It.Kma

Chương 4: Process Schronization – Đồng bộ giữa các tiến trình

Nội dung

- Nhu cầu đồng bộ hóa (synchronisation)
- Vấn đề đồng bộ
- Giải pháp « **busy waiting** »
  - Biến cờ hiệu
  - Kiểm tra luân phiên
  - Peterson
  - Cấm ngắt
  - Test and Set
- Các giải pháp « **SLEEP and WAKEUP** »
  - Semaphore
  - Monitors

1. Nhu cầu đồng bộ hóa

- Trong hệ thống, nhiều tiến trình liên lạc với nhau
- HĐH luôn cần cung cấp những cơ chế đồng bộ hóa để bảo đảm hoạt động đồng thời của các tiến trình không tác động sai lệch đến nhau
- Việc tác động sai lệch do:
  - Yêu cầu độc quyền truy xuất
  - Yêu cầu phối hợp

1.1. Yêu cầu độc quyền truy xuất

- Tài nguyên trong hệ thống phân 2 loại:
  - Tài nguyên chia sẻ: cho phép nhiều tiến trình đồng thời truy xuất
  - Tài nguyên không thể chia sẻ: tại một thời điểm chỉ có một tiến trình sử dụng
- Không thể chia sẻ do:
  - Đặc điểm phần cứng
  - Nhiều tiến trình đồng thời sử dụng tài nguyên này sẽ gây ra kết quả không dự đoán trước được
- Giải pháp:
  - HĐH cần đảm bảo vấn đề độc quyền truy xuất tài nguyên: tại một thời điểm chỉ cho phép một tiến trình sử dụng tài nguyên

1.2. Yêu cầu phối hợp đồng bộ

- Các tiến trình trong hệ thống hoạt động độc lập, thường không đồng bộ
- Khi có nhiều tiến trình phối hợp hoàn thành một tác vụ có thể dẫn đến yêu cầu đồng bộ:
  - Tiến trình này sử dụng kết quả của tiến trình kia
  - Cần hoàn thiện các tiến trình con mới có thể hoàn thiện tiến trình cha

2. Vấn đề đồng bộ(1)

❖ Bài toán 1: Producer - Consumer

Producer

```
while(1){
  /*produce an item */
  while(Counter==BUFFER_SIZE);
  Buffer[IN] = nextProduced;
  IN = (IN+1)%BUFFER_SIZE;
  Counter++;
}
```

Consumer

```
while(1){
  while(Counter == 0);
  nextConsumed = Buffer[OUT];
  OUT=(OUT+1)%BUFFER_SIZE;
  Counter--;
  /*consume the item*/
}
```

Nhận xét

- *Producer* sản xuất một sản phẩm
- *Consumer* tiêu thụ một sản phẩm

⇒Số sản phẩm còn trong **Buffer** không thay đổi

2. Vấn đề đồng bộ(2)

Counter++

```
Load R1, Counter
Inc R1
Store Counter, R1
```

Counter−−

```
Load R2, Counter
Dec R2
Store Counter, R2
```

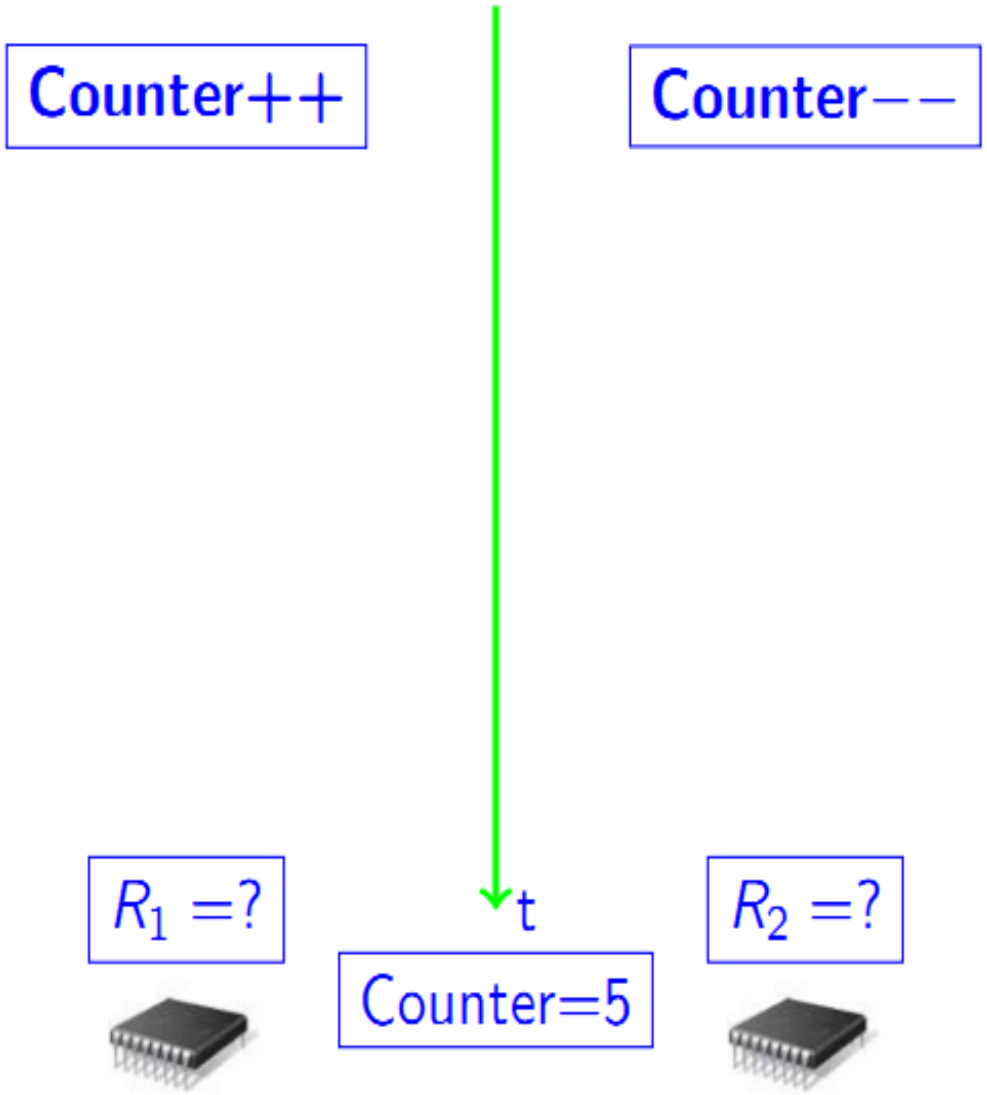
2. Vấn đề đồng bộ(3)

Counter++

```
Load R1, Counter
Inc R1
Store Counter, R1
```

Counter−−

```
Load R2, Counter
Dec R2
Store Counter, R2
```



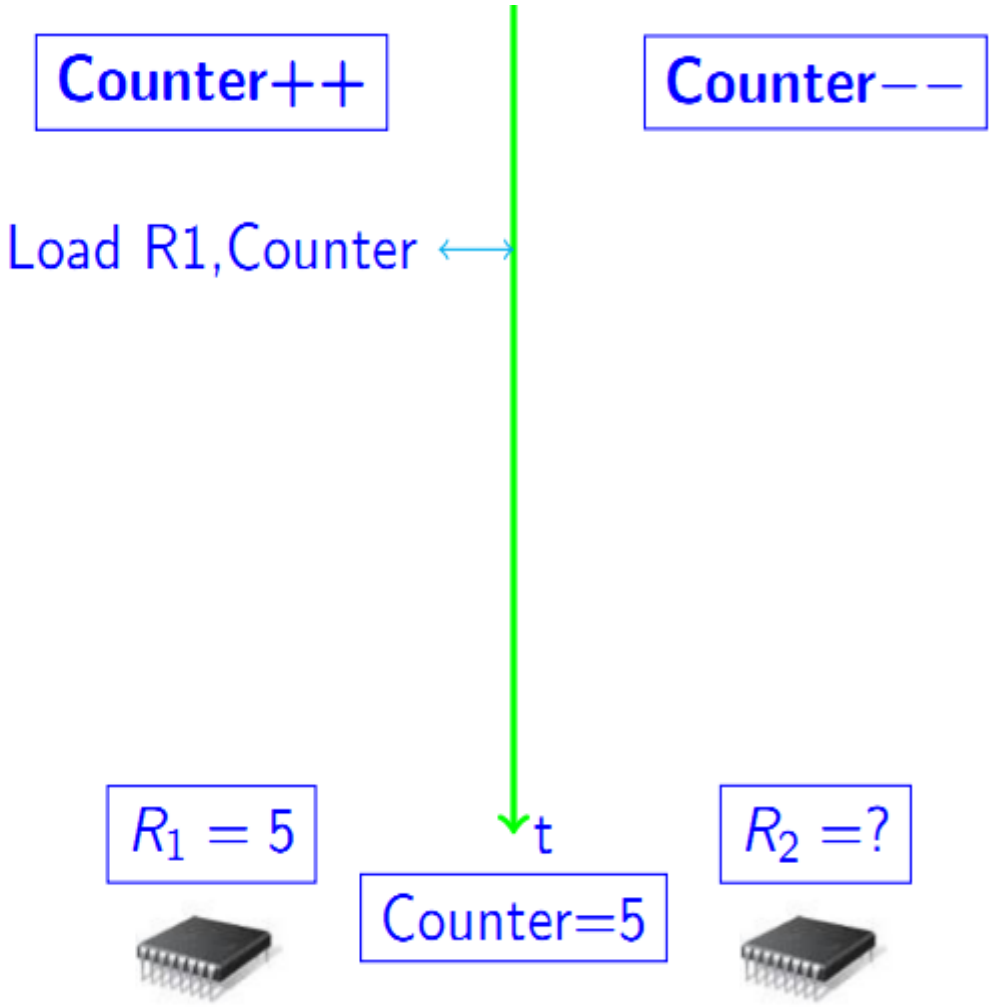
2. Vấn đề đồng bộ(4)

Counter++

```
Load R1, Counter
Inc R1
Store Counter, R1
```

Counter−−

```
Load R2, Counter
Dec R2
Store Counter, R2
```



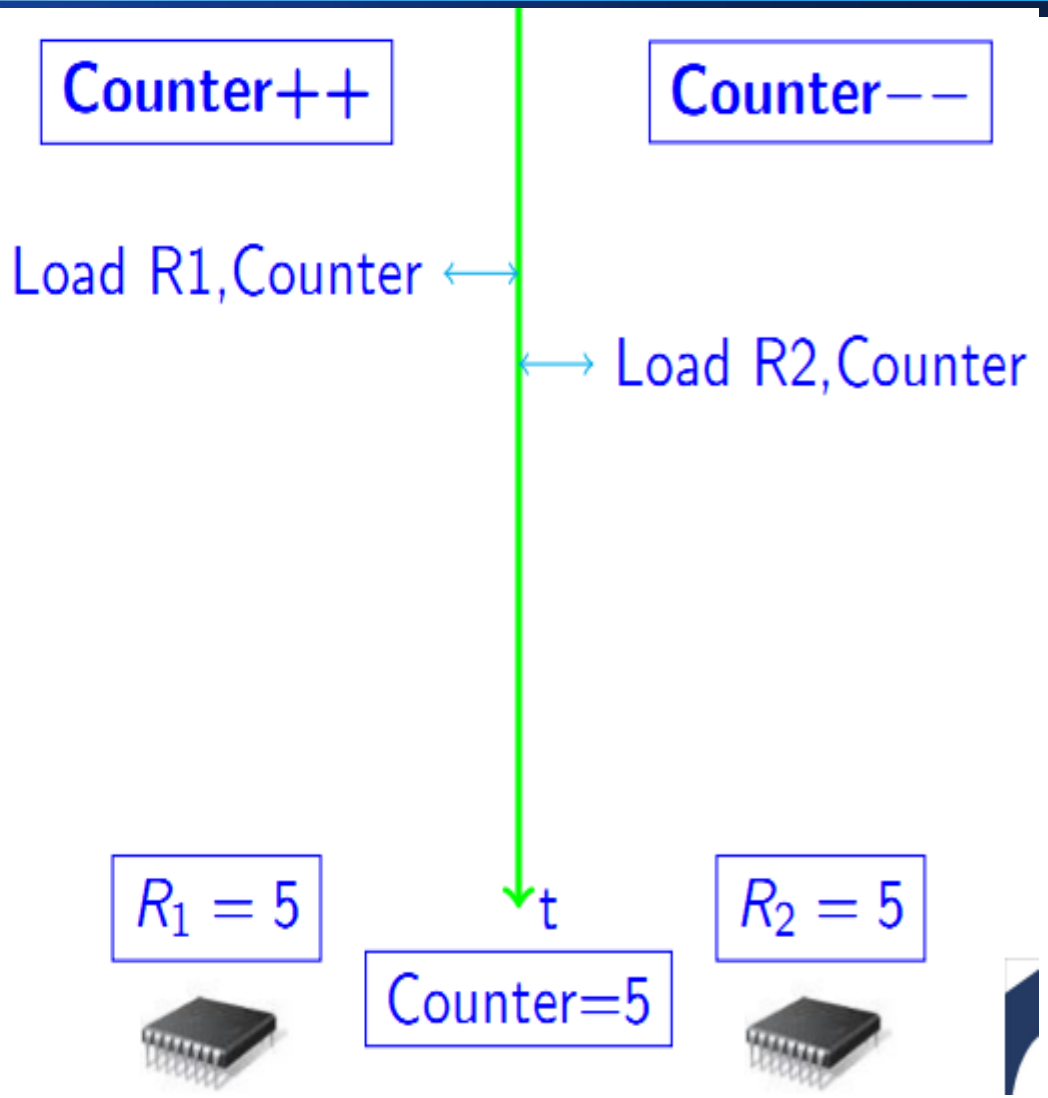
2. Vấn đề đồng bộ(5)

Counter++

Load R1, Counter
Inc R1
Store Counter, R1

Counter−−

Load R2, Counter
Dec R2
Store Counter, R2



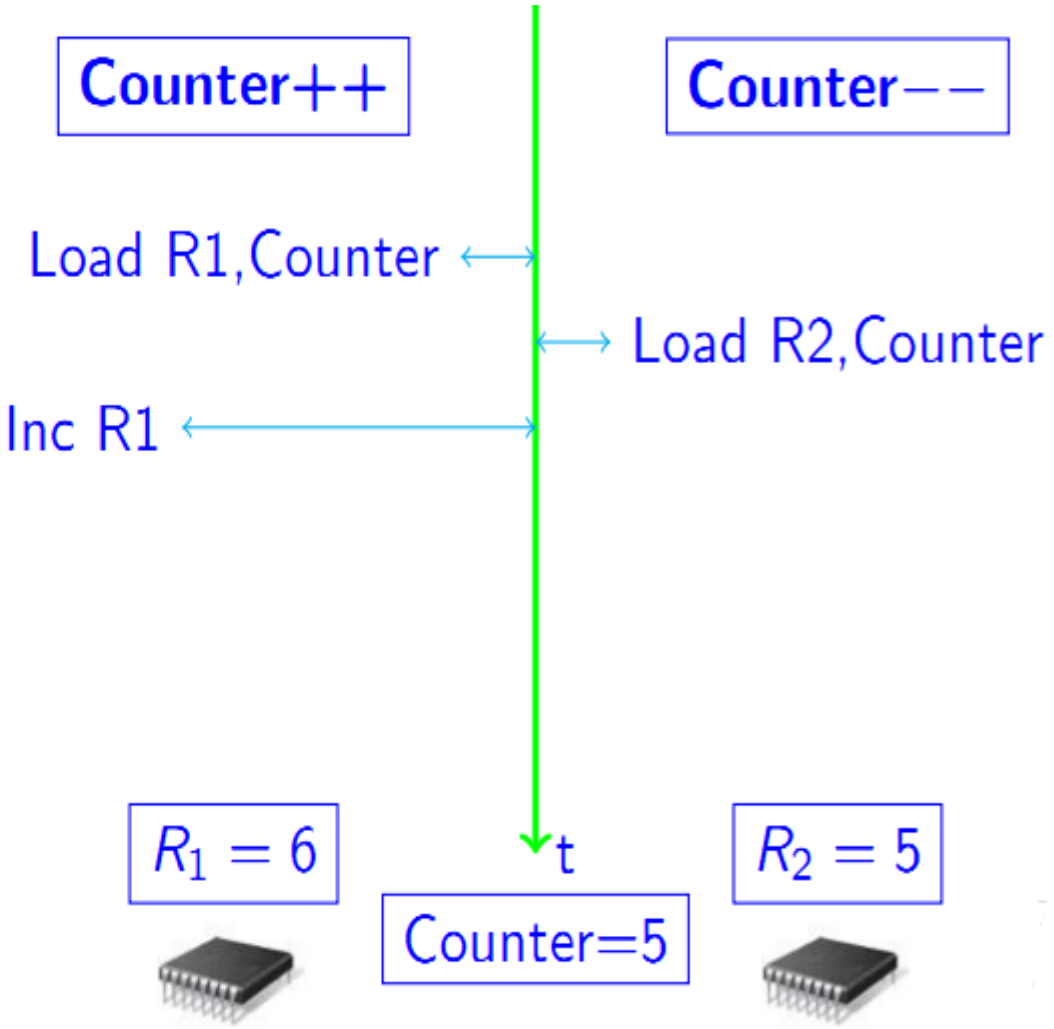
2. Vấn đề đồng bộ(6)

