



# Đồng bộ hóa

**Môn học: Hệ điều hành**

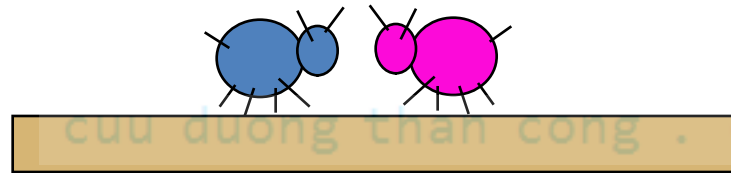
# Nội dung

- Xử lý đồng hành và các vấn đề:
  - Vấn đề tranh đoạt điều khiển (Race Condition)
  - Vấn đề phối hợp xử lý
- Bài toán đồng bộ hóa
  - Yêu cầu độc quyền truy xuất (Mutual Exclusion)
  - Yêu cầu phối hợp xử lý (Synchronization)
- Các giải pháp đồng bộ hoá
  - Busy waiting
  - Sleep & Wakeup
- Các bài toán đồng bộ hoá kinh điển
  - Producer – Consumer
  - Readers – Writers
  - Dining Philosophers



# Nhiều tiến trình “chung sống hoà bình” trong hệ thống ?

- ĐỪNG HY VỌNG
- An toàn khi các tiến trình hoàn toàn độc lập
  - Làm sao có được ??
- Thực tế
  - Các tiến trình chia sẻ tài nguyên chung ( file system, CPU...)
  - Concurrent access => bugs.
    - Ví dụ : Dê con qua cầu



■ Xử lý đồng hành = ...nhức đầu 🤯



# Các vấn đề

- Tranh chấp

- Nhiều tiến trình truy xuất đồng thời một tài nguyên mang bản chất không chia sẻ được
  - Xảy ra vấn đề tranh đoạt điều khiển (Race Condition)
- Kết quả ?
  - Khó biết , thường là ...sai 
- Luôn luôn nguy hiểm ?
  - ...Không, nhưng đủ để cân nhắc kỹ càng 

- Phối hợp

- Các tiến trình không biết tương quan xử lý của nhau để điều chỉnh hoạt động nhịp nhàng
  - Cần phối hợp xử lý (Rendez-vous)
- Kết quả : khó biết, không bảo đảm ăn khớp



# Nội dung bài giảng

- Xử lý đồng hành và các vấn đề:
  - Vấn đề tranh đoạt điều khiển (Race Condition)
  - Vấn đề phối hợp xử lý
- Bài toán đồng bộ hóa
  - Yêu cầu độc quyền truy xuất (Mutual Exclusion)
  - Yêu cầu phối hợp xử lý (Synchronization)
- Các giải pháp đồng bộ hoá
  - Busy waiting
  - Sleep & Wakeup
- Các bài toán đồng bộ hoá kinh điển
  - Producer – Consumer
  - Readers – Writers
  - Dining Philosophers



# Tranh đoạt điều khiển (Race condition) - Ví dụ

- Đếm số người vào Altavista : dùng 2 threads cập nhật biến đếm **hits** => P1 và P2 chia sẻ biến **hits**  
**hits = 0**

P1

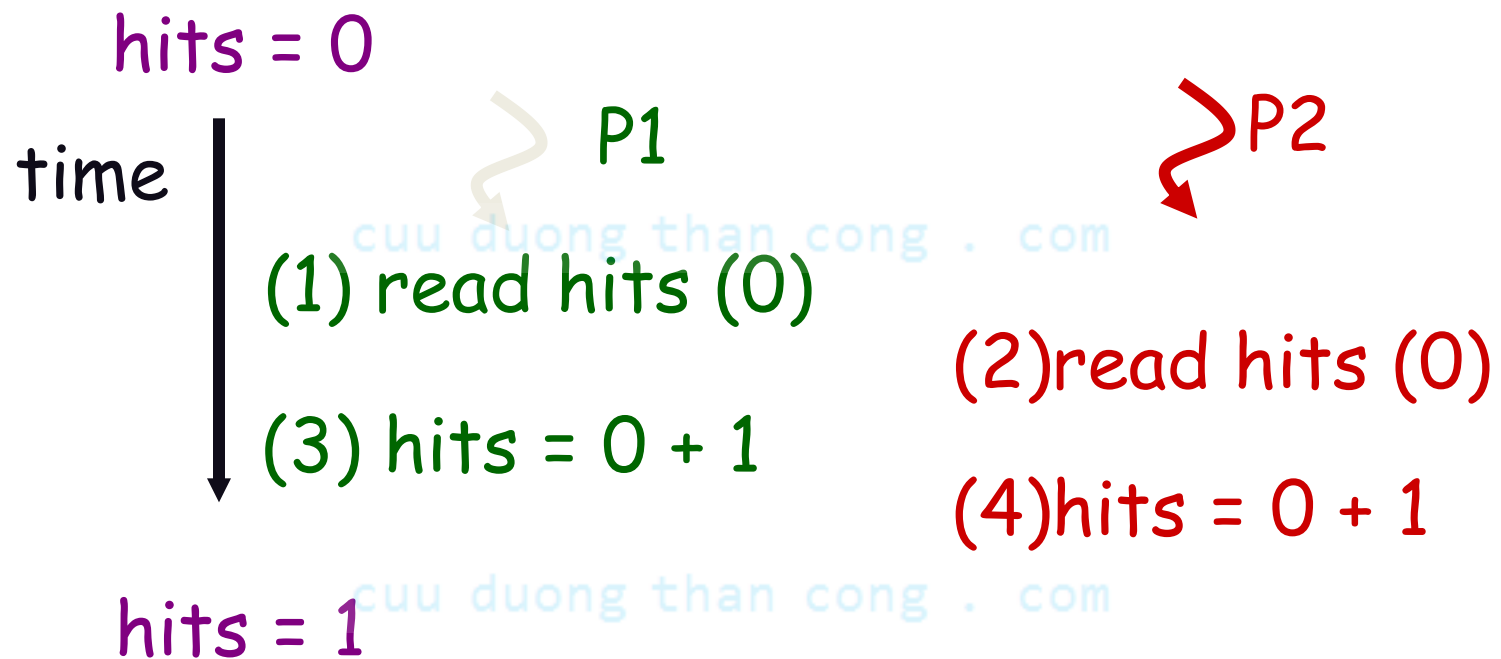
```
read hits;  
hits = hits + 1;
```

```
read hits;  
hits = hits + 1;
```

