



Đồng bộ tiến trình

Nội dung

- Nhu cầu thông tin giữa các tiến trình
- Tranh đoạt điều khiển và miền găng
- Các giải pháp đồng bộ

Nhu cầu thông tin giữa các 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!
```

Bài toán 1

■ P1

```
if (So_du > Tien_rut)
```



```
So_du = So_du - Tien_rut  
else  
    Access denied!
```

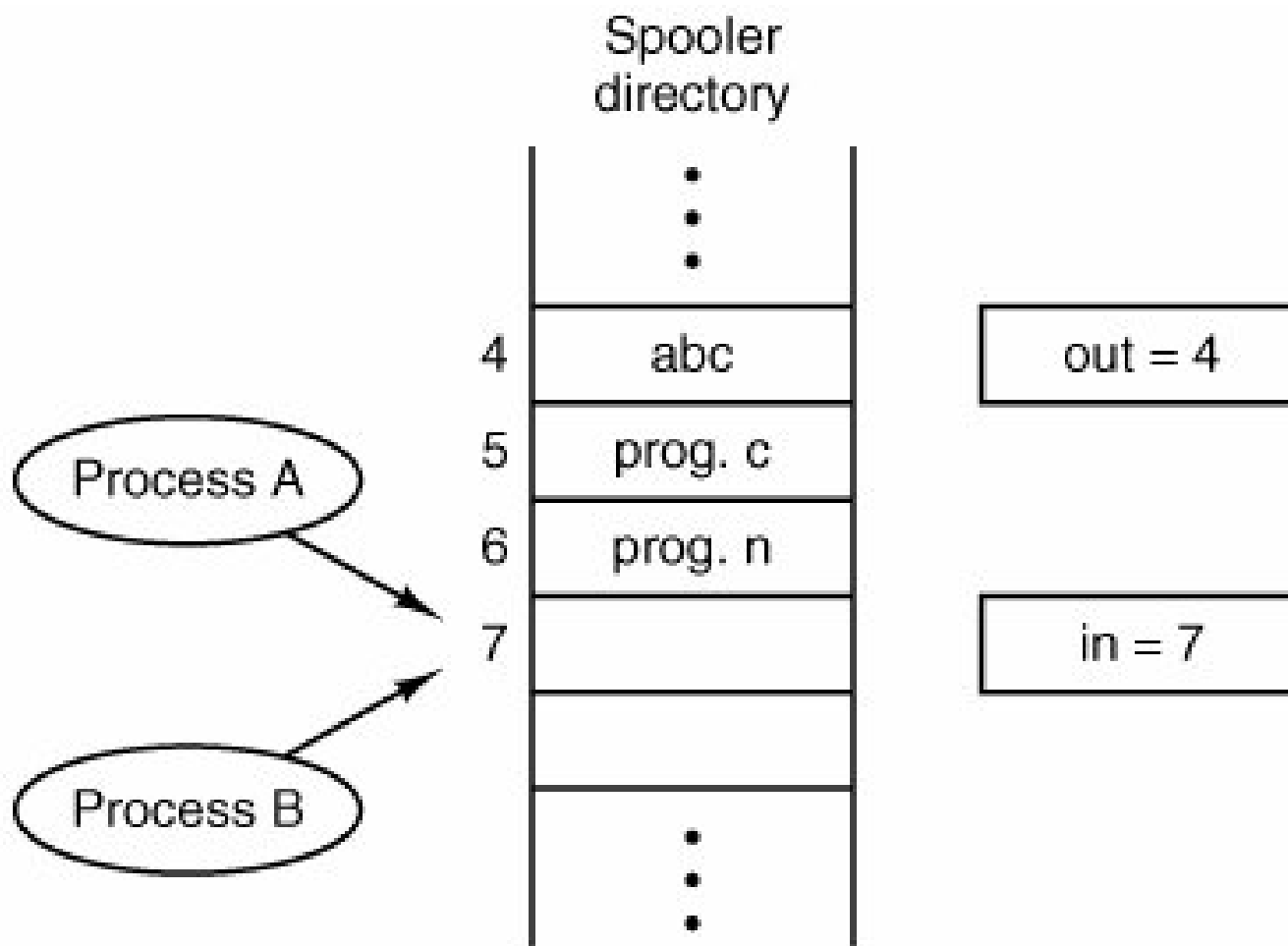
■ P2