Counter++

```
Load R1, Counter
Inc R1
Store Counter, R1
```

Counter−−

```
Load R2, Counter
Dec R2
Store Counter, R2
```



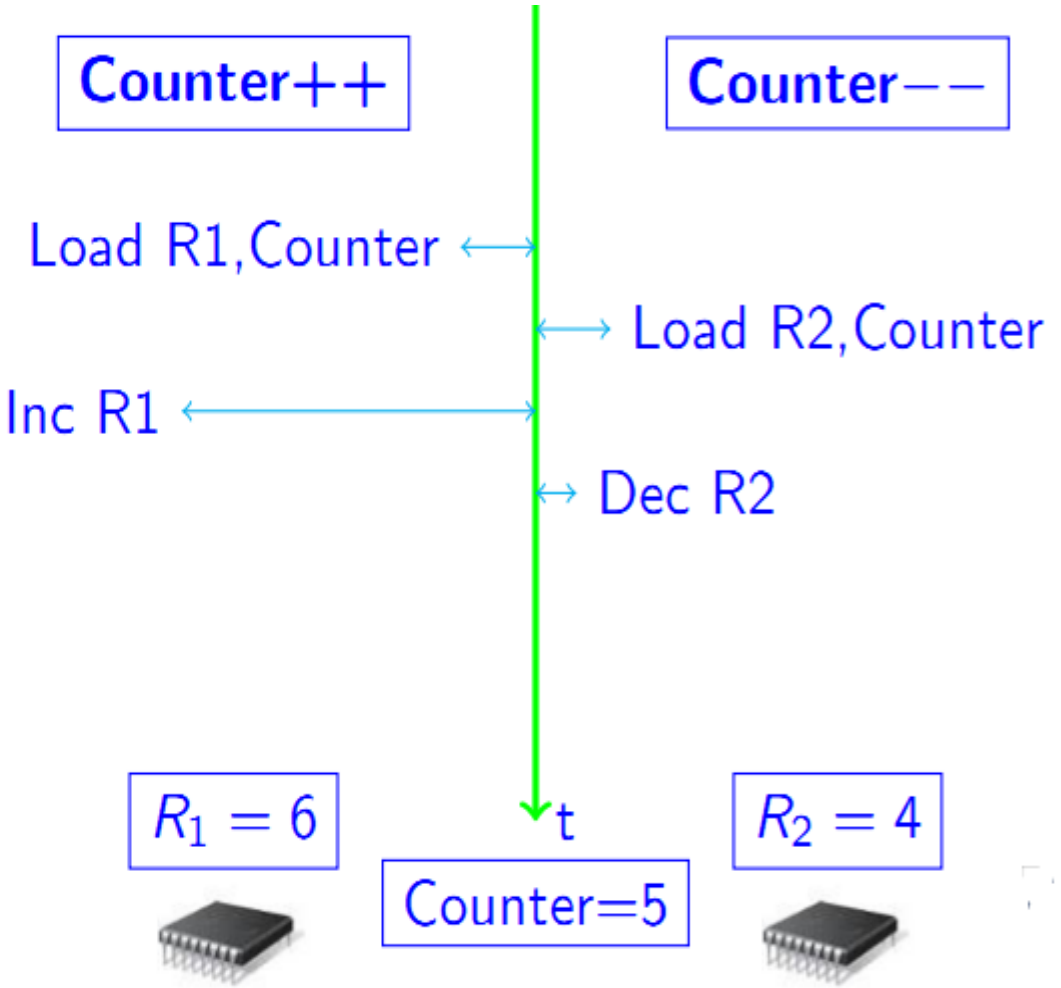
2. Vấn đề đồng bộ(7)

Counter++

Load R1, Counter
Inc R1
Store Counter, R1

Counter−−

Load R2, Counter
Dec R2
Store Counter, R2



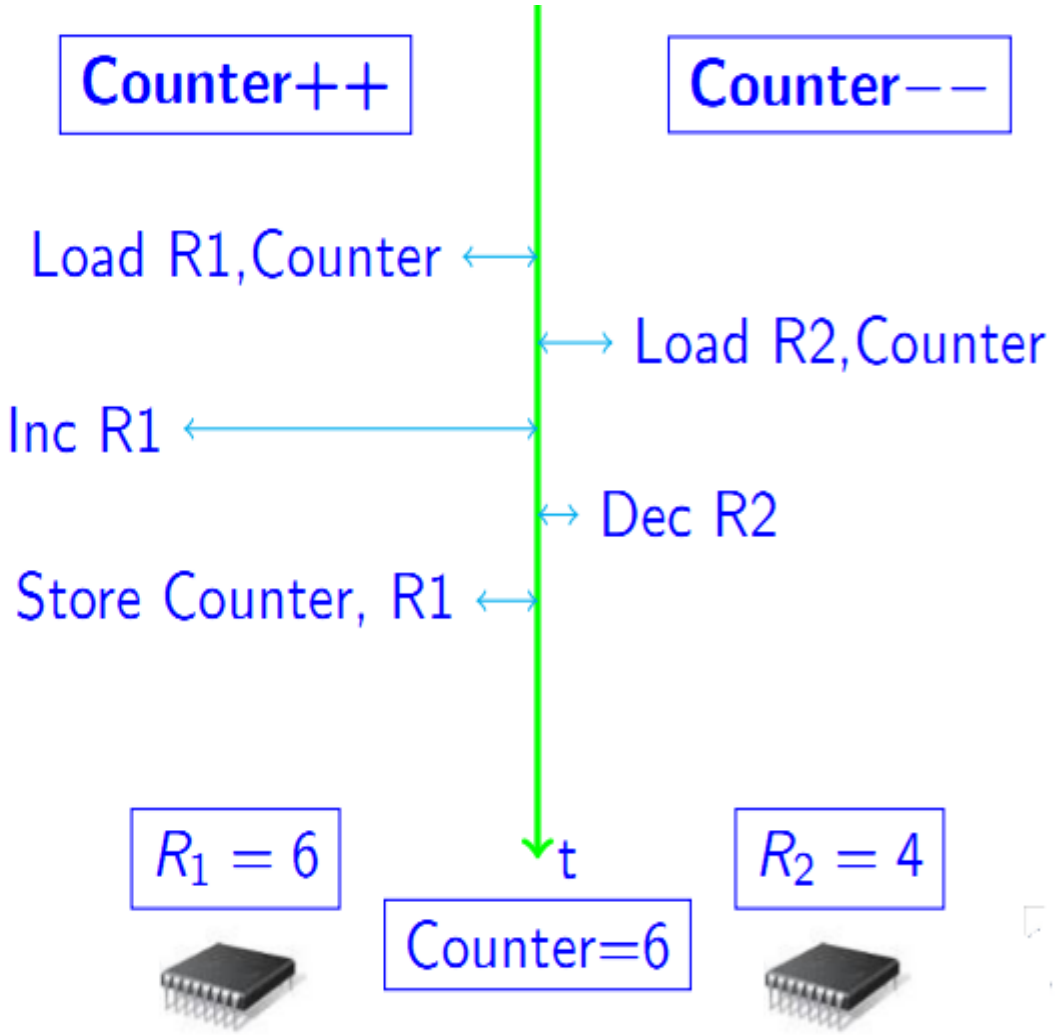
2. Vấn đề đồng bộ(8)

Counter++

Load R1, Counter
Inc R1
Store Counter, R1

Counter−−

Load R2, Counter
Dec R2
Store Counter, R2



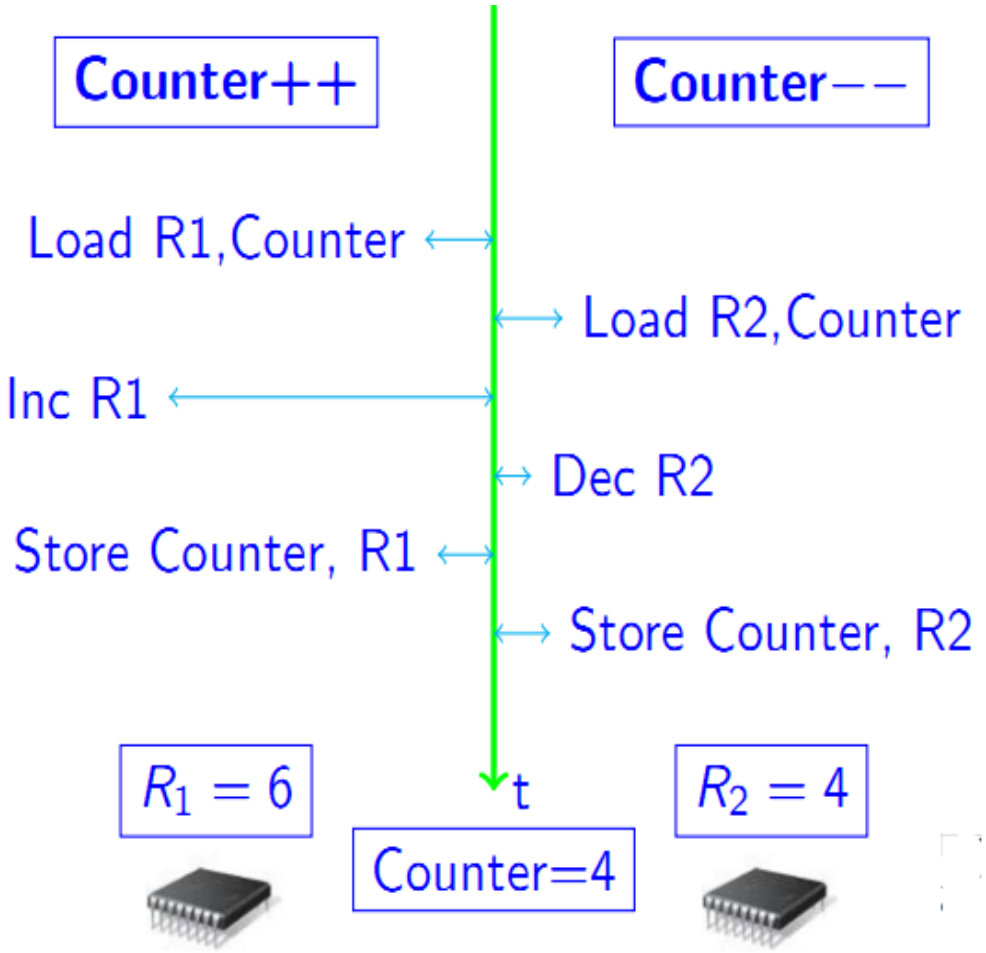
2. Vấn đề đồng bộ(9)

Counter++

Load R1, Counter
Inc R1
Store Counter, R1

Counter−−

Load R2, Counter
Dec R2
Store Counter, R2



2. Vấn đề đồng bộ(10)

- Bài toán 2:
  - Khách hàng có tài khoản 800K, cần thực hiện yêu cầu rút 400K. Việc thực hiện yêu cầu thông qua tiến trình P1
  - Ở một vị trí khác, Hacker có được mật khẩu của khách hàng, truy nhập vào tài khoản khách hàng yêu cầu rút 700K. Việc thực hiện yêu cầu hacker thông qua tiến trình P2
  - Cả P1, P2 đều truy nhập vào biến dùng chung là **account** của khách hàng; mỗi tiến trình rút tiền có biến **require**(số tiền cần rút)
  - Cả 2 tiến trình P1, P2 đều có đoạn rút tiền và cập nhật tài khoản:

```
if (account >= require)
  account -= require;
else
  printf("Error");
```

2. Vấn đề đồng bộ(11)

- Bài toán 2(tiếp)
  - Tình huống nảy sinh:
    - P1 kiểm tra thấy **account > require** nên thực hiện đoạn code trên để rút tiền nhưng chưa cập nhật tài khoản(chưa thực hiện lệnh **account -= require**) do hết thời gian sử dụng CPU được phân phối cho nó
    - P2 kiểm tra điều kiện vẫn thỏa mãn(**account** vẫn = 800K) nên nó thực hiện việc rút tiền. Giả sử P2 được phân phối đủ thời gian sử dụng CPU, cập nhật **account** = 100K và kết thúc
    - P1 quay lại thực hiện(vì đã kiểm tra điều kiện từ lần trước) nốt công việc và cập nhật **account** = -300 => tình huống lỗi
  - Giải pháp:
    - Áp dụng cơ chế ***truy xuất độc quyền*** trên tài nguyên đó (**account**): khi một tiến trình đang sử dụng tài nguyên thì các tiến trình khác không được sử dụng

2. Vấn đề đồng bộ(12): đoạn găng

- Critical session(- *critical region* đoạn găng, đoạn găng)
- Trong ví dụ trên, tiến trình P1, P2 đều bao gồm chuỗi các lệnh riêng và các lệnh thực hiện rút tiền:

…//các lệnh kết nối, lệnh kiểm tra

```
if (account >= require)
   account -= require;
   print('drawed money');
else
   printf("Error");
// các lệnh kết thúc tiến trình….
```

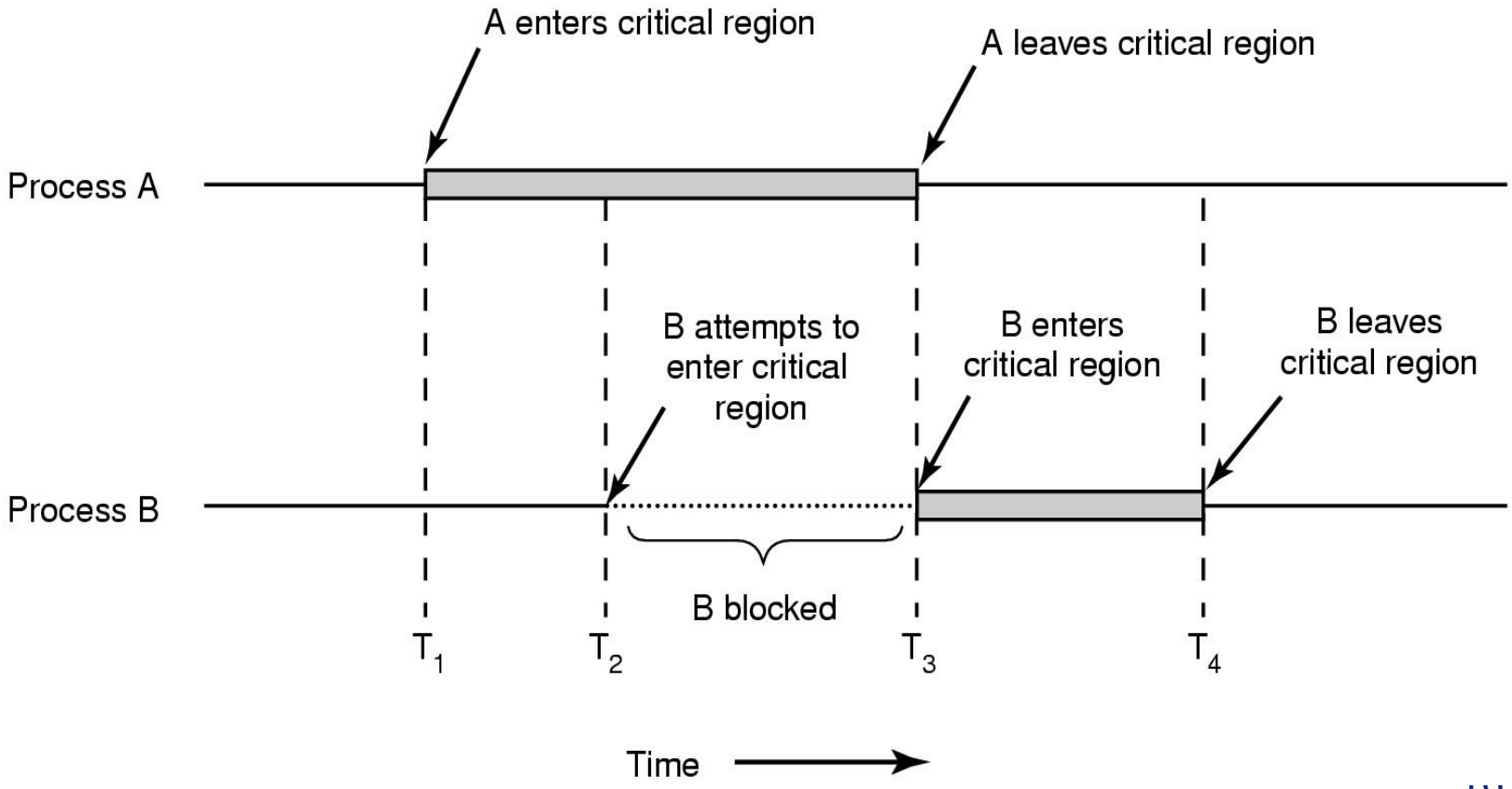
Đoạn lệnh thao tác tài nguyên Chung, có thể xảy ra mâu thuẫn => **Đoạn găng**