Kết quả cuối cùng là bao nhiêu ?

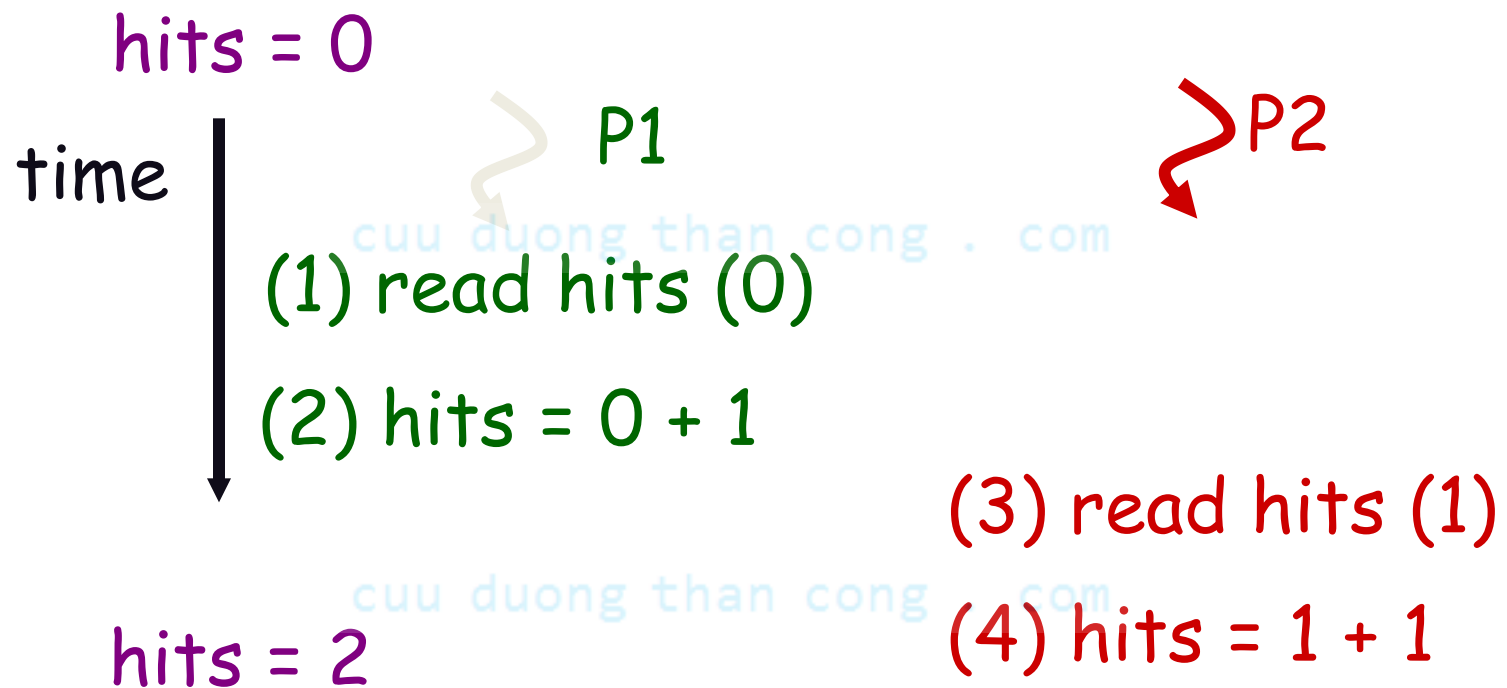


# Tranh đoạt điều khiển (Race condition) - Ví dụ





# Tranh đoạt điều khiển (Race condition) - Ví dụ





# Tranh đoạt điều khiển (Race condition) - Ví dụ (tt)

`i=0;`

Thread a:

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

Thread b:

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

- Ai thắng?
- Có bảo đảm rằng sẽ có người thắng?
- Nếu mỗi tiến trình xử lý trên 1 CPU thì sao?