```
if (So_du > Tien_rut)  
    So_du = So_du -  
    Tien_rut
```



```
else  
    Access denied!
```

Bài toán 2



Bài toán 2

■ Process A

Next_free_slot = 7

Put file in slot7

Wait the print job

■ Process B

Next_free_slot = 7

Put file in slot7

Wait for the print
job (for ever!!!)

Tranh đoạt điều khiển (race condition)

`i=0;`

Thread a:

```
while(i < 10)
    i = i + 1;
print "A won!";
```

Thread b:

```
while(i > -10)
    i = i - 1;
print "B won!";
```

Miền găng

- Race condition (tương tranh): nhiều tiến trình cùng thực thi mà kết quả phụ thuộc vào thứ tự thực thi của các tiến trình.
- Miền găng (critical section): đoạn chương trình có khả năng gây ra lỗi truy xuất đối với tài nguyên chung.

Nguyên tắc

1. *Tại 1 thời điểm chỉ có 1 tiến trình trong miền găng.*
2. *Không có ràng buộc về tốc độ của các tiến trình và số lượng bộ xử lý trong hệ thống*
3. *Một tiến trình tạm dừng bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng.*
4. *Không có tiến trình nào phải chờ vô hạn để được vào miền găng.*

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

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

■ Dùng cờ hiệu (biến lock)

```
while (TRUE) {  
    while (lock == 1); // wait  
    lock = 1;  
    critical-section ();  
    lock = 0;  
    Noncritical-section ();  
}
```

Nhận xét: có thể vi phạm điều kiện 1

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

■ Kiểm tra luân phiên

Tiến trình A:

```
while (TRUE) {  
    while (turn != 0); // wait  
        critical-section ();  
    turn = 1;  
    Noncritical-section ();  
}
```

Tiến trình B:

```
while (TRUE) {  
    while (turn != 1); // wait  
        critical-section ();  
    turn = 0;  
    Noncritical-section ();  
}
```

Nhận xét: Có thể vi phạm điều kiện 3!

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

■ Giải pháp Peterson:

```
int turn;  
int interest[2] = {FALSE, FALSE};  
while (TRUE) {  
    int j = 1-i; // nếu i là tiến trình 1 thì j là tiến trình 0  
    interest[i]= TRUE;  
    turn = j;  
    while (turn == j && interest[j]==TRUE);  
    critical-section ();  
    interest[i] = FALSE;  
    Noncritical-section ();  
}
```

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 “busy and waiting”- Thực hiện bằng phần cứng

■ Lệnh Test-and-Set Lock (TSL)

```
int Test-and-Set Lock(int target) {  
    int tmp = target;  
    target = 1;  
    return tmp;  
}  
  
while (1) {  
    while (Test-and-Set Lock(lock));  
    critical-section ();  
    lock = 0;  
    Noncritical-section ();  
}
```

Giải pháp “sleep and wakeup”

- Chuyển tiến trình chưa đủ điều kiện vào miền găng sang trạng thái blocked (sleep) để khỏi chiếm dụng CPU.
- Một tiến trình ra khỏi miền găng sẽ “wakeup” tiến trình đang khóa để tiếp tục vào miền găng.
- Sleep và wakeup là các thao tác đơn (không thể bị ngắt ở giữa)

Giải pháp “sleep and wakeup”

int busy; // 1 nếu miền găng đang bị chiếm, ngược lại là 0
int blocked; // đếm số lượng tiến trình đang bị khóa