2. Vấn đề đồng bộ(13): đoạn găng

- Khái niệm **đoạn găng(miền găng):**
  - là đoạn mã lệnh có khả năng xảy ra mâu thuẫn khi truy xuất tài nguyên chung
- HĐH cần cài đặt các giải pháp đồng bộ để giải quyết vấn đề độc quyền truy xuất
  - Quyết định tiến trình nào sẽ được vào đoạn găng(thực hiện các lệnh trong miền này)
  - Khi một tiến trình đang ở đoạn găng thì các tiến trình khác không thể vào đoạn găng

2. Vấn đề đồng bộ(14): đoạn găng

❖ Minh họa dòng thời gian của đoạn găng:



3. Giải pháp « **busy waiting** »

- Giải pháp phần mềm:
  - Sử dụng các biến cờ hiệu
  - Sử dụng việc kiểm tra luân phiên
  - Giải pháp của Peterson
- Giải pháp có sự hỗ trợ phần cứng:
  - Cấm ngắt
  - Test & Set

3.1. Sử dụng các biến cờ hiệu (Simaphore)(1)

- Ý tưởng:
  - Các tiến trình **chia sẻ một biến chung đóng vai trò « chốt cửa » (lock)** , biến này được **khởi động là 0**.
  - Một tiến trình muốn vào đoạn găng, trước tiên phải kiểm tra giá trị của biến **lock**. Nếu **lock = 0**, tiến trình đặt lại giá trị cho **lock = 1** và đi vào đoạn găng. Nếu **lock đang nhận giá trị** 1, tiến trình phải chờ đến khi **lock có giá trị 0**.
- Như vậy giá trị 0 của lock mang ý nghĩa là không có tiến trình nào đang ở trong đoạn găng, và lock=1 khi có một tiến trình đang ở trong đoạn găng.

```
while (TRUE) {
    while (lock == 1); { // đợi lock = 0}
    //trường hợp lock == 0
    lock = 1; //đặt lock = 1 để cấm các tiến trình khác
    critical-section (); //thực hiện đoạn găng
    lock = 0; //kết thúc đoạn găng phải đặt lock = 0 để giải phóng tài nguyên
    Noncritical-section (); // thực hiện các lệnh bên ngoài đoạn găng
}
```

3.1. Sử dụng các biến cờ hiệu (Simaphore)(2)

❖Nhận xét:

- Giải pháp này vẫn xảy ra trường hợp 2 tiến trình cùng trong đoạn găng khi:
  - Lock==0, Tiến trình P1 vào đoạn găng nhưng chưa kịp đặt lock=1 vì hết thời gian sử dụng CPU
  - P2 kiểm tra thấy lock==0, đặt lock=1 và đang thực hiện các lệnh trong đoạn găng nhưng chưa xong vì hết thời gian CPU
  - P1 được phân phối CPU và thực hiện các lệnh trong đoạn găng

  ⇒P1, P2 cùng trong đoạn găng

3.2. Sử dụng việc kiểm tra luân phiên(1)

- Ý tưởng:
  - Giải pháp **đề nghị cho hai tiến trình**.
  - Hai tiến trình **sử dụng chung biến *turn***
    - Khởi động với giá trị 0.
    - Nếu *turn = 0*, tiến trình P1 được vào đoạn găng, *turn*= 1 P2 được vào đoạn găng.
    - Nếu *turn = 1*, tiến trình P1 đi vào một vòng lặp chờ đến khi *turn* nhận giá trị *0*.
    - Khi tiến trình P1 rời khỏi đoạn găng, nó đặt giá trị *turn* về 1 để cho phép tiến trình P2 đi vào đoạn găng.

3.2. Sử dụng việc kiểm tra luân phiên(2)

It.kma

- Cấu trúc tiến trình P1

```
while (TRUE) {
    while (turn != 0); {// wait}
    critical-section ();//thực hiện đoạn găng xong mới đặt turn=1
    turn = 1;
    Noncritical-section ();
}
```

- Cấu trúc tiến trình P2

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

3.2. Sử dụng việc kiểm tra luân phiên(3)

❖Nhận xét:

- Ngăn được trường hợp 2 tiến trình đồng thời trong đoạn găng
- Xảy ra tình huống một tiến trình bị ngăn vào đoạn găng bởi một tiến trình bên ngoài đoạn găng khi:
  - Turn=0, P1 vào đoạn găng xong, đặt **turn=1** rồi ra và muốn nhanh chóng quay lại đoạn găng lần nữa
  - Nhưng P2 vẫn thực hiện các lệnh bên ngoài đoạn găng với lượng rất lớn nên thời gian thực hiện rất lâu không thể vào đoạn găng ngay được. Do đó turn vẫn = 1 và P1 không thể vào
- Số lần vào đoạn găng của P1, P2 là cân bằng(luân phiên nhau) do đó gặp vấn đề khi P1 cần vào đoạn găng liên tục còn P2 không cần thiết lắm.

3.3. Giải pháp Peterson(1)

- Do Peterson đề nghị
- Ý tưởng: kết hợp 2 giải pháp trên
  - P0, P1 sử dụng 2 biến chung *turn*, *interesse[2]*
    - *Turn* = 0 đến phiên P0, turn=1 đến phiên P1
    - *Interesse[i]*=TRUE, Pi muốn vào đoạn găng
  - Khởi tạo:
    - Interesse[0]=Interesse[1]=false; turn = 0 hoặc 1
  - Để có thể vào được đoạn găng:
    - Pi đặt giá trị *interesse[i]*=TRUE
    - Sau đó đặt turn=*j* (đề nghị thử tiến trình khác vào đoạn găng).
    - Nếu tiến trình Pj không quan tâm đến việc vào đoạn găng (*interesse[j]=FALSE*), thì Pi có thể vào đoạn găng, nếu không, Pi phải chờ đến khi *interesse[j]=FALSE*.
    - Khi tiến trình Pi rời khỏi đoạn găng, nó đặt lại giá trị cho *interesse[i]*= FALSE.

3.3. Giải pháp Peterson()

- Cấu trúc tiến trình Pi//gia su: p0 muon vao, p1 ko

```
//ban  dau:  interese[1]=interese[0]=false;  turn=0

while (TRUE) {
        interesse[i]= TRUE;//interesse[0]=true
        int j = 1-i; // j là tiến trình còn lại
            turn = j;
        while (turn == j && interesse[j]==TRUE);{//wait}
        critical-section ();
        interesse[i] = FALSE;
        Noncritical-section ();
    }
```

3.3. Giải pháp Peterson()

❖Nhận xét:

- Ngăn chặn được tình trạng mâu thuẫn truy xuất:
  - Pi chỉ có thể vào đoạn găng khi *interesse[j]=FALSE* hoặc *turn = i*.
  - Nếu cả hai tiến trình đều muốn vào đoạn găng thì *interesse[i] = interesse[j] =TRUE* nhưng giá trị của *turn* chỉ có thể hoặc là 0 hoặc là 1, do vậy chỉ có một tiến trình được vào đoạn găng

3.3. Giải pháp Peterson()

❖Nhận xét:

- Ngăn chặn được tình trạng mâu thuẫn truy xuất:
  - Pi chỉ có thể vào đoạn găng khi *interesse[j]=FALSE* hoặc *turn = i*.
  - Nếu cả hai tiến trình đều muốn vào đoạn găng thì *interesse[i] = interesse[j] =TRUE* nhưng giá trị của *turn* chỉ có thể hoặc là 0 hoặc là 1, do vậy chỉ có một tiến trình được vào đoạn găng

Khi có nhiều tiến trình muốn vào đoạn găng???

3.4. Cấm ngắt

- Ý tưởng:
  - *Cho phép tiến trình cấm tất cả các ngắt trước khi vào đoạn găng, và phục hồi ngắt khi ra khỏi đoạn găng.*
  - Khi đó, ngắt đồng hồ cũng không xảy ra, do vậy hệ thống không thể tạm dừng hoạt động của tiến trình đang xử lý để cấp phát CPU cho tiến trình khác, nhờ đó tiến trình hiện hành yên tâm thao tác trên đoạn găng mà không sợ bị tiến trình nào khác tranh chấp.

3.4. Cấm ngắt

- Ý tưởng:
  - *Cho phép tiến trình cấm tất cả các ngắt trước khi vào đoạn găng, và phục hồi ngắt khi ra khỏi đoạn găng.*
  - Khi đó, ngắt đồng hồ cũng không xảy ra, do vậy hệ thống không thể tạm dừng hoạt động của tiến trình đang xử lý để cấp phát CPU cho tiến trình khác, nhờ đó tiến trình hiện hành yên tâm thao tác trên đoạn găng mà không sợ bị tiến trình nào khác tranh chấp.
- Nhận xét:
  - Không được ưa chuộng vì rất thiếu thận trọng khi cho phép tiến trình người dùng được phép thực hiện lệnh cấm ngắt.
  - Hệ thống có nhiều CPU, lệnh cấm ngắt chỉ có tác dụng trên CPU đang xử lý tiến trình hiện tại, các tiến trình hoạt động trên các CPU khác vẫn có thể truy xuất đến đoạn găng

3.5. Test & Set(1)

- Giải pháp:
  - Tập lệnh máy có thêm một **chỉ thị** đặc biệt cho phép **kiểm tra** và **cập nhật** nội dung một vùng nhớ trong một **thao tác đơn vị**, gọi là chỉ thị ***Test-and-Set Lock*** **(TSL)**
  - Định nghĩa:

```
Test-and-Setlock(boolean  &target)
{  boolean temp = target;
   target = TRUE;//thiết lập giá trị mới = True để khóa
   return temp;//lấy giá trị cũ để kiểm tra
  }
```

3.5. Test & Set(2)

- Nếu có hai chỉ thị TSL xử lý đồng thời (trên hai CPU khác nhau), chúng sẽ được xử lý tuần tự.
- Có thể cài đặt giải pháp truy xuất độc quyền với TSL bằng cách sử dụng thêm một biến chung lock, được khởi tạo là FALSE. Tiến trình phải kiểm tra giá trị của biến lock trước khi vào đoạn găng, nếu lock = FALSE, tiến trình có thể vào đoạn găng.
- Cấu trúc một ctr sử dụng giải pháp TSL:

```
while (TRUE) {
    while (Test-and-Setlock(lock)){//wait}
    critical-section ();
    lock = FALSE;//ra khỏi đoạn găng, lock=False(không khóa)
    Noncritical-section ();
}
```

3.5. Test & Set(3)

- Nhận xét:
  - Giảm nhẹ công việc lập trình nhưng việc cài đặt TSL như một lệnh máy không đơn giản
  - Khi có nhiều CPU, việc điều phối thực hiện TSL cho từng CPU gặp khó khăn.

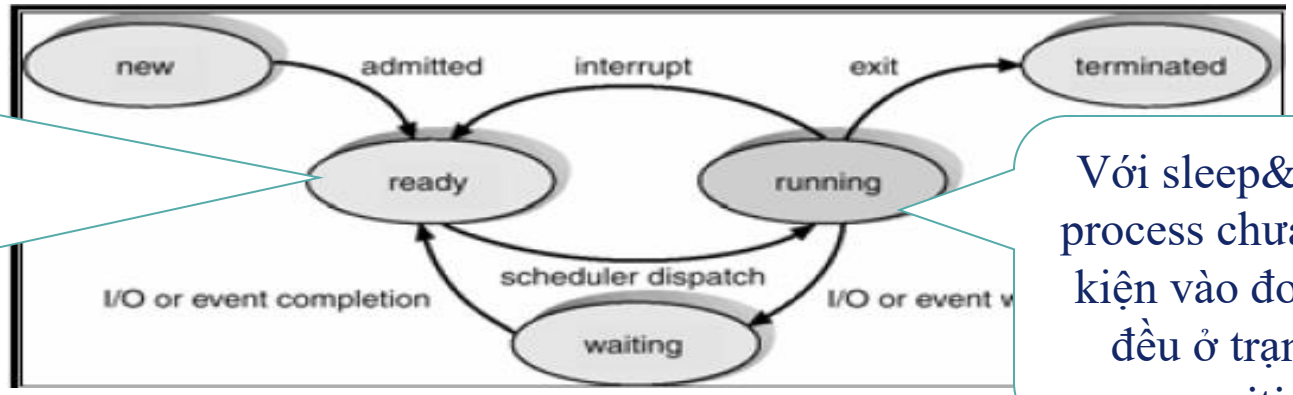
3.6. Kết luận về giải pháp **Busy waiting**

- Hoạt động chung:
  - Tất cả các giải pháp trên đều phải **thực hiện một vòng lặp để kiểm tra** xem có được phép vào đoạn găng.
  - Nếu điều kiện chưa cho phép, 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 đoạn găng như thế được gọi các giải pháp « *busy waiting* »-"*Bận* ***vì chờ***".
- Hạn chế:
  - Việc kiểm tra như thế tiêu thụ rất nhiều thời gian sử dụng CPU, do vậy tiến trình đang chờ vẫn chiếm dụng CPU(để kiểm tra điều kiện).
- Xu hướng giải quyết vấn đề đồng bộ hoá là nên tránh các giải pháp « ***busy waiting*** »

4. Các giải pháp « SLEEP and WAKEUP »(1)

- Khắc phục nhược điểm của các giải pháp **busy waiting** bằng cách: cho một tiến trình chưa đủ điều kiện vào đoạn găng chuyển sang trạng thái **waiting**
  - Tạm khóa tiến trình không cho sử dụng CPU ngay vì tiến trình chỉ sử dụng CPU khi ở trạng thái **running**
  - Tiến trình chỉ có thể chuyển sang trạng thái **running** khi đang ở trạng thái **ready**(sẵn sàng)

Với busy waiting, process chưa đủ điều kiện vào đoạn găng đều ở trạng thái ready



Với sleep&wakeup process chưa đủ điều kiện vào đoạn găng đều ở trạng thái waiting

4. Các giải pháp « SLEEP and WAKEUP »(2)

❖Giải pháp:

- Hệ điều hành sử dụng 2 thủ tục sleep và wakeup
  - *SLEEP* là một lời gọi hệ thống có tác dụng tạm dừng hoạt động của tiến trình (chuyển sang trạng thái **waiting**) 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 tái kích hoạt* (đặt về trạng thái **ready**).
- *Ý tưởng*:
  - Khi một tiến trình chưa đủ điều kiện vào đoạn găng, nó gọi *SLEEP* để tự khóa đến khi có một tiến trình khác gọi *WAKEUP* để giải phóng cho nó.
  - Một tiến trình gọi *WAKEUP* khi ra khỏi đoạn găng để đánh thức một tiến trình đang chờ, tạo cơ hội cho tiến trình này vào đoạn găng

4. Các giải pháp « SLEEP and WAKEUP »(3)

- Cấu trúc chương trình trong giải pháp SLEEP and WAKEUP

```
int busy;   // =1 nếu đoạn găng đang bị chiếm, nếu không là 0
int blocked;   // đếm số lượng tiến trình đang bị tạm khóa
while (TRUE) {
        if (busy){
                ++blocked;// = blocked + 1;
                sleep();
        }
        else busy = 1;
                critical-section ();
        busy = 0;
        if(blocked){
                wakeup(process);//truyền vào process cần đánh thức
                blocked = blocked - 1;
        }
        Noncritical-section ();
}
```

4. Các giải pháp « SLEEP and WAKEUP »(4)

- Các giải pháp phổ biến:
  - Simaphore(Dijktra đề xuất)
  - Monitors

4.1. Kỹ thuật đèn báo (Semaphore)

- Là một biến nguyên S, khởi tạo bằng khả năng phục vụ của tài nguyên nó điều độ
  - Số tài nguyên có thể phục vụ tại một thời điểm (VD 3 máy in)
  - Số đơn vị tài nguyên có sẵn (VD 10 chỗ trống trong buffer)
- Chỉ có thể thay đổi giá trị bởi 2 thao tác cơ bản P và V
  - Thao tác P(S) (wait(S))

```
wait(S){
    while(S<=0);
        S--;
}
```

  - Thao tác V(S) (signal(S))

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

  - Các thao tác P(S) và V(S) xử lý không tách rời
- Đèn báo là công cụ điều độ tổng quát

4.1. Kỹ thuật đèn báo (Semaphore)

- Điều độ nhiều tiến trình qua đoạn găng
  - Sử dụng biến phân chia mutex kiểu Semaphore
  - Khởi tạo mutex =1
  - Thuật toán cho tiến trình Pi

```
do{
    wait(mutex);
        {Đoạn găng của tiến trình}
    signal(mutex)
        {Phần còn lại của tiến trình}
}while(1);
```