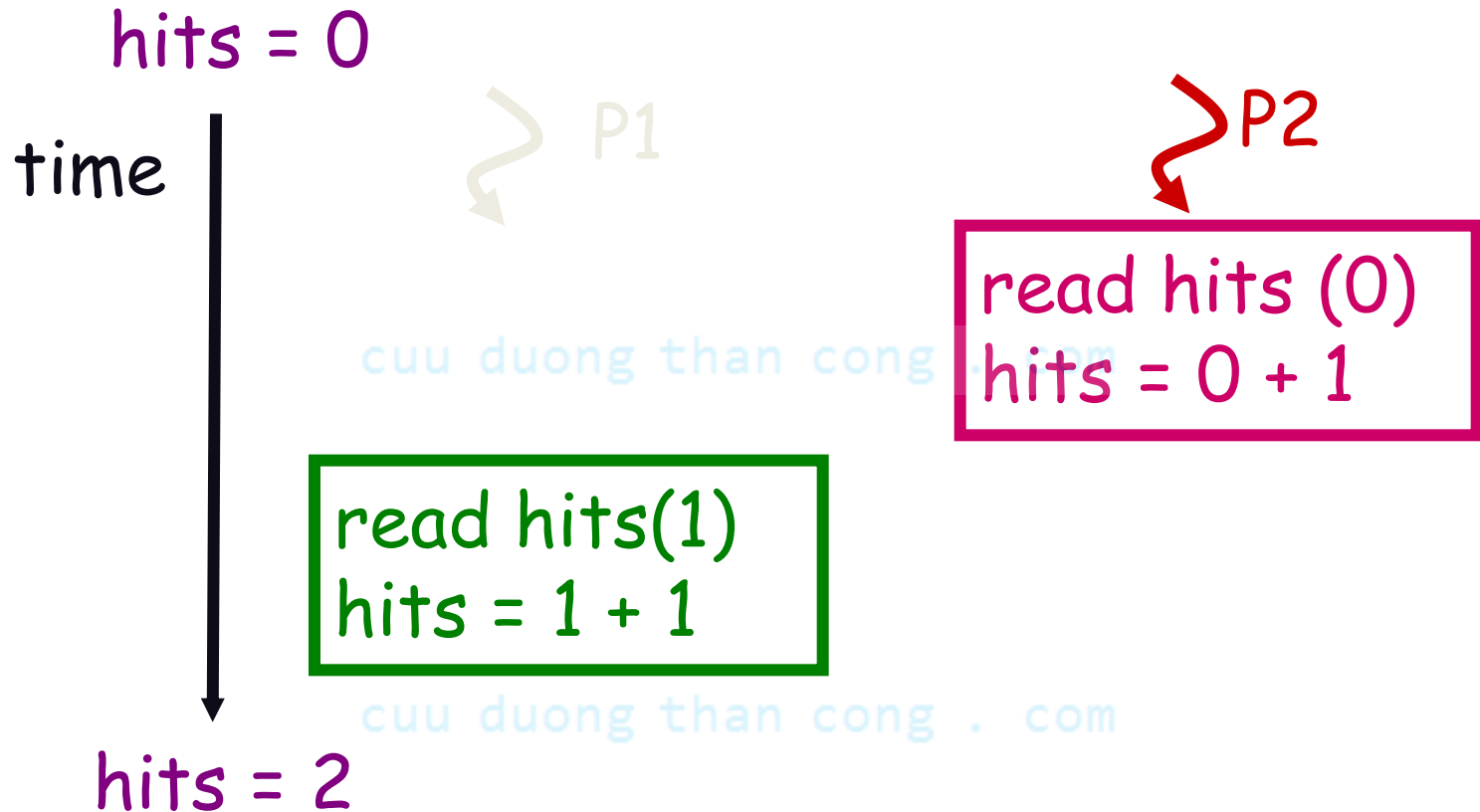


# Tranh đoạt điều khiển (Race condition)- Nhận xét

- Kết quả thực hiện tiến trình phụ thuộc vào kết quả điều phối
  - Cùng input, không chắc cùng output
  - Khó debug lỗi sai trong xử lý đồng hành
- Xử lý
  - Làm lơ
    - Dễ , nhưng có phải là giải pháp
  - Không chia sẻ tài nguyên chung : dùng 2 biến hits1,hits2; xây cầu 2 lane...
    - Nên dùng khi có thể, nhưng không bao giờ có thể đảm bảo đủ tài nguyên, và cũng không là giải pháp đúng cho mọi trường hợp
  - Giải pháp tổng quát : có hay không ?
    - Lý do xảy ra Race condition ? → **Bad interleavings** : một tiến trình “xen vào” quá trình truy xuất tài nguyên của một tiến trình khác
    - Giải pháp : bảo đảm tính **atomicity** cho phép tiến trình hoàn tất trọn vẹn quá trình truy xuất tài nguyên chung trước khi có tiến trình khác can thiệp