```
while (1) {  
    if (busy){  
        blocked = blocked + 1;  
        → sleep();  
    }  
    else busy = 1;  
    critical-section ();  
    busy = 0;  
    if(blocked){  
        wakeup(process);  
        blocked = blocked - 1;  
    }  
    Noncritical-section ();  
}
```

Tình huống:

- A vào miền găng
- B vào sau nên phải “ngủ”
- B chưa “ngủ” thì CPU chuyển cho A.
- A ra khỏi miền găng và “đánh thức” B
- B đang “thức” nên không nhận tín hiệu đánh thức.
- B không bao giờ được vào 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

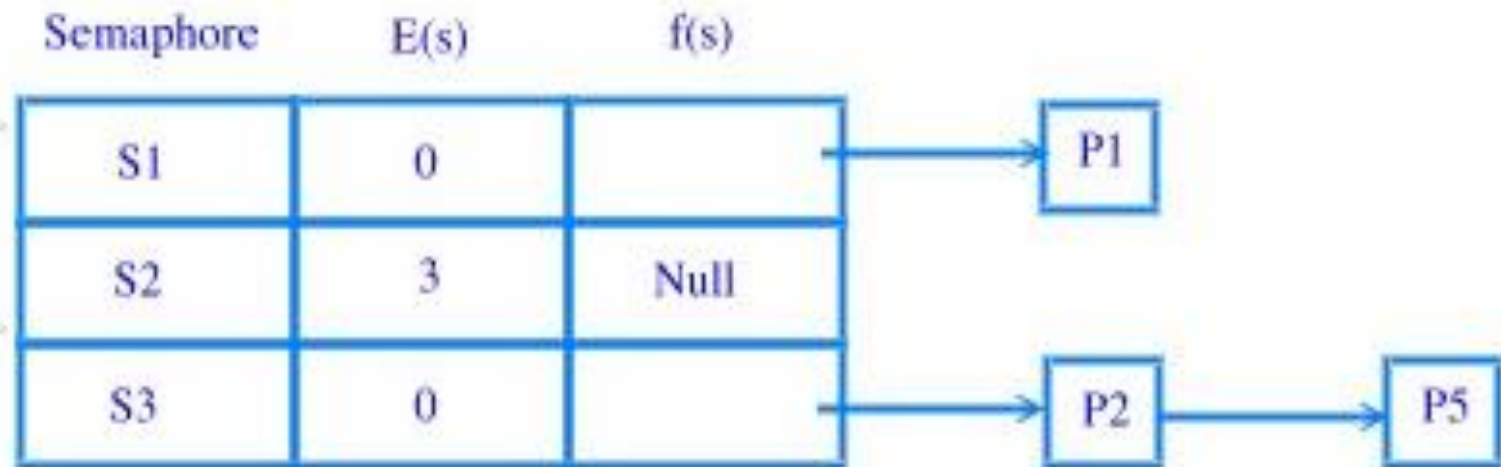
Semaphore

- 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.

Vấn đề khi sử dụng semaphore

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

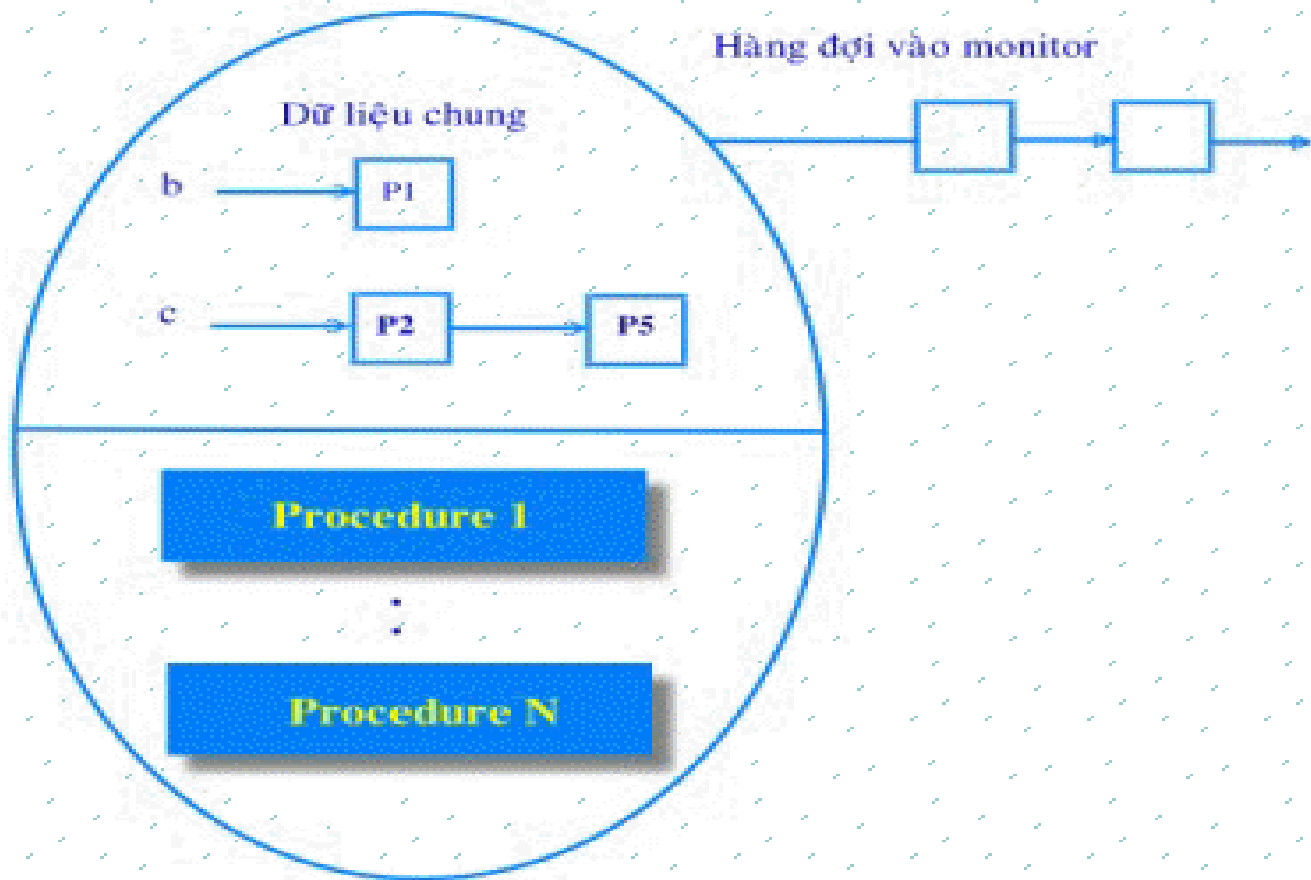
Người lập trình quên gọi Up(s), hậu quả là từ đó về sau không có tiến trình nào vào được miền găng!

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.

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

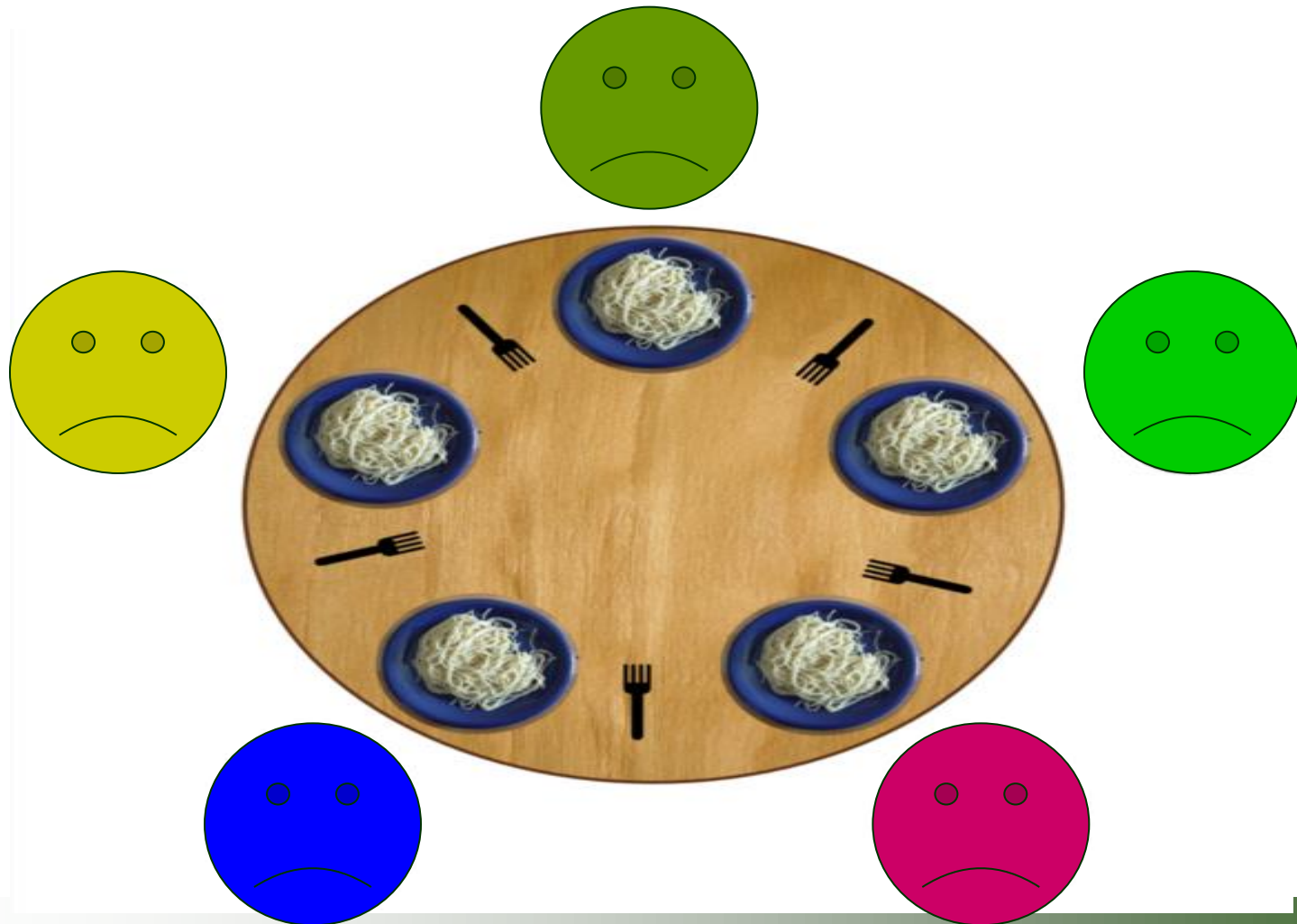
Tiến trình sử dụng monitor