- Điều độ thứ tự thực hiện bên trong các tiến trình
  - Hai tiến trình P1 và P2 thực hiện đồng thời
    - P1 chứa lệnh S1, P2 chứa lệnh S2
    - Yêu cầu S2 thực hiện chỉ khi S1 thực hiện xong
  - Sử dụng đèn báo synch được khởi tạo giá trị 0
  - Đoạn mã cho P1 và P2

P_1	P_2
Phần đầu	Phần đầu
S_1	wait(synch)
Signal(synch)	S_2
Phần cuối	Phần cuối

4.1. Kỹ thuật đèn báo (Semaphore)
Hủy bỏ chờ đợi tích cực

- Sử dụng 2 thao tác cơ bản
  - Block(): ngừng tạm thời tiến trình đang thực hiện
  - Wakeup(P): thực hiện tiếp t.trình P dừng bởi lênh block()
- Khi tiến trình gọi P(S) và đèn báo S không dương
  - Tiến trình phải dừng bởi gọi tới câu lệnh block()
  - Lệnh block() đặt tiến trình vào hàng đợi gắn với đèn báo S
  - Hệ thống lấy lại CPU giao cho tiến trình khác (điều phối CPU)
  - Tiến trình chuyển sang trạng thái chờ đợi (waiting)
  - Tiến trình nằm trong hàng đợi đến khi tiến trình khác thực hiện thao tác V(S) trên cùng đèn báo S
- Tiến trình đưa ra lời gọi V(S)
  - Lấy một tiến trình trong hàng đợi ra (nếu có)
  - Chuyển tiến trình lấy ra từ trạng thái chờ sang trạng thái sẵn sàng và đặt lên hàng đợi sẵn sàng bởi gọi tới wakeup(P)
  - Tiến trình mới sẵn sàng có thể trưng dụng CPU từ tiến trình đang thực hiện nếu thuật toán điều phối CPU cho phép

Cài đặt đèn báo

Semaphore S

```
typedef struct{
   int value;
   struct process * Ptr;
}Semaphore;
```

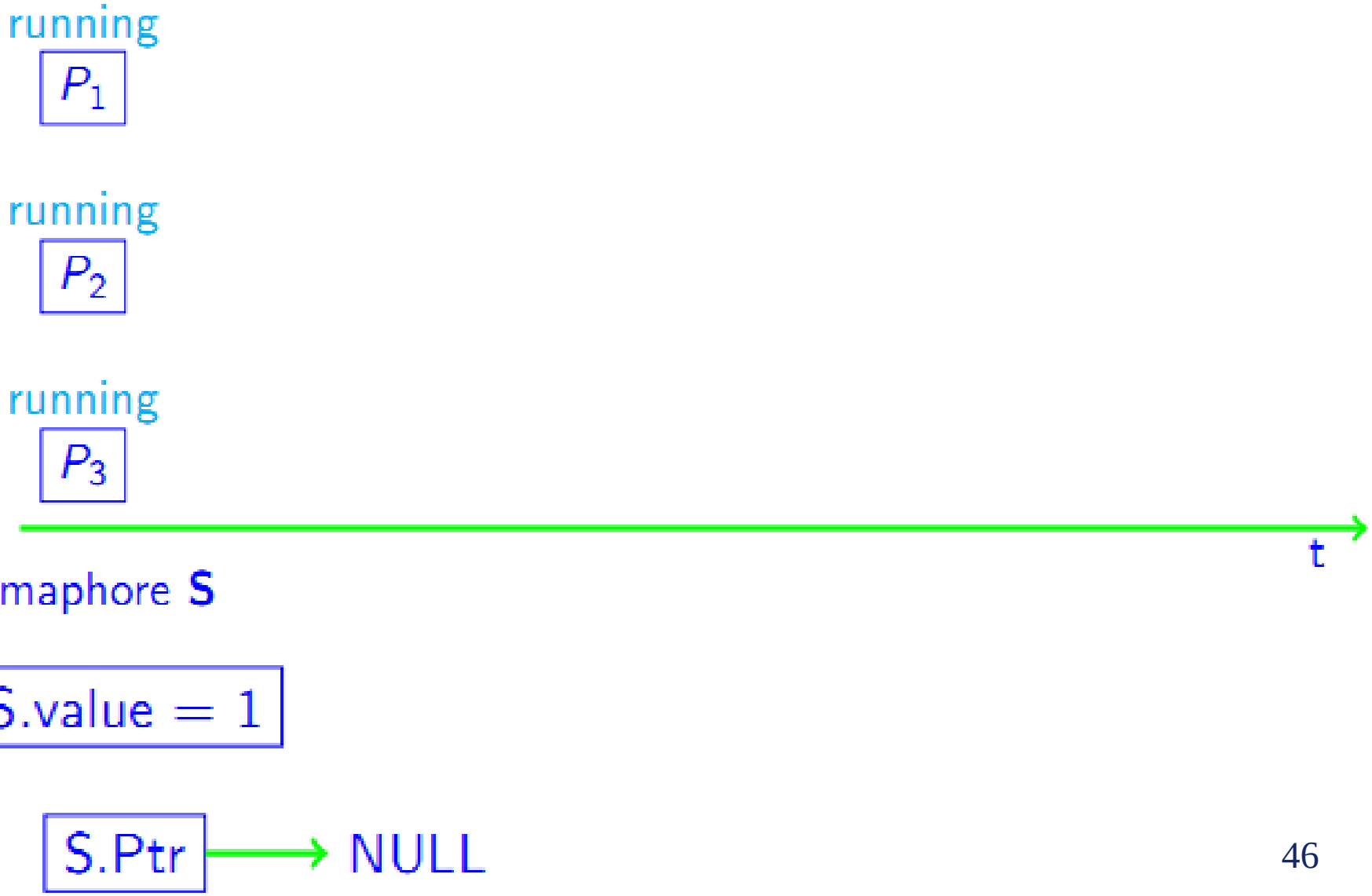
wait(S)/P(S)

```
void wait(Semaphore S) {
   S.value--;
   if(S.value < 0) {
      Thêm tiên trình vào S.Ptr
      block();
    }
}
```

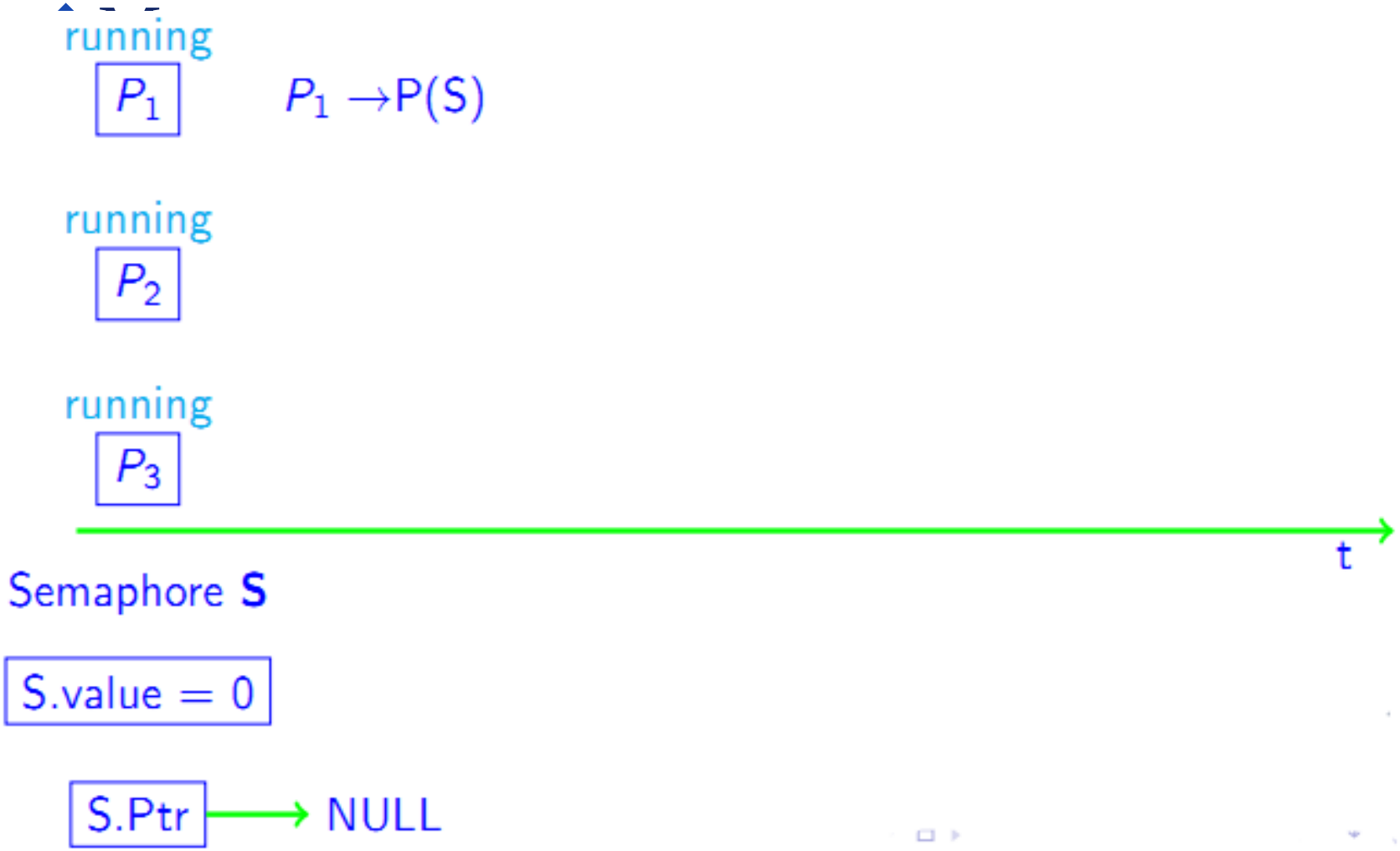
signal(S)/V(S)

```
void signal(Semaphore S) {
 S.value++;
 if(S.value ≤ 0) {
   Lấy ra tiến trình P từ S.Ptr
   wakeup(P);
 }
}
```

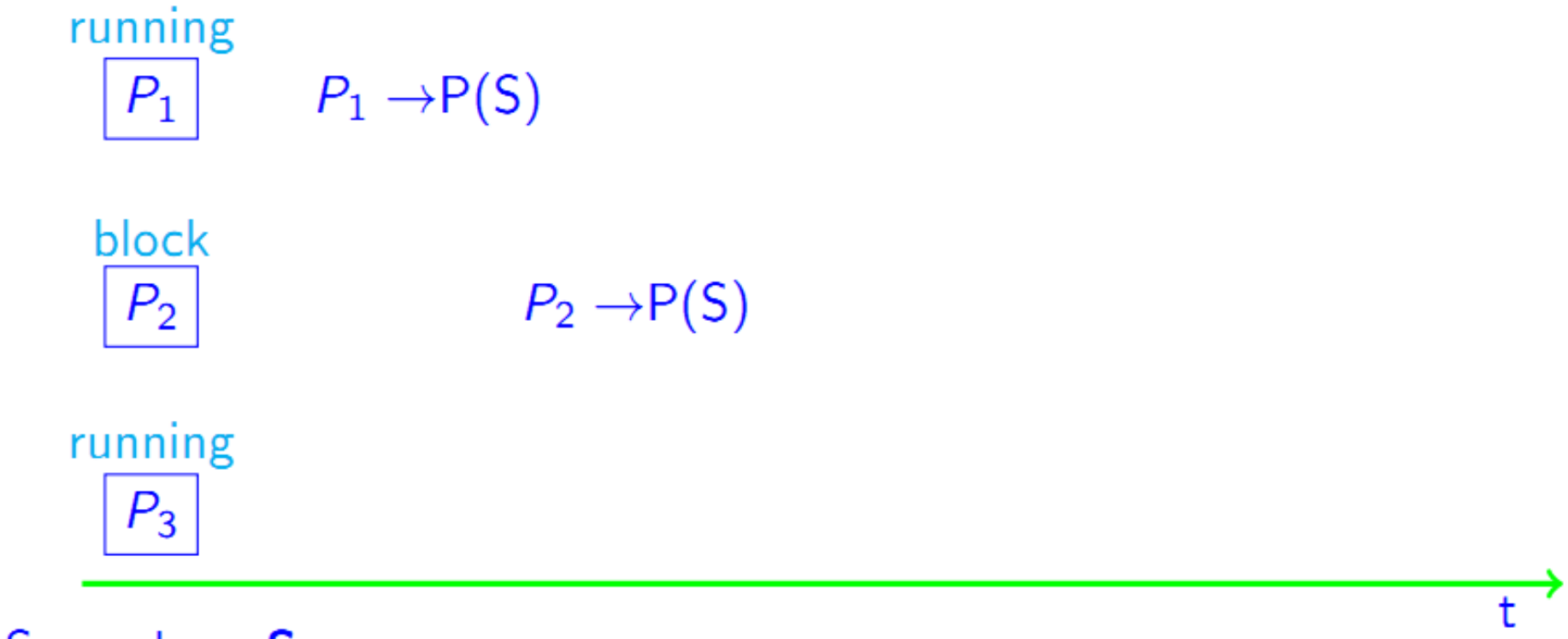
4.1. Kỹ thuật đèn báo (Semaphore)
Ví dụ điều độ



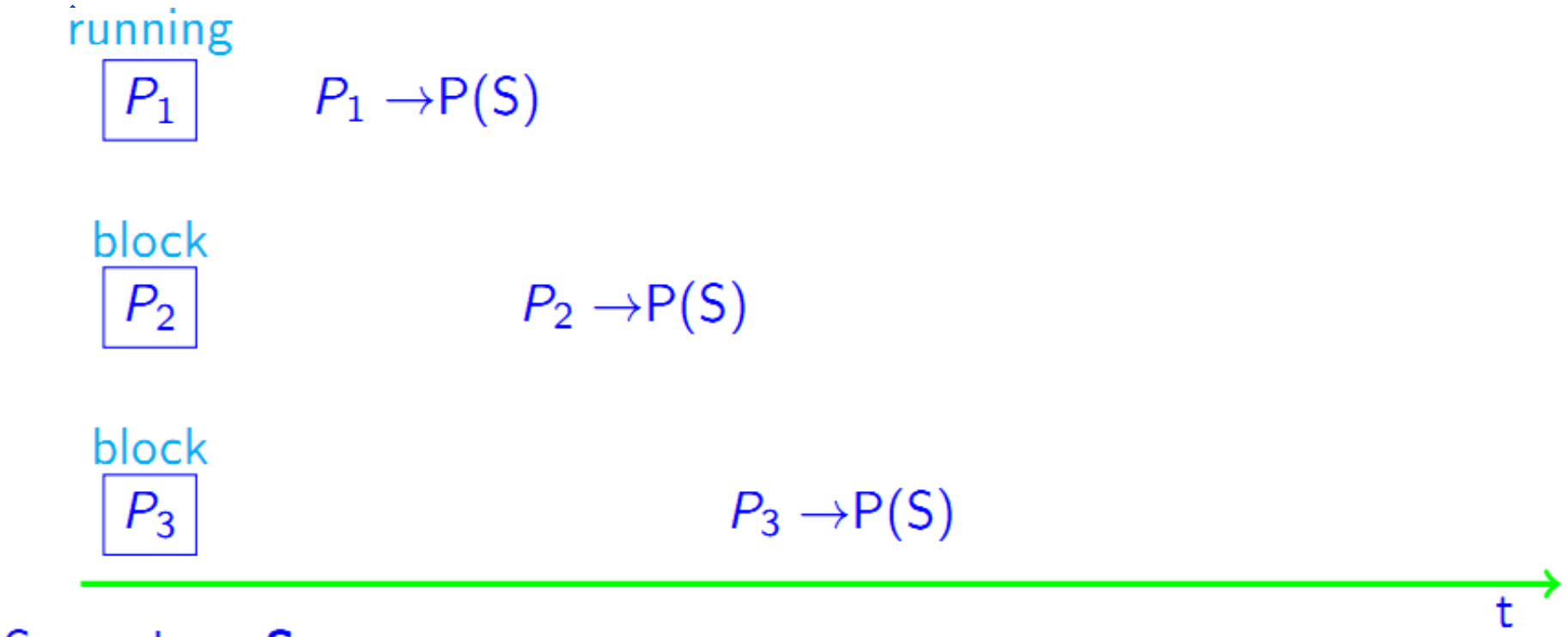
4.1. Kỹ thuật đèn báo (Semaphore)