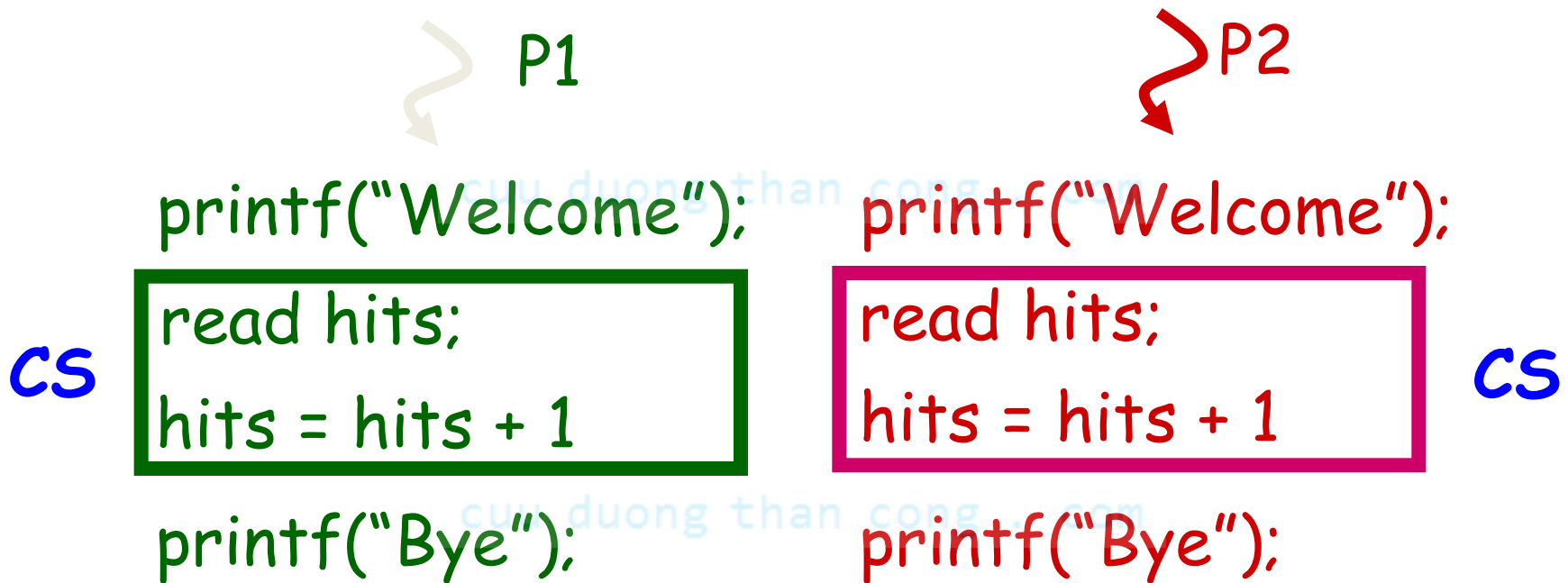


# Atomicity : loại bỏ Race Condition



# Miền găng (Critical Section) & Khả năng độc quyền (Mutual Exclusion)

- Miền găng (CS) là đoạn chương trình có khả năng gây ra hiện tượng race condition



- ☛ Hỗ trợ Atomicity : Cần bảo đảm tính “độc quyền truy xuất” (**Mutual Exclusion**) cho miền găng (CS)