```
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 nĩa

```
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();  
}
```

Một số bài tập đồng bộ hóa

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 giá trị X có thể vượt quá 20. Dùng semaphore để bảo đảm X không vượt quá 20 ?

Một số bài tập đồng bộ hóa

Xét hai tiến trình:

$P1 \{A1; A2\}$ và $P2 \{B1; B2\}$

Dùng semaphore để đồng bộ sao cho cả A1 và B1 đều hoàn tất trước khi A2 hay B2 bắt đầu

Một số bài tập đồng bộ hóa

P1: $w = x1 * x2$

P2: $v = x3 * x4$

P3: $y = v * x5$

P4: $z = v * x6$

P5: $y = w * y$

P6: $z = w * z$

P7: $ans = y + z$

Dùng
Semaphore
để đồng bộ
các tiến
trình bên để
kết quả bài
toán được
chính xác

Bài tập 1

Xét giải pháp đồng bộ hoá sau :

```
while (TRUE) {  
    int j = 1-i;  
    flag[i]= TRUE; turn = i;  
    while (turn == j && flag[j]==TRUE);  
    critical-section ();  
    flag[i] = FALSE;  
    Noncritical-section ();  
}
```

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

Bài tập 2

Giả sử một máy tính không có lệnh TSL, nhưng có lệnh Swap có khả năng hoán đổi nội dung của hai từ nhớ chỉ bằng một thao tác không thể phân chia :

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

Sử dụng lệnh này có thể tổ chức truy xuất độc quyền không ?

Bài tập 3

- Nếu thuật toán Petterson không dùng biến turn thì có đảm bảo 4 điều kiện miễn găng không?

Bài tập 4

Xét hai tiến trình sau :

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

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

Đồng bộ hoá xử lý của hai tiến trình trên, sử dụng hai semaphore tổng quát, sao cho tại bất kỳ thời điểm nào cũng có $nb < na \leq nb + 10$

Bài tập 5

- Xét 2 tiến trình P1 và P2:
 - P1 {A1, A2}
 - P2 {B1, B2}
- Yêu cầu: A1 và B1 phải hoàn tất trước khi A2 hay B2 bắt đầu!
 - Dùng semaphore để thực hiện

Bài tập 6

```
do{
```

```
    X = X + 1;
```

```
    if (X == 20)
```

```
        X = 0;
```

```
while(1);
```

Dùng semaphore để đảm bảo $X \leq 20$.

*Chú ý: chương trình này được thực hiện bởi
2 tiến trình cùng lúc*