4.1. Kỹ thuật đèn báo (Semaphore)



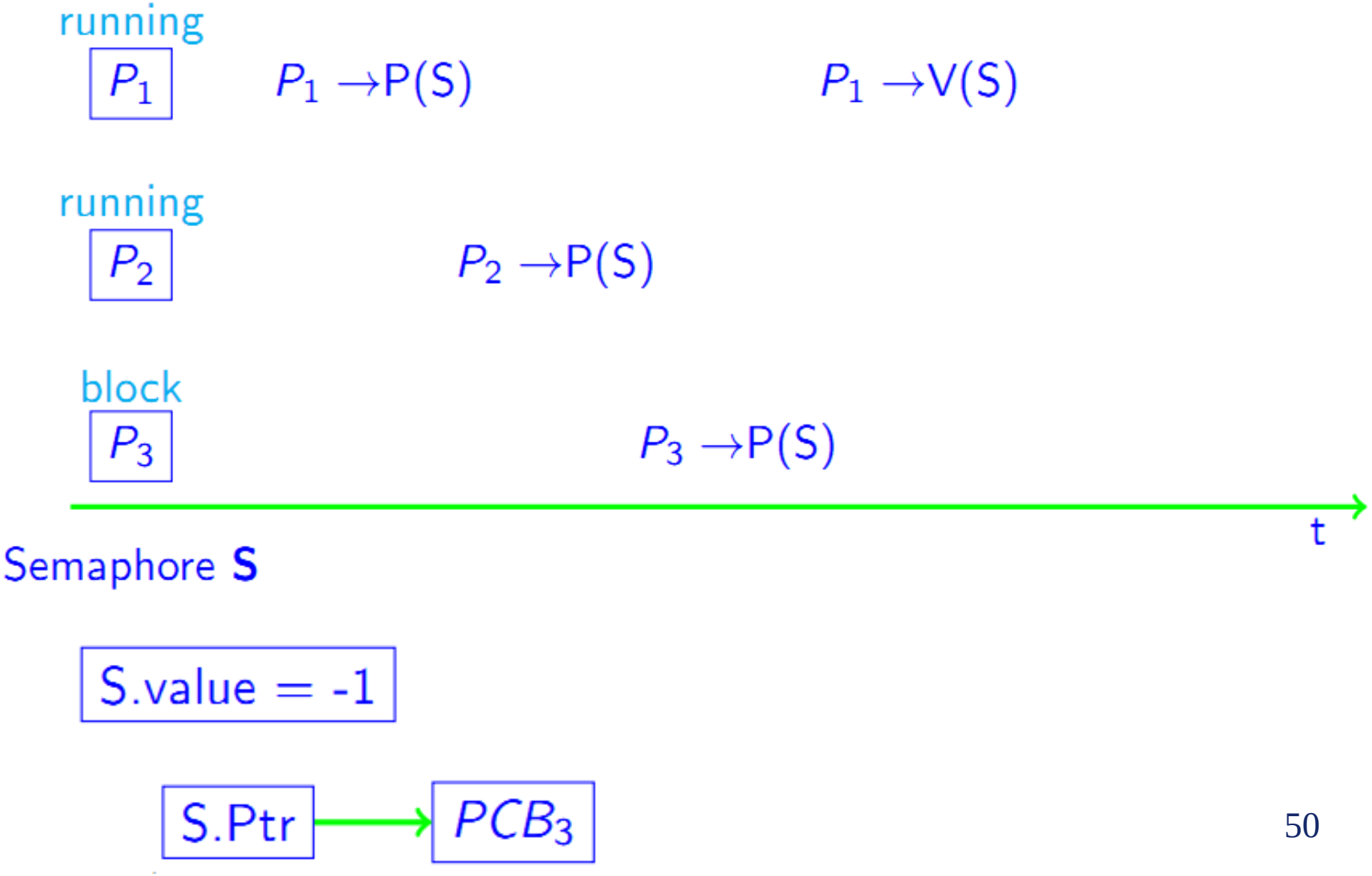
4.1. Kỹ thuật đèn báo (Semaphore)



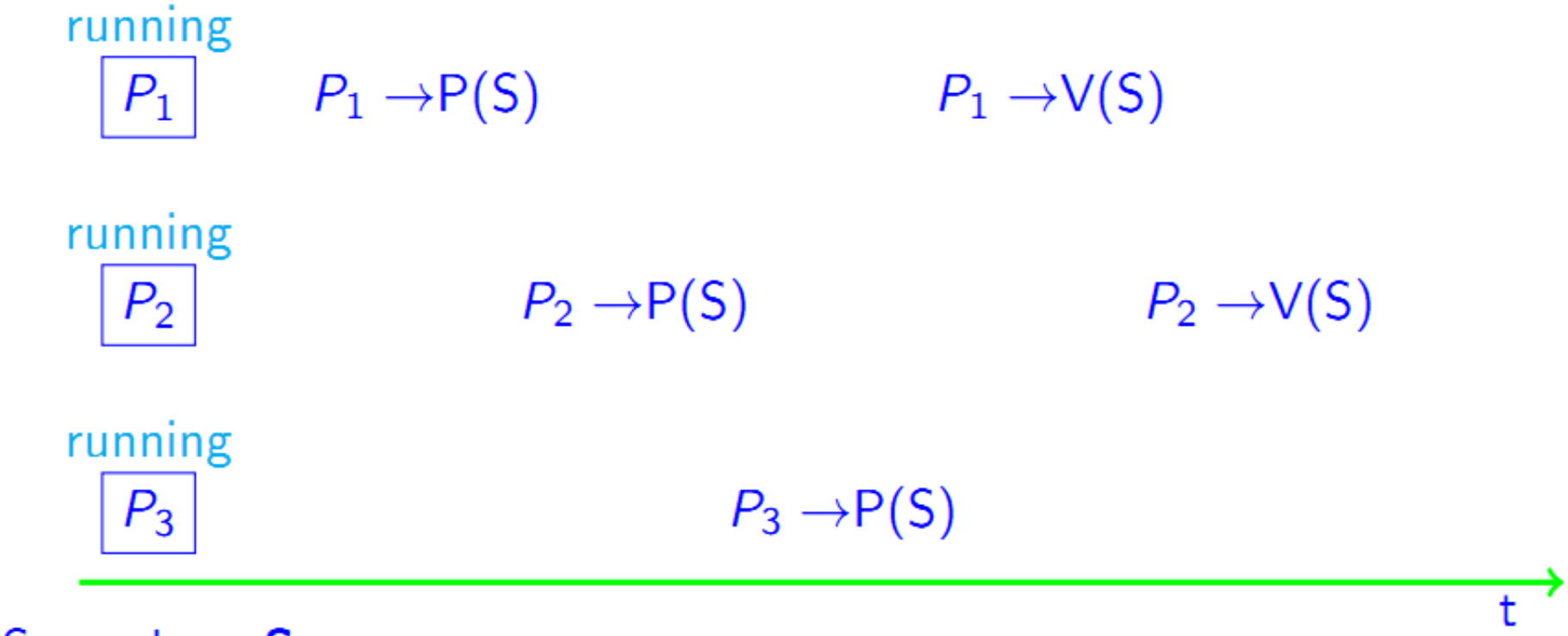
Semaphore **S**

S.value = -2

4.1. Kỹ thuật đèn báo (Semaphore)



4.1. Kỹ thuật đèn báo (Semaphore)

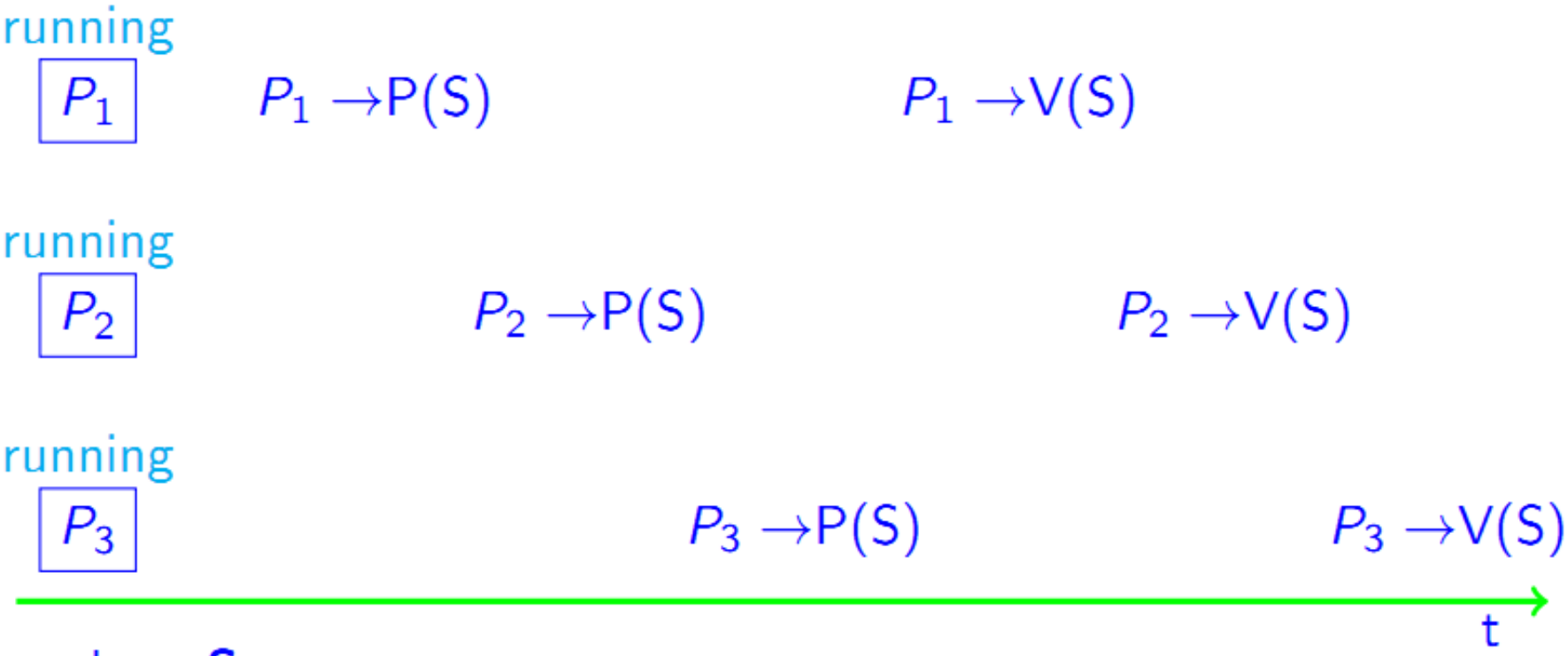


Semaphore **S**

S.value = 0

S.Ptr ⟶ NULL

4.1. Kỹ thuật đèn báo (Semaphore)



Semaphore **S**

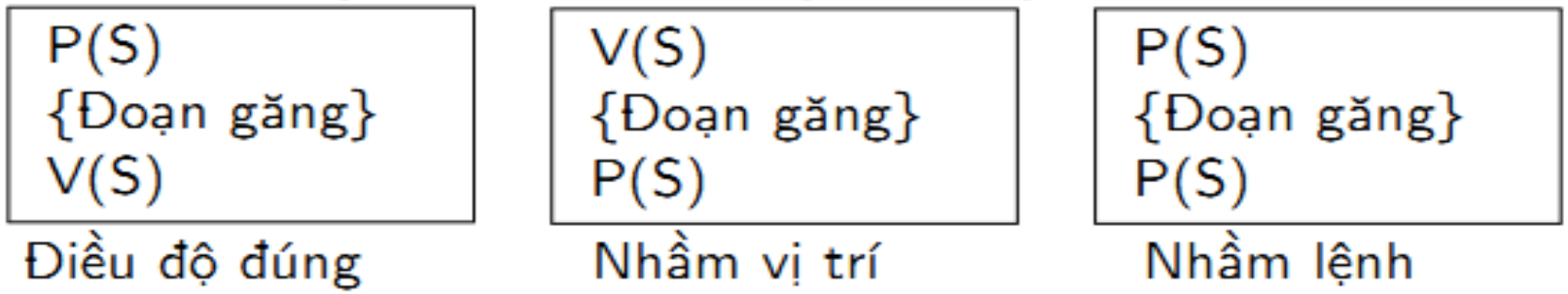
S.value = 1

S.Ptr ⟶ NULL

4.1. Kỹ thuật đèn báo (Semaphore)

Nhận xét

- Dễ dàng áp dụng cho các hệ thống phức tạp
- Không tồn tại hiện tượng chờ đợi tích cực
- Hiệu quả sử dụng phụ thuộc vào người dùng



- Các phép xử lý P(S) và V(S) không phân chia được

⇒bản thân P(S) và V(S) cũng là tài nguyên găng

⇒Cần phải điều độ

- Hệ thống 1 VXL: Cấm ngắt khi thực hiện wait(), signal()
- Hệ thống nhiều vi xử lý:
  - Không thể cấm ngắt trên VXL khác
  - Có thể dùng phương pháp khóa trong

4.1. Kỹ thuật đèn báo (Semaphore)
Đối tượng Semaphore trọng win32

- **Create Semaphore(…)**: Tạo một Semaphore
  - **LPSECURITY_ATTRIBUTES** IpSemaphoreAttributes

=> Trỏ tới cấu trúc an ninh, thẻ trả về được kế thừa

  - **LONG** InitialCount

=> Giá trị khởi tạo cho đối tượng Semaphore

  - **LONG** MaximumCount

=> Giá trị lớn nhất của đối tượng Semaphore

  - **LPCTSTR** IpName

⇒ Tên của đối tượng Semaphore

Ví dụ CreateSemaphore(NULL, 0,1,NULL)

Trả về thẻ (HANDLE) của đối tượng Semaphore hoặc NULL

  - **WaitForSingleObject** (HANDLE h, DWORD time)
  - **ReleaseSemaphore**(…)
    - **HANDLE** hSemaphore: Thẻ của đối thượng Semaphore
    - **LONG** IReleaseCount: Giá trị được tăng lên
    - **LPLONG** IpPreviousCount: Giá trị trước đó

Ví dụ: ReleaseSemaphore(S,1,NULL)

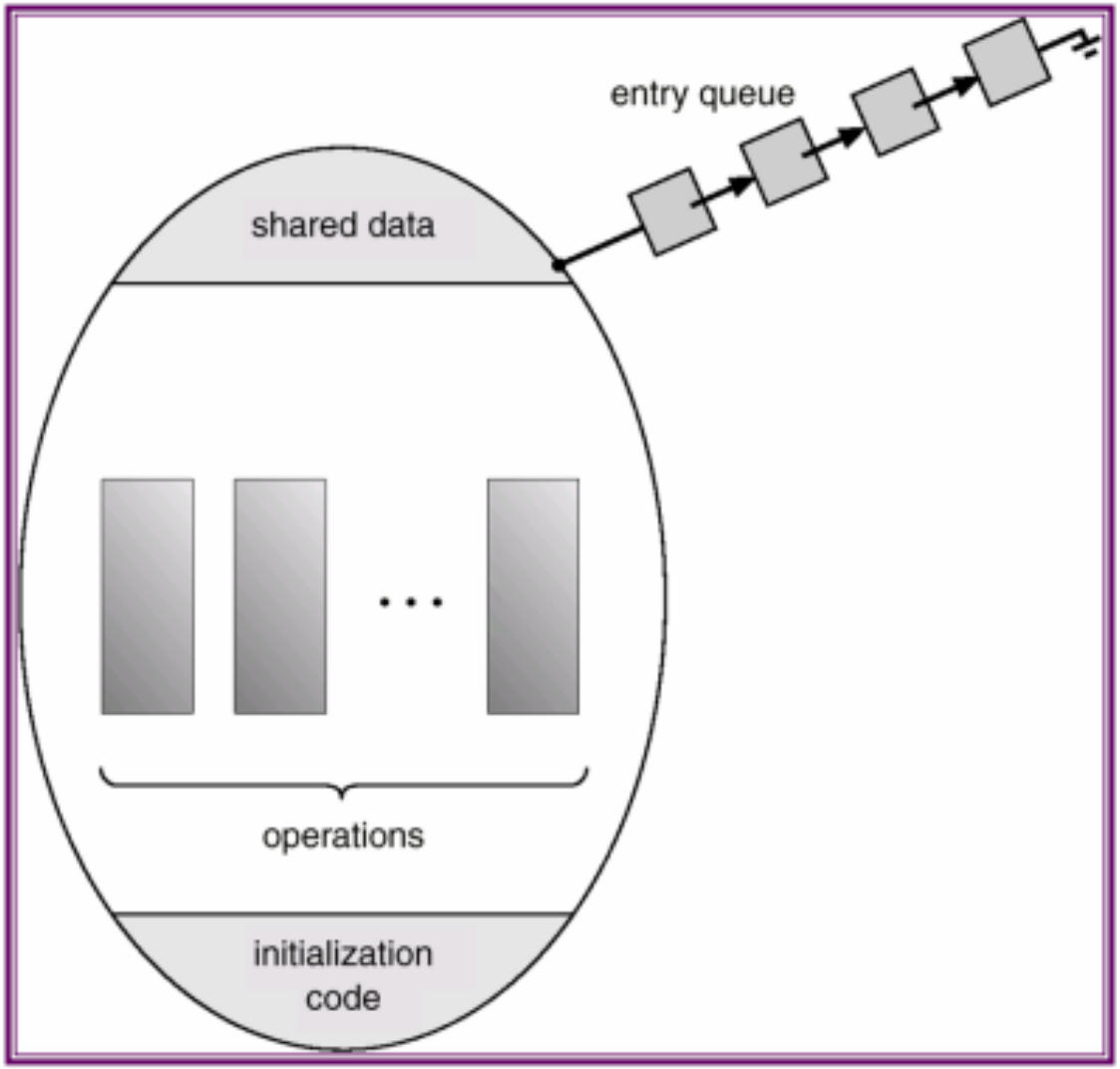
4.2. Monitor

- Là một kiểu dữ liệu đặc biệt, được đề nghị bởi HOARE 1974
- Bao gồm các thủ tục, dữ liệu cục bộ, đoạn mã khởi tạo
- Các tiến trình chỉ có thể truy nhập tới các biến bởi gọi tới các thủ tục trong Monitor
- Tại một thời điểm chỉ có một tiến trình được quyền sử dụng Monitor
  - Tiến trình khác muốn sử dụng, phải chờ
- Cho phép các tiến trình đợi trong Monitor
  - Sử dụng các biến điều kiện

```
Monitor monitorName{
//Khai báo các biến chung
Procedure P1(…){
   ….
}
….
Procedure Pn(…){
   ….
}
{
        Mã khởi tạo
}
};
```