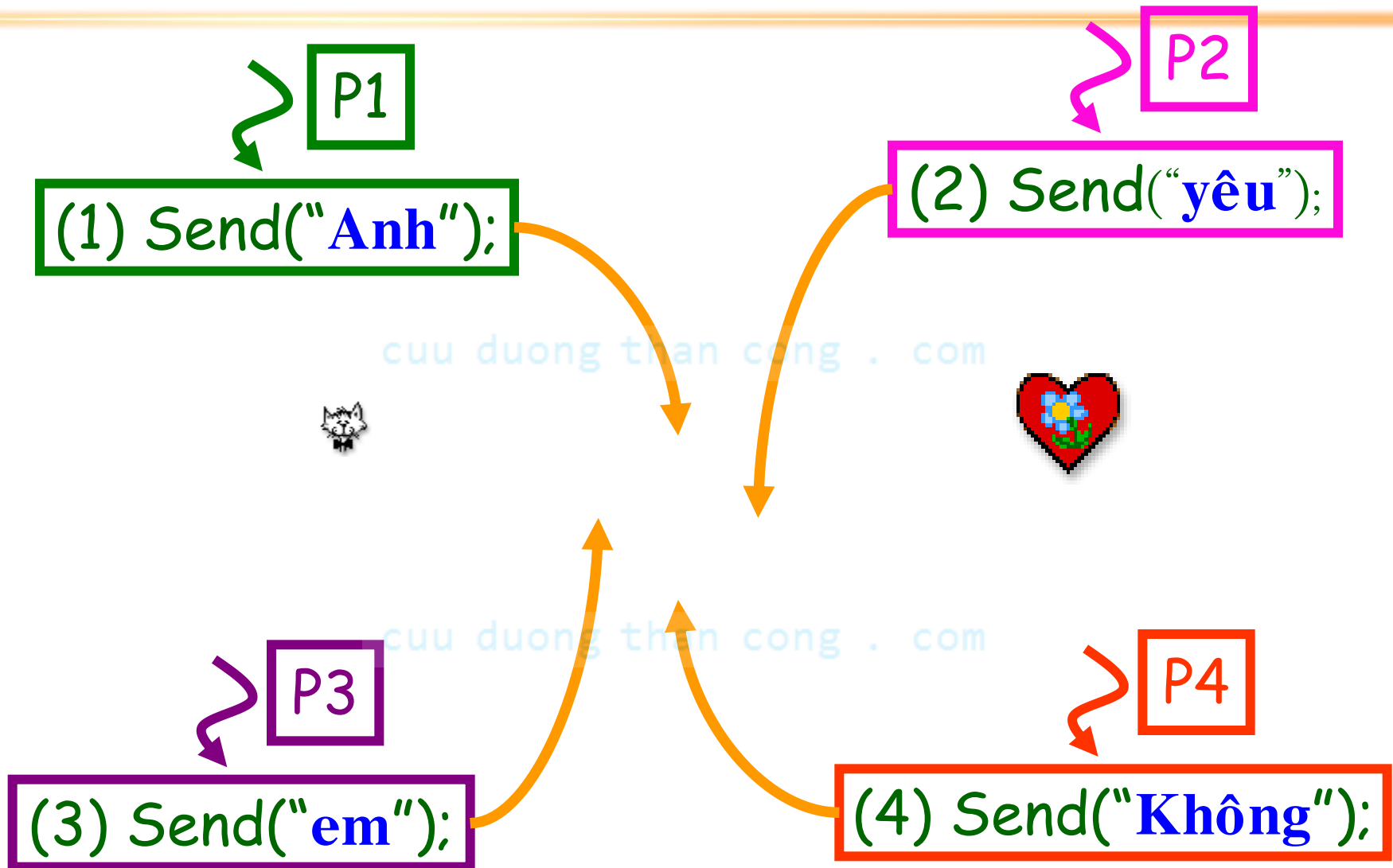


# Nội dung bài giảng

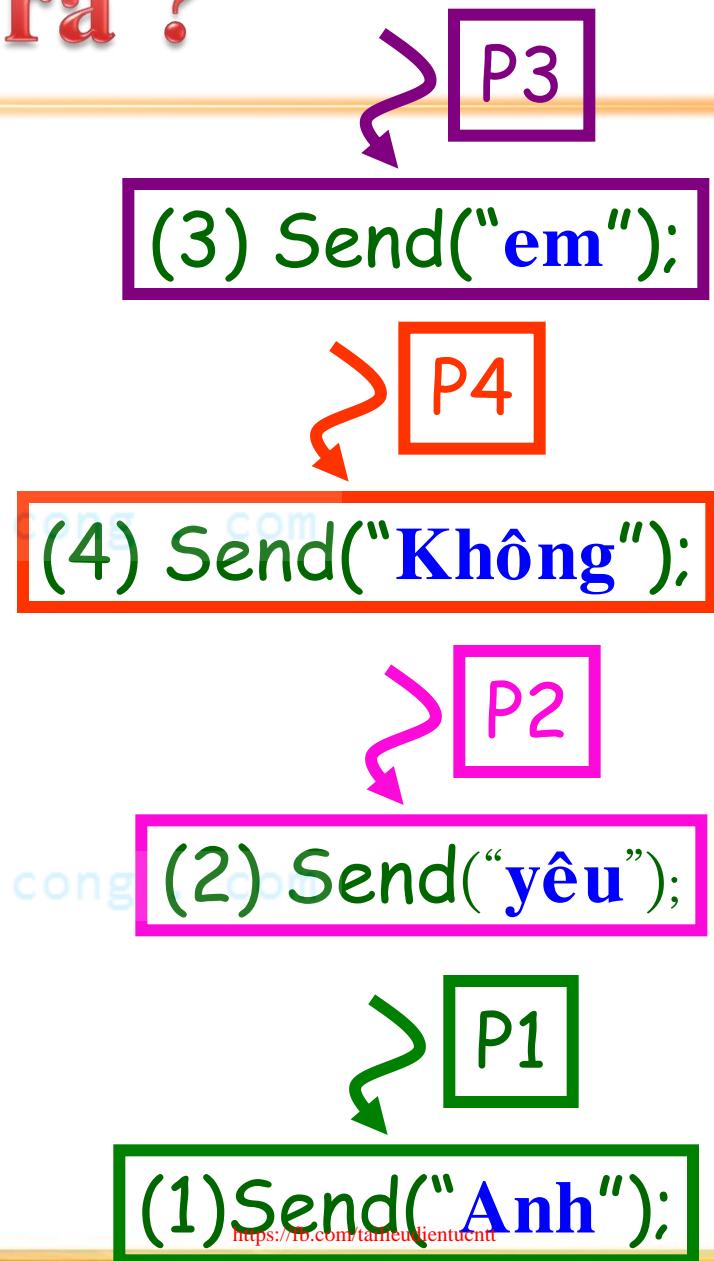
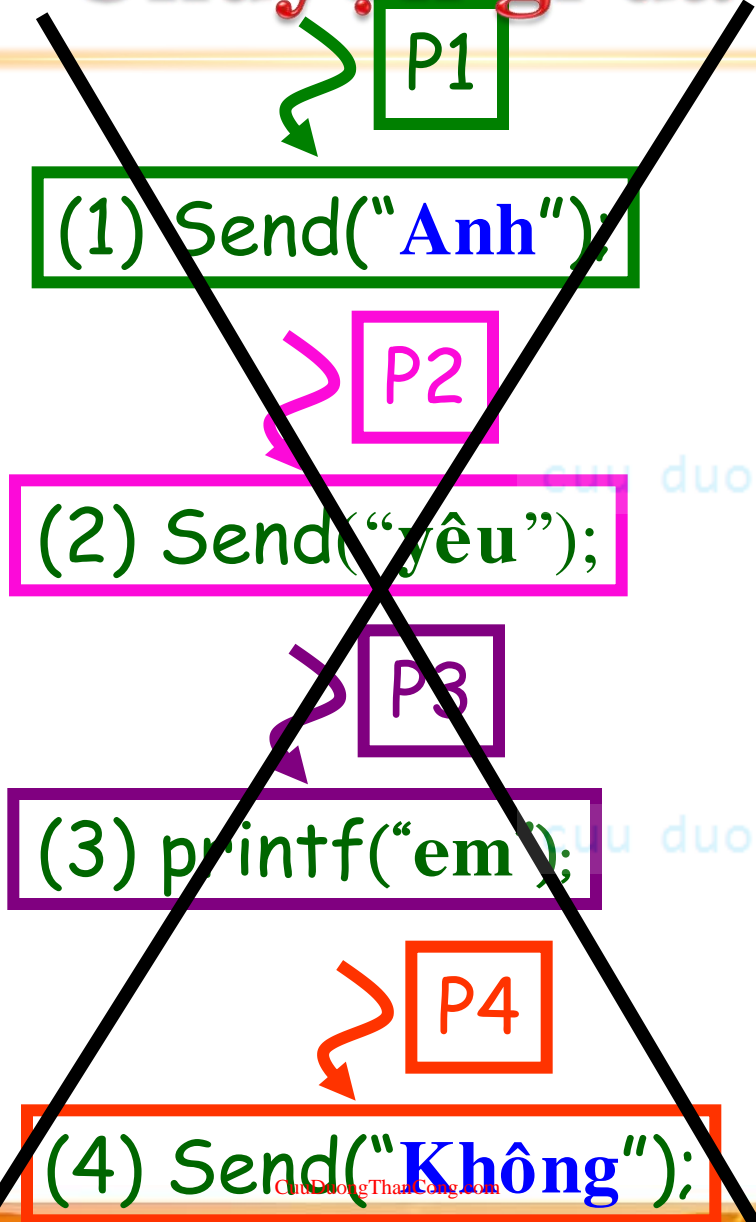
- Xử lý đồng hành và các vấn đề:
  - Vấn đề tranh đoạt điều khiển (Race Condition)
  - Vấn đề phối hợp xử lý
- Bài toán đồng bộ hóa
  - Yêu cầu độc quyền truy xuất (Mutual Exclusion)
  - Yêu cầu phối hợp xử lý (Synchronization)
- Các giải pháp đồng bộ hoá
  - Busy waiting
  - Sleep & Wakeup
- Các bài toán đồng bộ hoá kinh điển
  - Producer – Consumer
  - Readers – Writers
  - Dining Philosophers



