
        up(chopstick[i]);  
        up(chopstick[(i+1)%5]);
```

```
        // thinking ...
```

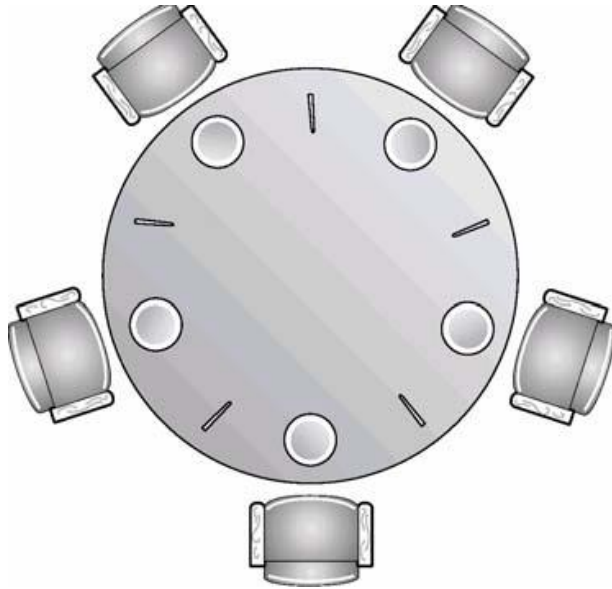
```
    }
```

```
}
```

VI. Tắc nghẽn - Deadlock

1. Khái niệm tắc nghẽn:

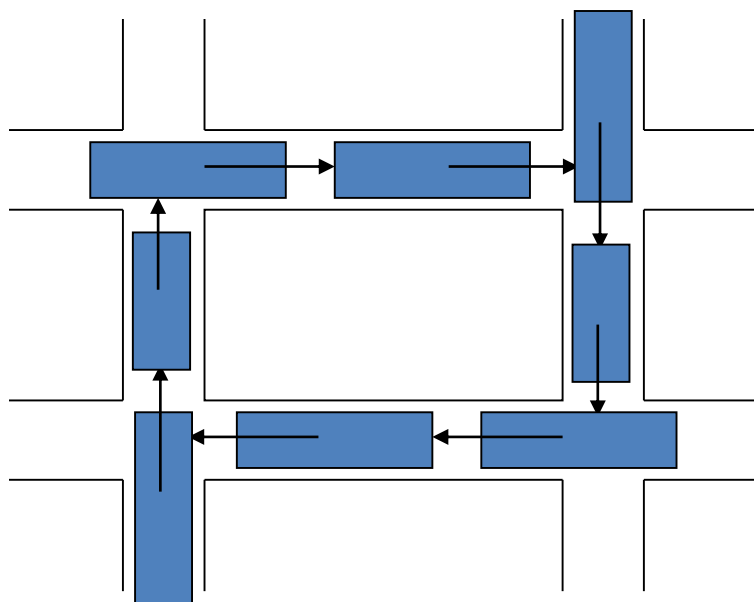
a) Ví dụ 1: Vấn đề triết gia ăn tối:



```
semaphore chopstick[5]= {1,1,1,1,1};  
void Dining(int i)  
{  
    while(TRUE) {  
        down(chopstick[i]);  
        down(chopstick[(i+1)%5]);  
        // eating ...  
        up(chopstick[i]);  
        up(chopstick[(i+1)%5]);  
        // thinking ...  
    }  
}
```

Điều gì xảy ra sau khi năm triết gia đồng thời lấy xong đũa bên phải ? (thực hiện xong `down(chopstick[i])`)

b) Ví dụ 2: Vấn đề kẹt xe:



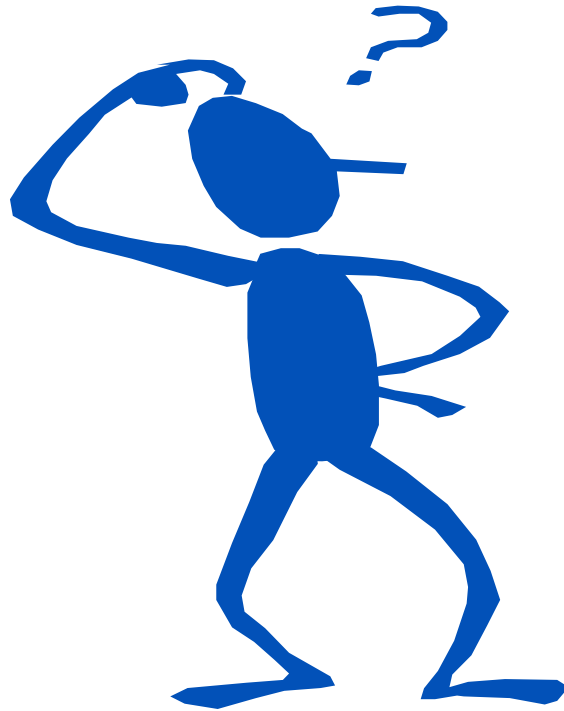
c) Khái niệm tắc nghẽn:

Một tập hợp các tiến trình được gọi là ở trong tình trạng tắc nghẽn khi:

- Mỗi tiến trình trong tập hợp đều yêu cầu tài nguyên.
- Các tài nguyên này đang được chiếm giữ bởi các tiến trình khác trong tập hợp.

Câu hỏi: trong ví dụ 1 và 2, cái gì là tài nguyên gây tắc nghẽn?

Q & A



Câu hỏi ôn tập

1. Nêu khái niệm tranh đoạt điều khiển và độc quyền truy xuất.
2. Miền găng tránh vấn đề tranh đoạt điều khiển như thế nào?
3. Nêu 3 yêu cầu của miền găng.
4. Giải pháp đồng bộ Busy waiting khác giải pháp Sleep & Wakeup ở điểm nào? Phương pháp nào tối ưu hơn?
5. Các hàm TestAndSet, up, down phải thực hiện trong điều kiện đặc biệt nào? Tại sao?

6. Trong giải pháp semaphore áp dụng cho miền găng, nếu khởi động giá trị value là 2 thay vì 1 thì điều gì xảy ra?
7. Ngoài vấn đề miền găng, semaphore có thể được sử dụng trong tình huống nào?
8. Nêu khái niệm tắc nghẽn.
9. Trạng thái an toàn là gì?

Bài tập

1. Xét giải pháp đồng bộ Peterson sửa đổi như sau:

```
while(TRUE)
```

```
{
```

```
    flag[0]=TRUE;
```

```
    turn= 0;
```

```
    while (turn==1 && flag[1] );
```

Critical section

```
    flag[0]= FALSE;
```

Non-critical section

```
}
```

```
while(TRUE)
```

```
{
```

```
    flag[1]=TRUE;
```

```
    turn= 1;
```

```
    while (turn==0 && flag[0] );
```

Critical section

```
    flag[1]= FALSE;
```

Non-critical section

```
}
```

Đây có phải là giải pháp đảm bảo độc quyền truy xuất không?

2. Một biến X được chia sẻ bởi hai tiến trình cùng thực hiện đoạn code sau:

```
do{  
    X = X+1;  
    if (X==20) X = 0;  
}while (TRUE);
```

Bắt đầu với giá trị $X=0$, chứng tỏ rằng X có thể vượt quá 20. Hãy sửa đoạn chương trình trên để X không vượt quá 20

3. Xét hai tiến trình xử lý đoạn chương trình sau:

```
process P1 {    A1();  A2();  }  
process P2 {    B1();  B2();  }
```

Đồng bộ hóa sự hoạt động của hai tiến trình này sao cho cả A_1 và B_1 đều hoàn tất trước khi A_2 hay B_2 bắt đầu

4. Tổng quát hóa câu 2 cho các tiến trình xử lý đoạn chương trình sau

```
process P1 { for(i=1; i<=100; i++) Ai(); }
```

```
process P2 { for(j=1; j<=100; j++) Bj(); }
```

Đồng bộ hóa hoạt động của hai tiến trình này sao cho với mọi $2 \leq k \leq 100$, A_k chỉ có thể bắt đầu khi B_{k-1} đã kết thúc và B_k chỉ có thể bắt đầu khi A_{k-1} kết thúc.

5. Xét hai tiến trình sau:

```
process A {  
    while(TRUE)  
        na = na+1;  
}
```

na và nb khởi tạo là 1

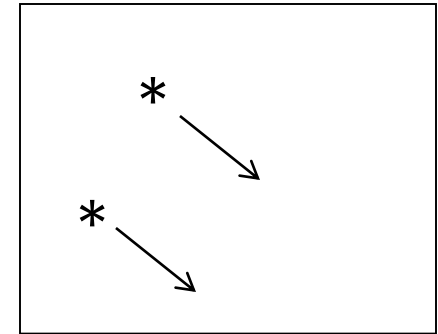
```
process B {  
    while (TRUE)  
        nb = nb+1;  
}
```

- a) Đồng bộ hóa hai tiến trình trên dùng semaphore sao cho tại bất cứ thời điểm nào đều có $na \leq nb + 10$
- b) Thực hiện câu a với điều kiện $nb \leq na \leq nb + 10$

Bài tập thực hành

1. Viết chương trình đồng thời in ra 1000 chữ "World" và 1000 chữ "Earth". Trong đó từ "World" có màu đỏ và từ Earth có màu xanh. Quan sát thấy hiện tượng gì? Giải thích lý do.
2. Dùng mutex sửa lại bài 1 để khắc phục hiện tượng trên.

3. Viết chương trình cho phép 2 (và tổng quát là n) trái banh bay cùng lúc trong màn hình.



4. Áp dụng Semaphore, sửa chương trình trái banh bay trên, sao cho tại một thời điểm chỉ có tối đa 2 quả banh đi vào 1/3 phải màn hình.