Cú pháp của Monitor

4.2. Monitor
Mô hình

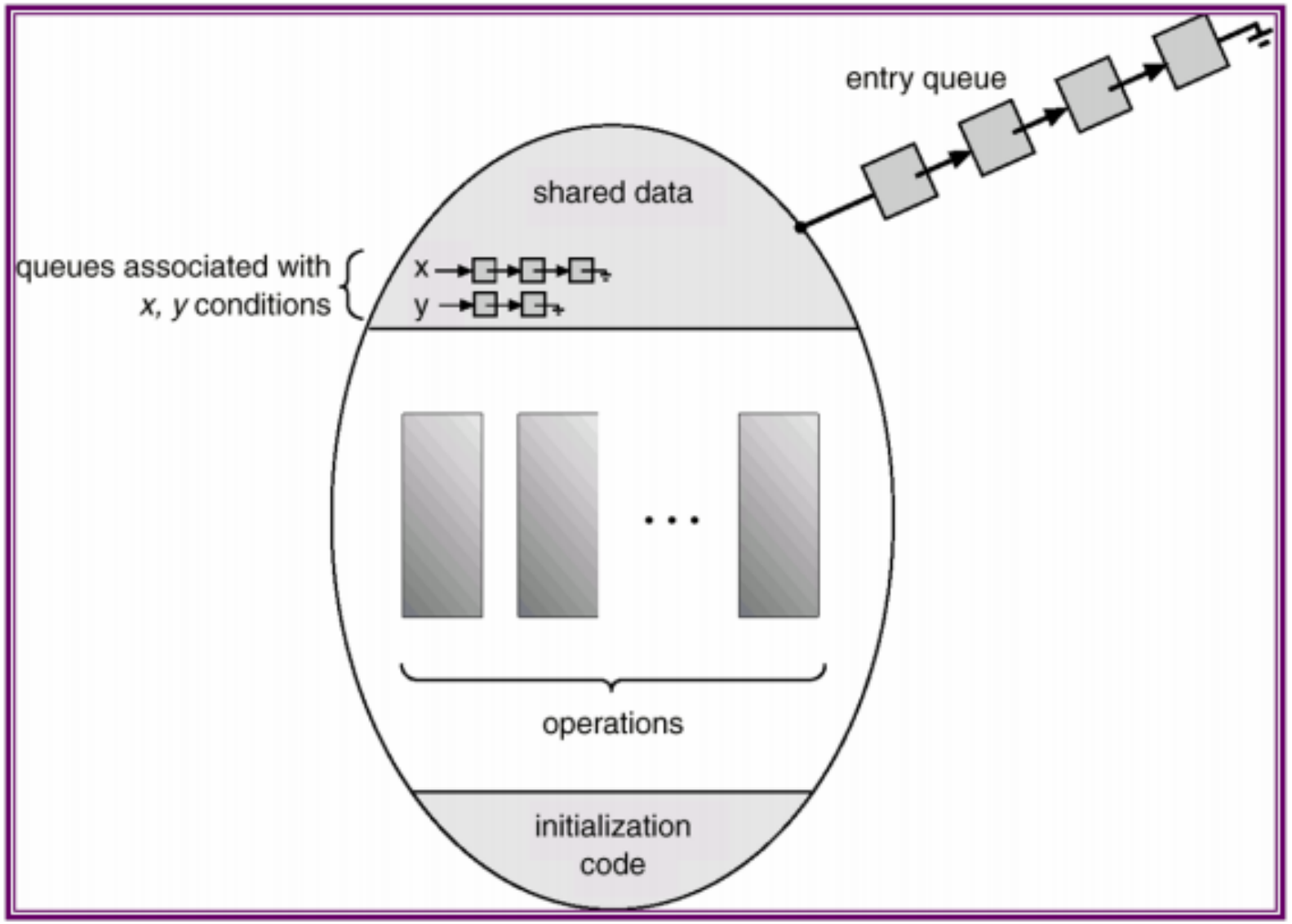


4.2. Monitor
Biến điều kiện

- Thực chất là tên của một hàng đợi
- Khai báo: condition x,y;
- Các biến điều khiển chỉ có thể được sử dụng với 2 thao tác

Wait()	Được gọi bởi các thủ tục của Monitor (cú pháp: x.wait() hoặc wait(x)) cho phép tiến trình đưa ra lời gọi bị tạm dừng (block) cho tới khi được một tiến trình khác kích hoạt bởi gọi tới signal()
Signal()	Được gọi bởi các thủ tục của Monitor (Cú pháp: x.signal() hoặc signal(x)) kích hoạt chính xác một tiến trình đang đợi tại biến điều kiện x (nằm trong hàng đợi x) ra tiếp tục hoạt động. Nếu không có tiến trình nào đang đợi, thao tác không có hiệu lực.

4.2. Monitor
Mô hình



4.2. Monitor
Sử dụng – 1 tài nguyên chung

```
Monitor Resource{
   Condition Nonbusy;
   Boolean Busy
//-- Phần dành người dùng --
   void Acquire(){
      if(busy) Nonbusy.wait();
         busy=true;
   }
   void Release(){
      busy=false
      signal(Nonbusy)
   }
//---- Phần khởi tạo ----
   busy= false;
   Nonbusy = Empty;
}
```

Cấu trúc tiến trình

while(1){

...

Resource.Acquire()

{Sử dụng tài nguyên}

Resource.Release()

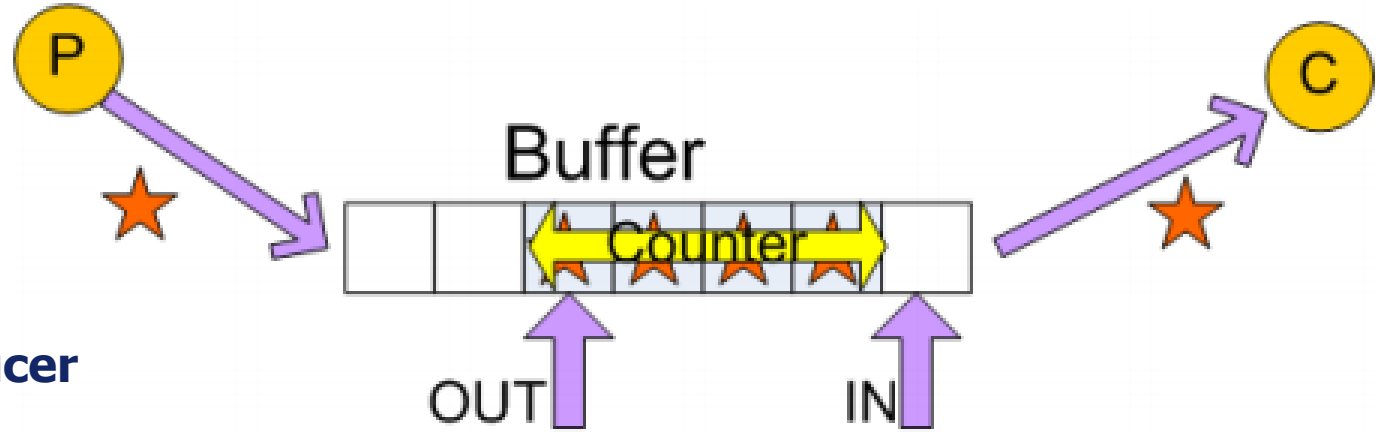
...

}

Một số bài toán

- Người sản xuất – Người tiêu thụ (Producer-Consumer)
- Triết gia ăn tối (Dining Philosophers)
- Người đọc và biên tập viên (Readers-Writers)
- Người thợ cắt tóc ngủ gật (Sleeping Barber)
- Bathroom Prolem
- Đồng bộ Barriers
- ….

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 1



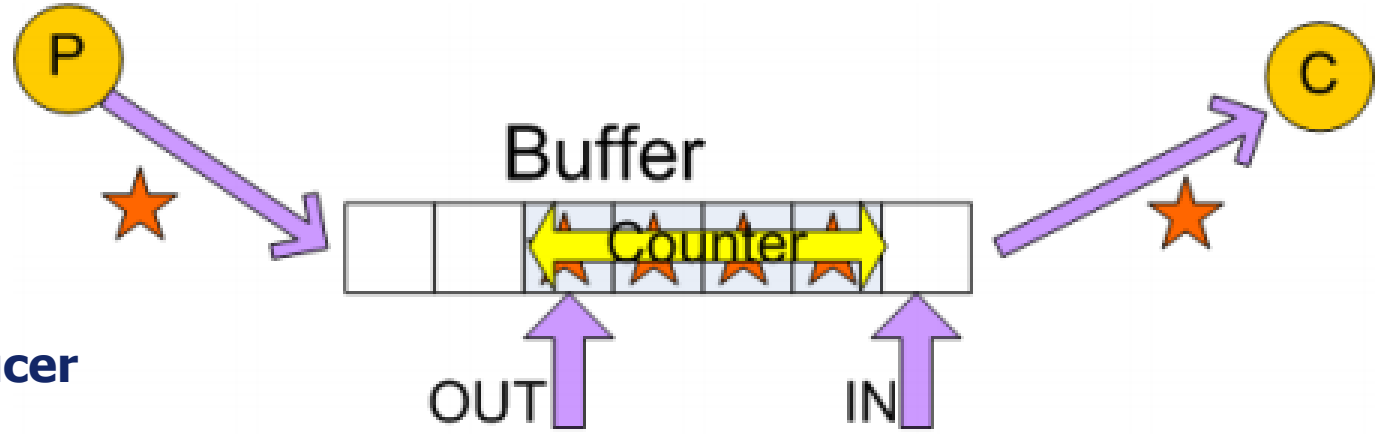
Producer

```
do{
    {tạo phần tử mới}
    if(Counter==SIZE);
    {Đặt phần tử mới vào Buffer
      IN=(IN+1)%SIZE;}
     Counter++;
}while(1);
```

Consumer

```
do{
    if(Counter==0);
    {Lấy 1 phần tử trong Buffer
       OUT = (OUT+1)%SIZE;}
     Counter--;
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 1



Producer

```
do{
    {tạo phần tử mới}
    if(Counter==SIZE) block();
    {Đặt phần tử mới vào Buffer
      IN=(IN+1)%SIZE;}
    Counter++;
    if(Counter==1)
       wakeup(Consumer);
}while(1);
```

Consumer

```
do{
    if(Counter==0); block();
    {Lấy 1 phần tử trong Buffer
      OUT = (OUT+1)%SIZE;}
    Counter--;
    if(Counter==SIZE-1)
       wakeup(Producer);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 2

- **Giải pháp**: Dùng một đèn báo Mutex để điều độ biến Counter
- **Khởi tạo**: Mutex = 1

Producer

```
do{
    {tạo phần tử mới}
    if(Counter==SIZE) block();
    {Đặt phần tử mới vào Buffer}
      wait(Mutex)
     Counter++;
      signal(Mutex);
     if(Counter==1)
        wakeup(Consumer);
}while(1);
```

Consumer

```
do{
    if(Counter==0); block();
    {Lấy 1 phần tử trong Buffer}
       wait(Mutex);
     Counter--;
      signal(Mutex);
    if(Counter==SIZE-1)
        wakeup(Producer);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 2

- **Giải pháp**: Dùng một đèn báo Mutex để điều độ biến Counter
- **Khởi tạo**: Mutex = 1

Producer

```
do{
    {tạo phần tử mới}
    if(Counter==SIZE) block();
    {Đặt phần tử mới vào Buffer}
      wait(Mutex)
     Counter++;
      signal(Mutex);
     if(Counter==1)
        wakeup(Consumer);
}while(1);
```

Consumer

```
do{
    if(Counter==0); block();
    {Lấy 1 phần tử trong Buffer}
       wait(Mutex);
     Counter--;
      signal(Mutex);
    if(Counter==SIZE-1)
        wakeup(Producer);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

- **Vấn đề**: Giả thiết Counter = 0
  - *Consumer* kiểm tra counter => gọi thực hiện lệnh block();
  - *Producer* Tăng counter lên 1 và gọi hàm wakeup(consumer)
  - *Consumer* chưa bị block => Câu lệnh wakeup bị bỏ qua

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
    signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

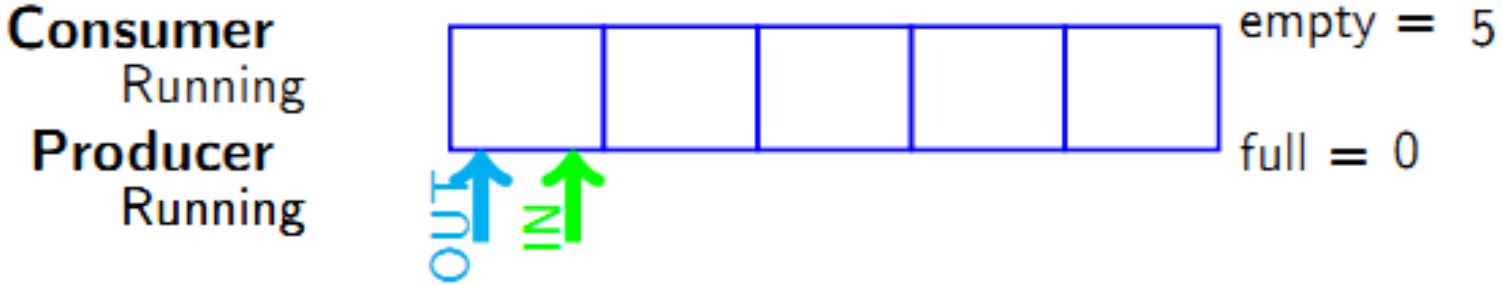
- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
   signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
     signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

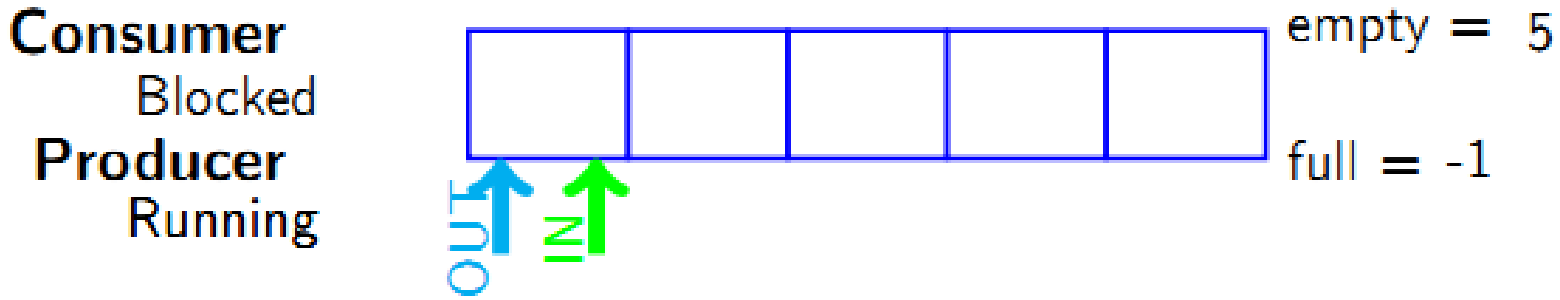
- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
   signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
     signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