# Phối hợp hoạt động

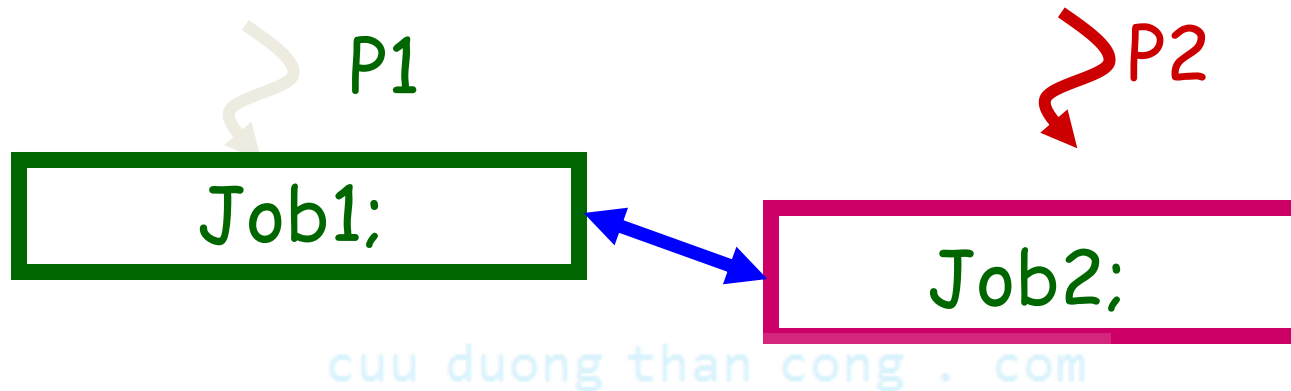


# Chuyện gì đã xảy ra ?





# Phối hợp xử lý



- Làm thế nào bảo đảm trình tự thực hiện Job1 - Job2 ?
  - P1 và P2 thực hiện “hẹn hò” (**Rendez-vous**) với nhau
- Hỗ trợ Rendez-vous : Bảo đảm các tiến trình phối hợp với nhau theo 1 trình tự xử lý định trước.



# Nội dung bài giảng

- Xử lý đồng hành và các vấn đề:
  - Vấn đề tranh đoạt điều khiển (Race Condition)
  - Vấn đề phối hợp xử lý
- Bài toán đồng bộ hóa
  - Yêu cầu độc quyền truy xuất (Mutual Exclusion)
  - Yêu cầu phối hợp xử lý (Synchronization)
- Các giải pháp đồng bộ hoá
  - Busy waiting
  - Sleep & Wakeup
- Các bài toán đồng bộ hoá kinh điển
  - Producer – Consumer
  - Readers – Writers
  - Dining Philosophers



# Bài toán đồng bộ hoá (Synchronization)

- Nhiều tiến trình chia sẻ tài nguyên chung đồng thời:
  - Tranh chấp  $\Rightarrow$  Race Condition
  - $\rightarrow$  Nhu cầu “độc quyền truy xuất” (**Mutual Exclusion**)
- Các tiến trình phối hợp hoạt động :
  - Tương quan diễn tiến xử lý ?
  - $\rightarrow$  Nhu cầu “hò hẹn” (**Rendez-vous**)
- Thực hiện đồng bộ hoá :
  - Lập trình viên đề xuất **chiến lược**
    - Các tiến trình liên quan trong bài toán phải tôn trọng các luật đồng bộ
  - Giải pháp sử dụng các **cơ chế** đồng bộ :
    - Do lập trình viên / phần cứng / HĐH / NNLT cung cấp



# Mô hình đảm bảo Mutual Exclusion

- **Nhiệm vụ của lập trình viên:**
  - Thêm các đoạn code đồng bộ hóa vào chương trình gốc
  - Thêm thế nào : xem mô hình sau ...