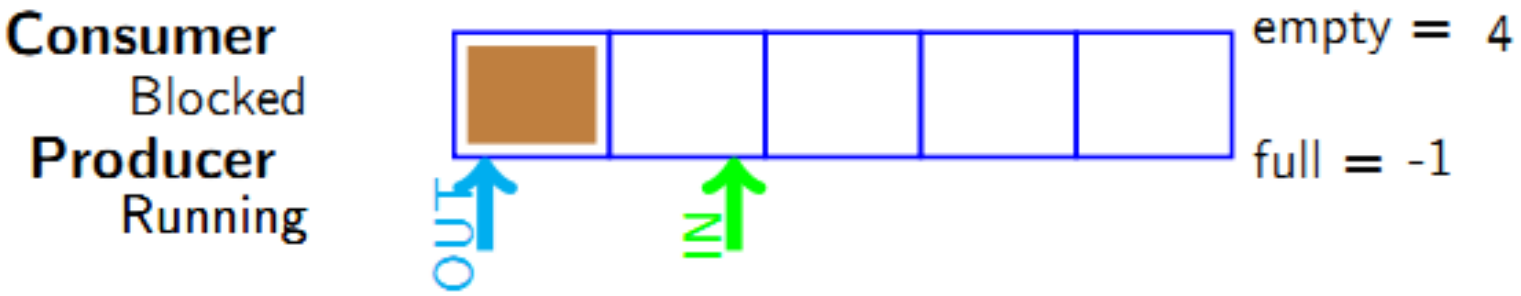
- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
    signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

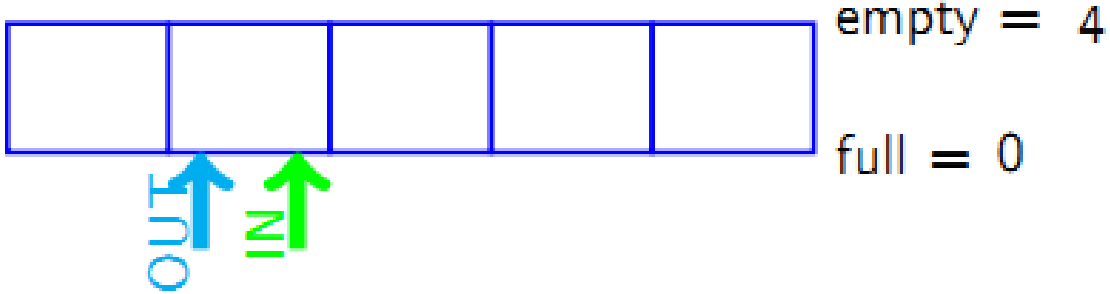
Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
   signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
     signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Consumer
Running
Producer
Running



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

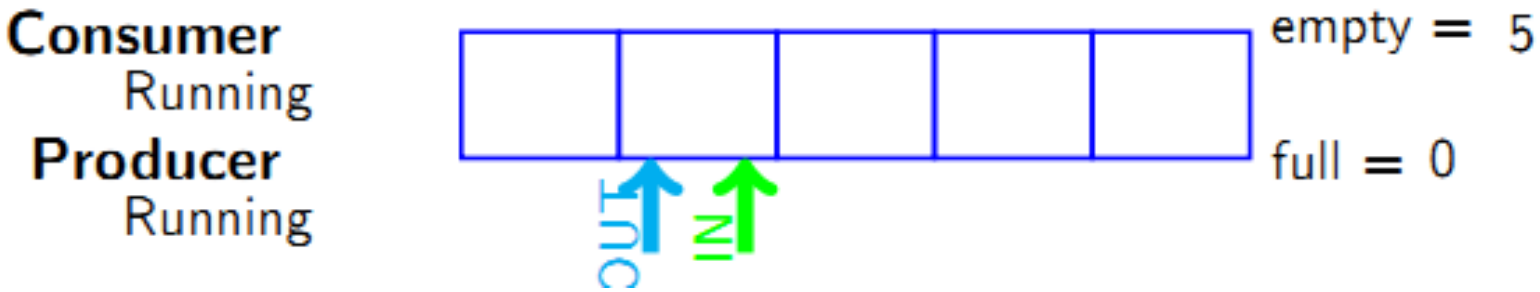
- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
   signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
     signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

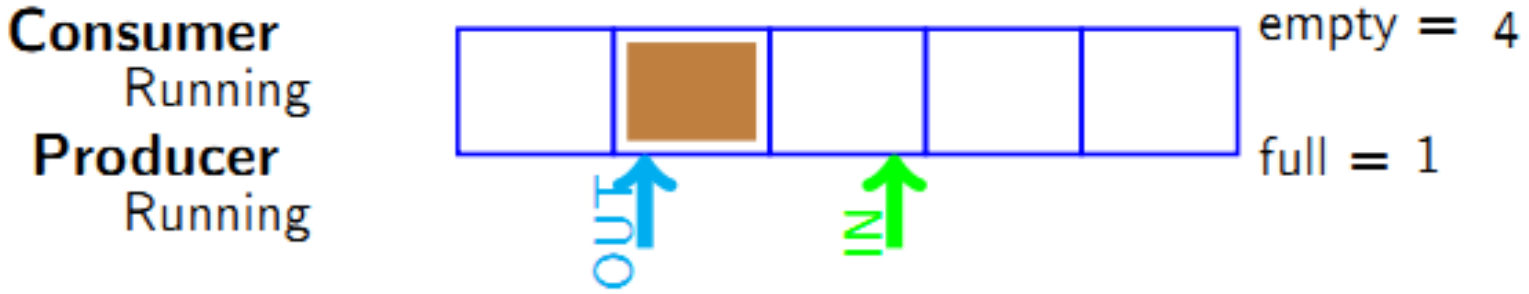
- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
   signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
     signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

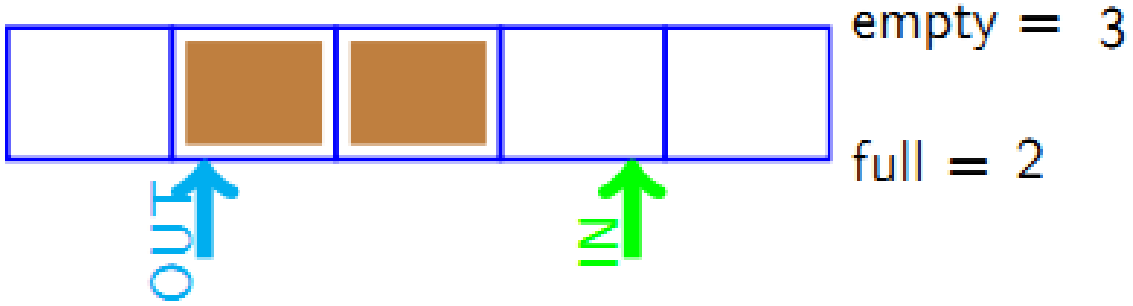
Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
   signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
    signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Consumer
Running
Producer
Running



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

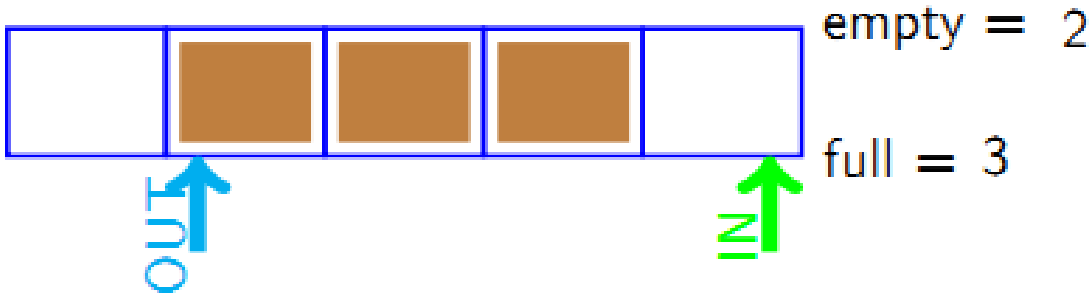
Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
   signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
     signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Consumer
Running
Producer
Running



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

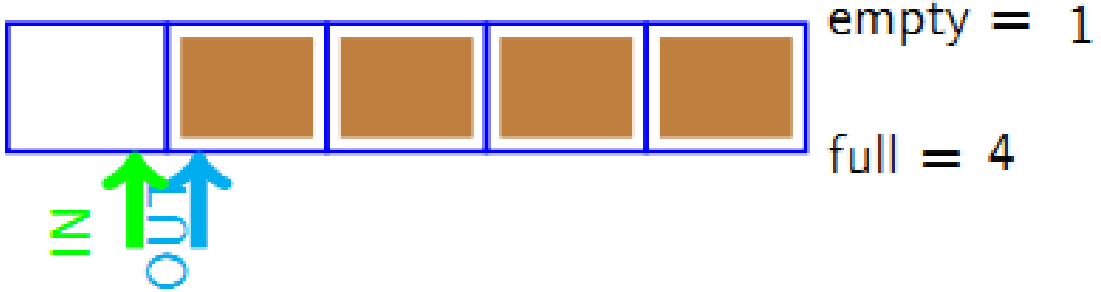
Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
   signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
     signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Consumer
Running
Producer
Running



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

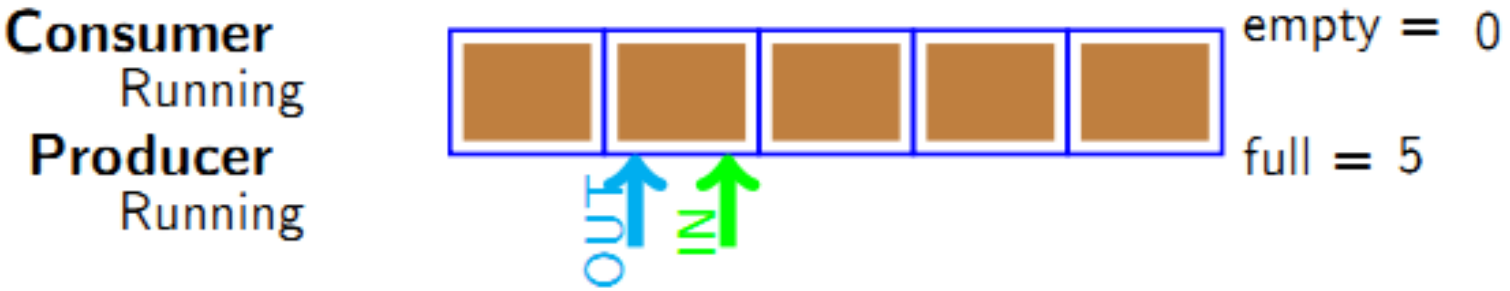
- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
    signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

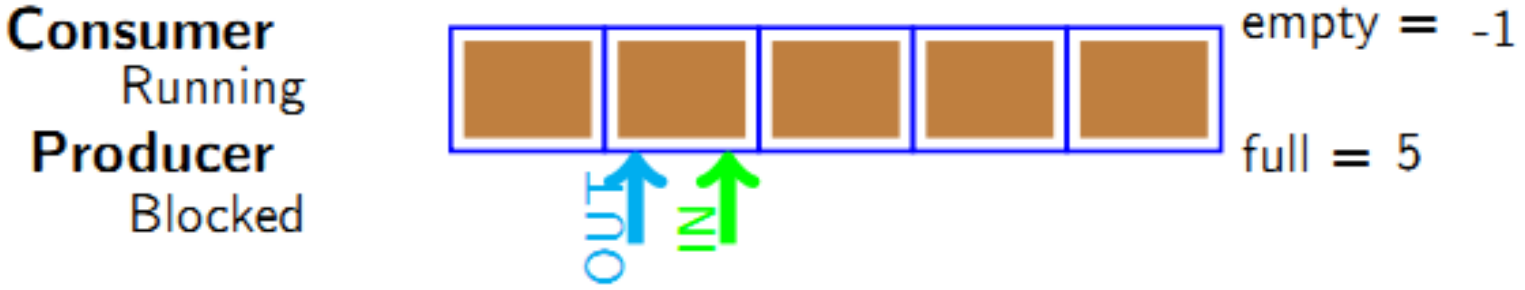
- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
    signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

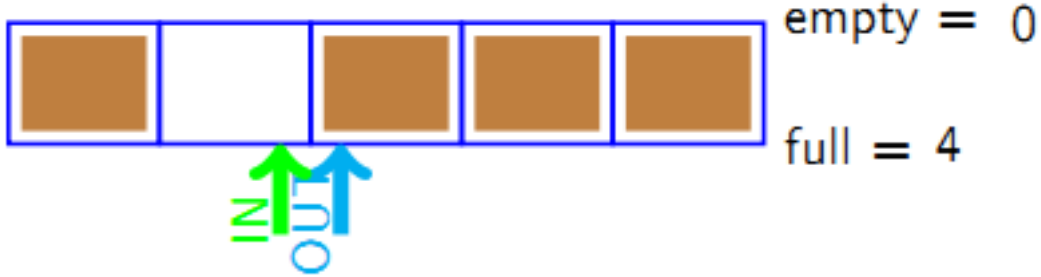
Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
    signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Consumer
Running
Producer
Running



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 3

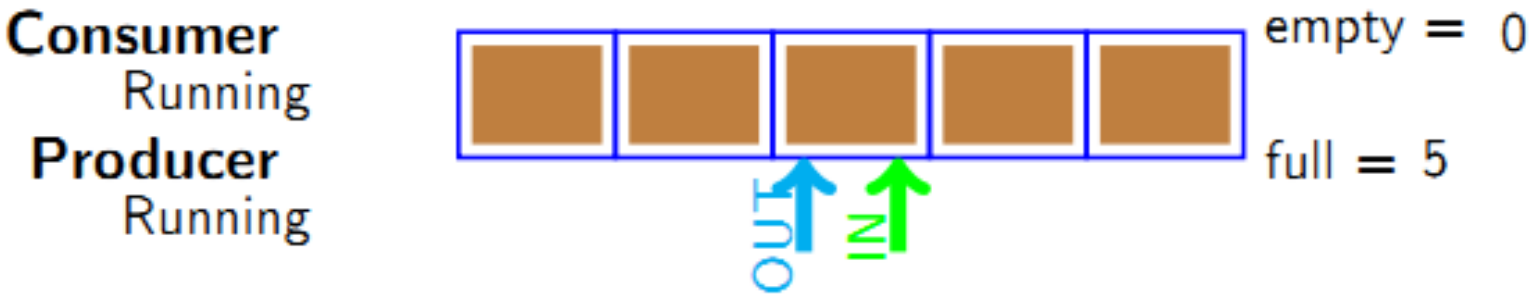
- **Giải pháp**: Sử dụng 2 đèn báo full, empty được khởi tạo
  - Full =0: Số phần tử trong hòm thư
  - Empty = Buffer_size: Số chỗ trống trong hòm thư

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    {Đặt phần tử mới vào Buffer}
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    {Lấy 1 phần tử trong Buffer}
    signal(empty);
    {Xử lý phần tử vừa lấy ra}
} while(1);
```



Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 4

- **Vấn đề:** Khi có nhiều Producers và Consumer, các biến IN và OUT trở thành tài nguyên găng giữa chúng
- Giải quyết: Dùng đèn báo thứ 3 (mutex<-1) để đồng bộ các tiến trình cùng loại.

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 4

- **Vấn đề:** Khi có nhiều Producers và Consumer, các biến IN và OUT trở thành tài nguyên găng giữa chúng
- Giải quyết: Dùng đèn báo thứ 3 (mutex<-1) để đồng bộ các tiến trình cùng loại.

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    wait(mutex)

        {Đặt phần tử mới vào Buffer}
    signal(mutex)
    signal(full)

}while(1);
```

Consumer

```
do{
    wait(full)
    wait(mutex)

    {Lấy 1 phần tử trong Buffer}
    signal(mutex)
    signal(empty)

    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Ví dụ về đồng bộ tiến trình
Vấn đề sản xuất – tiêu thụ 4

- **Vấn đề:** Khi có nhiều Producers và Consumer, các biến IN và OUT trở thành tài nguyên găng giữa chúng
- Giải quyết: Dùng đèn báo thứ 3 (mutex<-1) để đồng bộ các tiến trình cùng loại.

Producer

```
do{
    {tạo phần tử mới}
    wait(empty)
    wait(mutex)

        {Đặt phần tử mới vào Buffer}
    signal(mutex)
    signal(full)

}while(1);
```

Consumer

```
do{
    wait(mutex)
    wait(full)

    {Lấy 1 phần tử trong Buffer}
    signal(mutex)
    signal(empty)

    {Xử lý phần tử vừa lấy ra}
} while(1);
```

Ví dụ về đồng bộ tiến trình Người đọc và biên tập viên

- Nhiều tiến trình (Readers) cùng truy nhập một sơ sở dữ liệu (CSDL)
- Một số tiến tình (Writers) cập nhật CSDL
- Cho phép số lượng tùy ý các tiến trình Readers cùng truy nhập CSDL
  - Đang tồn tại một tiến tình Reader truy nhập CSDL, mọi tiến trình Readers khác mới xuất diện đều được truy nhập CSDL (Tiến trình Writers phải xếp hàng chờ đợi)
- Chỉ cho phép một tiến trình Wwiters cập nhật CSDL tại một tời điểm
- Vấn đề không trưng dụng. Các tiến trình ở trong đoạt găng mà không bị ngắt

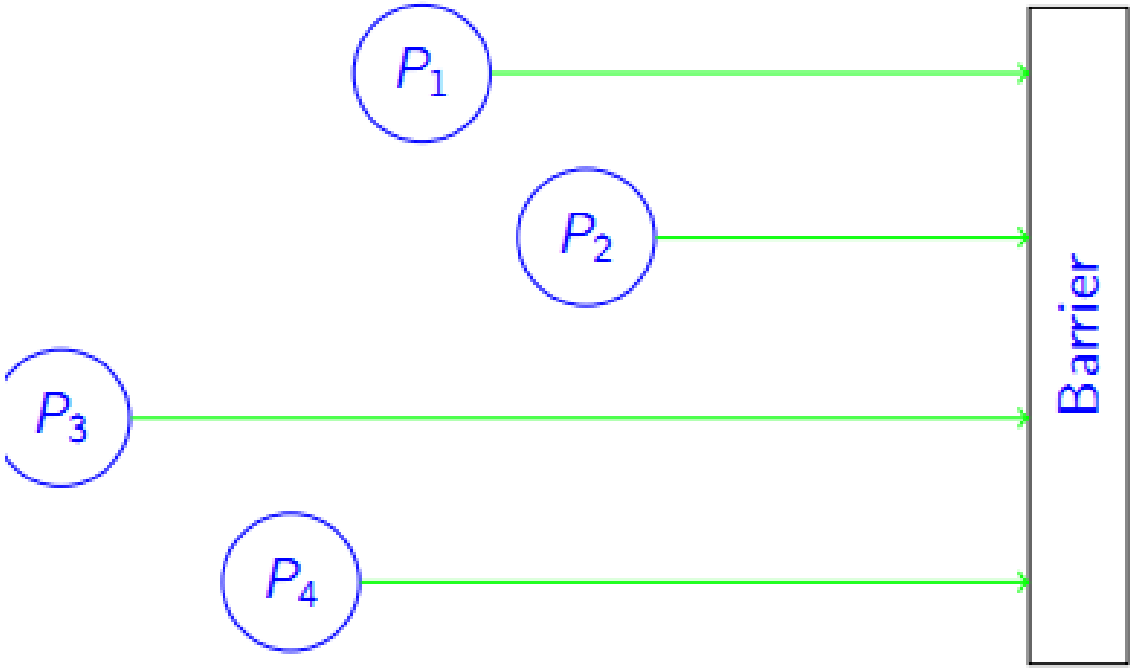
Ví dụ về đồng bộ tiến trình
Thợ cắt tóc ngủ gật

- N ghế đợi cho khách hàng
- Một người thợ chỉ có thể cắt tóc cho một khách hàng tại một thời điểm
  - Không có khách hàng đợi, thợ cắt tóc ngủ
- Khi một khách hàng tới
  - Nếu thợ cắt tóc đang ngủ => đánh thức anh ta dậy làm việc
    - Không còn ghế đợi trống => bỏ đi
    - Còn ghế đợi trống => ngồi đợi

Ví dụ về đồng bộ tiến trình
Vấn đề phòng tắm

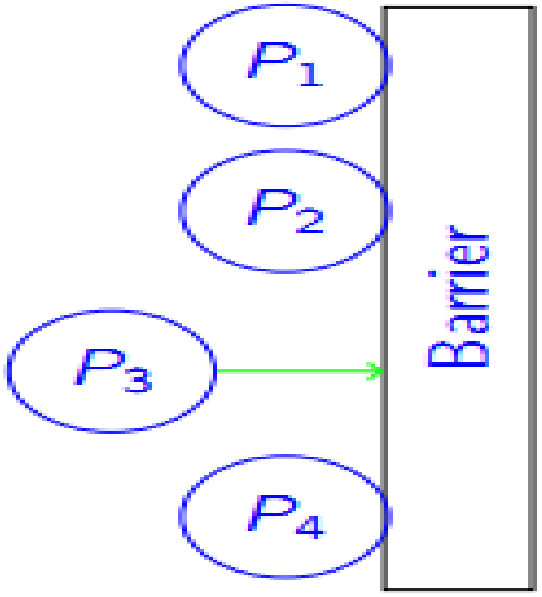
- Thường được dùng cho mục đích minh họa vấn đề phân phối tài nguyên trong nghiên cứu HĐH và tính toán song song
- Bài toán
  - Một phòng tắm được sử dùng cho cả nam và nữ nhưng không đồng thời
  - Nếu phòng tắm trống thì bất kỳ ai cũng có thể vào
  - Nếu phòng tắm đã có người thì chỉ duy nhất người cùng giới tính được vào
  - Số người trong phòng tắm trong cùng một thời điểm được giới hạn.
- Yêu cầu cài đặt bài toán thỏa mãn các ràng buộc
  - Có 2 kiểu tiến tình male() và female()
  - Mỗi t/trình ở trong phòng tắm một khoảng tgian ngẫu nhiên

Ví dụ về đồng bộ tiến trình
Đồng bộ barriers



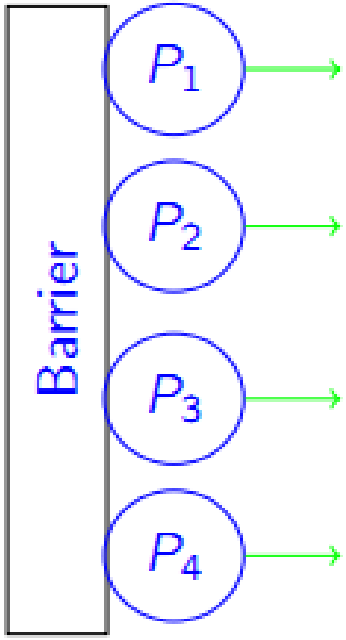
- Cá tiến trình hướng tới một ba-ri-e chung

Ví dụ về đồng bộ tiến trình
Đồng bộ barriers



- Cá tiến trình hướng tới một ba-ri-e chung
- Khi đạt tới Ba-ri-e, tất cả các tiến trình đều gọi block ngoại trừ tiến trình cuối cùng.

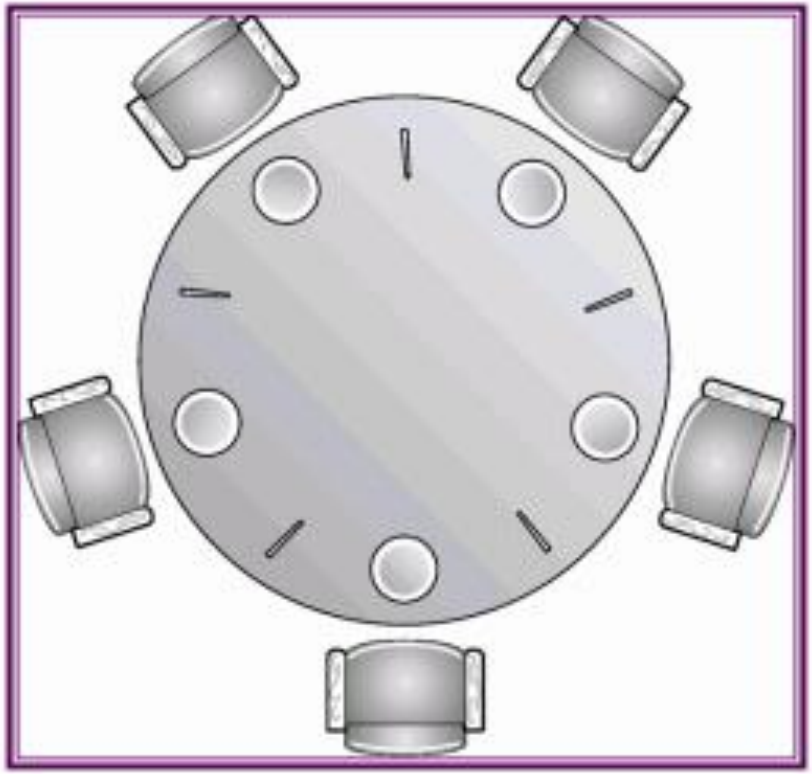
Ví dụ về đồng bộ tiến trình
Đồng bộ barriers



- Cá tiến trình hướng tới một ba-ri-e chung
- Khi đạt tới Ba-ri-e, tất cả các tiến trình đều gọi block ngoại trừ tiến trình cuối cùng.
- Khi tiến trình cuối cùng tới, đánh thức tất cả các tiến trình đang bị block và cùng vượt qua Ba-ri-e

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối

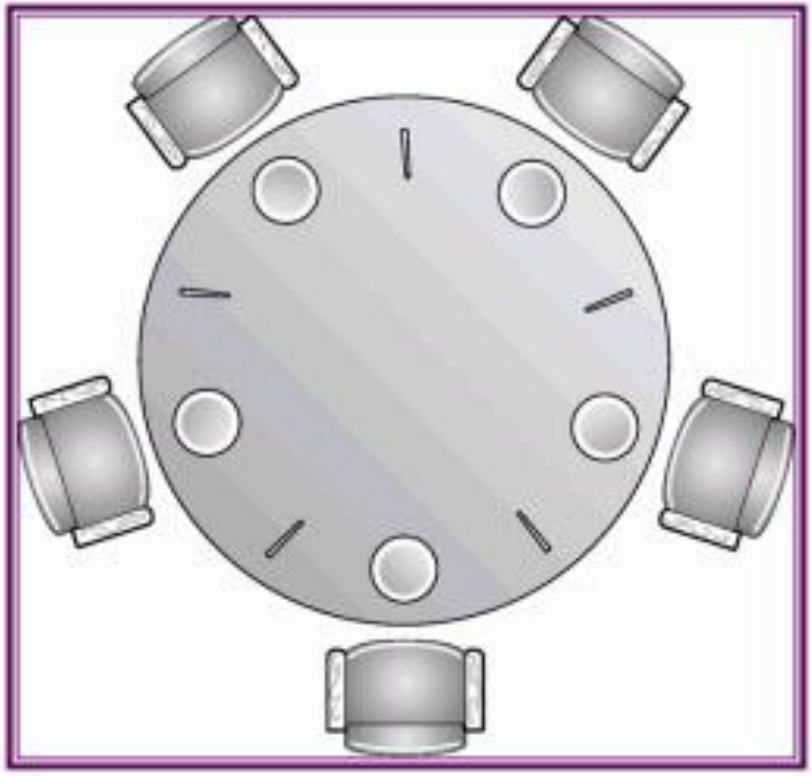
- Bài toán đồng bộ hóa tiến trình nổi tiếng, thể hiện tính trạng nhiều tiến trình phân chia nhiều tài nguyên



- 5 triết gia ăn tối quanh bàn tròn
  - Trước mỗi triết gia là một đia mì
  - Giữa 2 đĩa kề nhau là một cái dĩa (fork)
- Các triết gia thực hiện luân phiên, liên tục 2 việc: ăn, nghĩ
- Mỗi triết gia cần 2 cái dĩa để ăn
  - Chỉ lấy một dĩa tại một thời điểm
    - Lấy bên trái rồi lấy bên phải
- Ăn xong, triết gia để dĩa vào vị trí cũ

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối

- Bài toán đồng bộ hóa tiến trình nổi tiếng, thể hiện tính trạng nhiều tiến trình phân chia nhiều tài nguyên



- 5 triết gia ăn tối quanh bàn tròn
  - Trước mỗi triết gia là một đia mì
  - Giữa 2 đĩa kề nhau là một cái dĩa (fork)
- Các triết gia thực hiện luân phiên, liên tục 2 việc: ăn, nghĩ
- Mỗi triết gia cần 2 cái dĩa để ăn
  - Chỉ lấy một dĩa tại một thời điểm
    - Lấy bên trái rồi lấy bên phải
- Ăn xong, triết gia để dĩa vào vị trí cũ

Yêu cầu: Đồng bộ bữa tối của 5 triết gia

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – PP đơn giản

- Mỗi chiếc dĩa là một tài nguyên găng, được điều độ bởi một đèn báo fork[i]
- Semaphore fork[5] = {1,1,1,1,1};

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – PP đơn giản

- Mỗi chiếc dĩa là một tài nguyên găng, được điều độ bởi một đèn báo fork[i]
- Semaphore fork[5] = {1,1,1,1,1};
- Thuật toán cho triết gia Pi

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

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – PP đơn giản

- Mỗi chiếc dĩa là một tài nguyên găng, được điều độ bởi một đèn báo fork[i]
- Semaphore fork[5] = {1,1,1,1,1};
- Thuật toán cho triết gia Pi

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

- Nếu tất cả các triết gia cùng muốn ăn
  - Cùng lấy tất cả dĩa bên trái (gọi tới: wait(fork[i])
  - Cùng đợi chiếc dĩa bên phải (gọi tới: wait(fork[(i+1)%5]))

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – PP đơn giản

- Mỗi chiếc dĩa là một tài nguyên găng, được điều độ bởi một đèn báo fork[i]
- Semaphore fork[5] = {1,1,1,1,1};
- Thuật toán cho triết gia Pi

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

- Nếu tất cả các triết gia cùng muốn ăn **Bế tắc**
  - Cùng lấy tất cả dĩa bên trái (gọi tới: wait(fork[i])
  - Cùng đợi chiếc dĩa bên phải (gọi tới: wait(fork[(i+1)%5]))

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – Giải pháp 1

- Chỉ cho phép một triết gia lấy dĩa tại một thời điểm
- Semaphore: mutex <- 1

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – Giải pháp 1

- Chỉ cho phép một triết gia lấy dĩa tại một thời điểm
- Semaphore: mutex <- 1
- Thuật toán cho Triết gia Pi

```
do{
  wait(mutex)
    wait(fork[i])
      wait(fork[(i+1)%5]);
  Signal(mutex);
        {ăn}
      signal(fork[(i+1)%5]);
   signal(i);
        {nghĩ};
}while(1);
```

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – Giải pháp 1

- Chỉ cho phép một triết gia lấy dĩa tại một thời điểm
- Semaphore: mutex <- 1
- Thuật toán cho Triết gia Pi

```
do{
  wait(mutex)
    wait(fork[i])
      wait(fork[(i+1)%5]);
  Signal(mutex);
        {ăn}
      signal(fork[(i+1)%5]);
   signal(i);
        {nghĩ};
}while(1);
```

- **Có thể làm cho 2 triết gia không kề nhau cùng được ăn tại một thời điểm (P1 ăn, p2 chiếm mutex => p3 đợi)**

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – Giải pháp 2

- Thứ tự lấy dĩa của các triết gia khác nhau
  - Triết gia số hiệu chẵn lấy dĩa trái trước
  - Triết gia số hiệu lẻ lấy dĩa phải trước

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – Giải pháp 2

- Thứ tự lấy dĩa của các triết gia khác nhau
  - Triết gia số hiệu chẵn lấy dĩa trái trước
  - Triết gia số hiệu lẻ lấy dĩa phải trước
- Thuật toán cho triết gia Pi

```
do{
  j=i%2
    wait(fork[(i+1)%5])
        wait(fork[(i+1-j)%5]);
          {ăn}
        signal(fork[(i+1-j)%5]);
    signal((i+j)%5);
          {nghĩ};
}while(1);
```

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – Giải pháp 2

- Thứ tự lấy dĩa của các triết gia khác nhau
  - Triết gia số hiệu chẵn lấy dĩa trái trước
  - Triết gia số hiệu lẻ lấy dĩa phải trước
- Thuật toán cho triết gia Pi

```
do{
  j=i%2
    wait(fork[(i+1)%5])
        wait(fork[(i+1-j)%5]);
          {ăn}
        signal(fork[(i+1-j)%5]);
    signal((i+j)%5);
          {nghĩ};
}while(1);
```

- Giải quyết được bế tắc

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – Giải pháp khác

- Trả lại dĩa bên trái nếu không lấy được dĩa bên phải
  - Kiểm tra dĩa phải sẵn sàng trước khi gọi wait(fork[(i+1)%5])
  - Nếu không sẵn có: trả lại dĩa trái, đợi một thời gian rồi thử lại
  - Không bế tắc, nhưng không tiến triển: nạn đói (starvation)
  - Thực hiện trong thực tế, nhưng không đảm bảo về lý thuyết

Ví dụ về đồng bộ tiến trình
Bài toán triết gia ăn tối – Giải pháp khác

- Trả lại dĩa bên trái nếu không lấy được dĩa bên phải
  - Kiểm tra dĩa phải sẵn sàng trước khi gọi wait(fork[(i+1)%5])
  - Nếu không sẵn có: trả lại dĩa trái, đợi một thời gian rồi thử lại
  - Không bế tắc, nhưng không tiến triển: nạn đói (starvation)
  - Thực hiện trong thực tế, nhưng không đảm bảo về lý thuyết
- Công cụ điều độ monitor (cấp cao)

Q & A

- List Câu hỏi