**Kiểm tra và dành quyền vào CS**

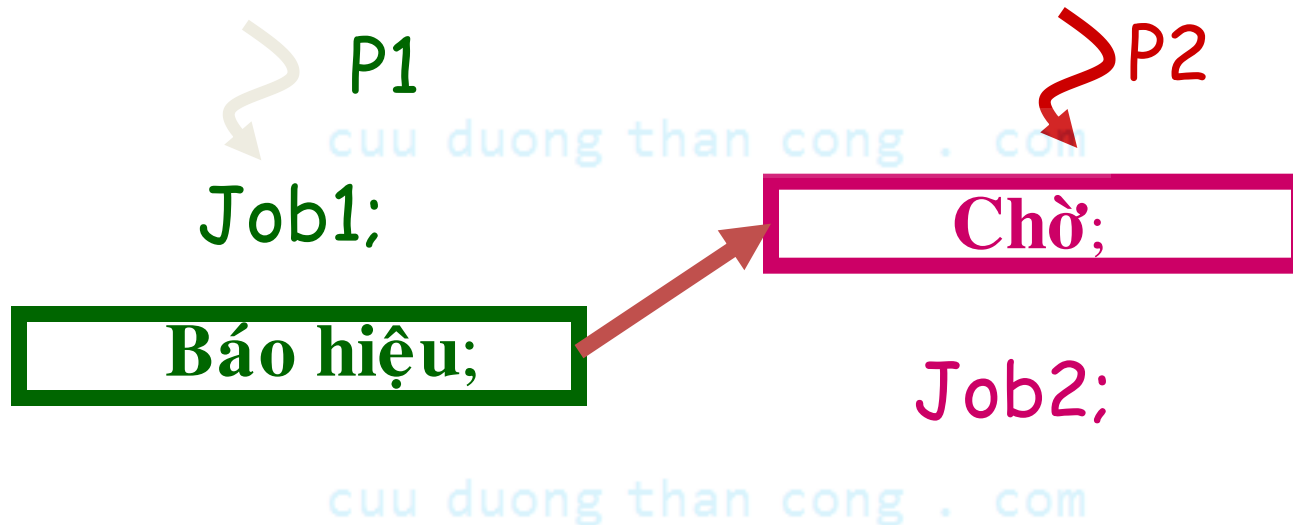
**CS;**

**Từ bỏ quyền sử dụng CS**



# Mô hình tổ chức phối hợp giữa hai tiến trình

- **Nhiệm vụ của lập trình viên:**
  - Thêm các đoạn code đồng bộ hóa vào 2 chương trình gốc
  - Thêm thế nào : xem mô hình sau ...



- **Nhiều tiến trình hơn thì sao ?**
  - Không có mô hình tổng quát
  - Tùy thuộc bạn muốn hện hò ra sao 😊



# Nội dung bài giảng

- Xử lý đồng hành và các vấn đề:
  - Vấn đề tranh đoạt điều khiển (Race Condition)
  - Vấn đề phối hợp xử lý
- Bài toán đồng bộ hóa
  - Yêu cầu độc quyền truy xuất (Mutual Exclusion)
  - Yêu cầu phối hợp xử lý (Synchronization)
- Các giải pháp đồng bộ hoá
  - Busy wating
  - Sleep & Wakeup
- Các bài toán đồng bộ hoá kinh điển
  - Producer – Consumer
  - Readers – Writers
  - Dinning Philosophers



# Giải pháp đồng bộ hoá

Một phương pháp giải quyết tốt bài toán đồng bộ hoá cần thoả mãn 4 điều kiện sau:

- **Mutual Exclusion** : Không có hai tiến trình cùng ở trong miền găng cùng lúc.
- **Progress** : Một tiến trình tạm dừng bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng
- **Bounded Waiting** : Không có tiến trình nào phải chờ vô hạn để được vào miền găng.
- Không có giả thiết nào đặt ra cho sự liên hệ về tốc độ của các tiến trình, cũng như về số lượng bộ xử lý trong hệ thống.





# Các giải pháp đồng bộ hoá

- Nhóm giải pháp **Busy Waiting**
  - Phần mềm
    - Sử dụng các biến cờ hiệu
    - Sử dụng việc kiểm tra luân phiên
    - Giải pháp của Peterson
  - Phần cứng
    - Cắm ngắt
    - Chỉ thị TSL
- Nhóm giải pháp **Sleep & Wakeup**
  - Semaphore
  - Monitor
  - Message



# Các giải pháp “Busy waiting”

**While (chưa có quyền) donothing() ;**

CS;

**Từ bỏ quyền sử dụng CS**

- Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng
- Không đòi hỏi sự trợ giúp của Hệ điều hành



# Nhóm giải pháp Busy-Waiting

- Các giải pháp Busy Waiting
  - Các giải pháp phần mềm
    - Giải pháp biến cờ hiệu
    - Giải pháp kiểm tra luân phiên
    - Giải pháp Peterson
  - Phần cứng
    - Cấm ngắt
    - Chỉ thị TSL



# Giải pháp phần mềm 1: Sử dụng biến cờ hiệu

int lock = 0

P0

NonCS;

```
while (lock == 1); // wait  
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;

P1

NonCS;

```
while (lock == 1); // wait  
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;



# Giải pháp phần mềm 1: Tình huống

int lock = 0

P0

NonCS;

```
while (lock == 1); // wait  
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;

P1

NonCS;

```
while (lock == 1); // wait  
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;



# Nhận xét Giải pháp phần mềm 1: Biến cờ hiệu

- Có thể mở rộng cho N tiến trình
- Không bảo đảm Mutual Exclusion
  - Nguyên nhân ?

Bị ngắt xử lý



```
while ( lock == 1); // wait  
lock = 1;
```

CS !

Tài nguyên dùng  
chung

- Bản thân đoạn code kiểm tra và dành quyền cũng là CS !



# Giải pháp phần mềm 2 : Kiểm tra luân phiên

int turn = 1

P0

NonCS;

while (turn != 0); // wait

CS;

turn = 1;

NonCS;

P1

NonCS;

while (turn != 1); // wait

CS;

turn = 0;

NonCS;





# Giải pháp phần mềm 2 : Tình huống

int turn = 1

P0

turn == 1  
Wait...  
CS;  
turn = 1  
NonCS;  
CS ? (turn == 1)

P1

CS;  
turn = 0;  
NonCS...

P0 không vào được CS lần 2 khi P1 dừng trong



# Nhận xét Giải pháp 2: Kiểm tra luân phiên

- Chỉ dành cho 2 tiến trình
- Bảo đảm Mutual Exclusion
  - Chỉ có 1 biến *turn*, tại 1 thời điểm chỉ cho 1 tiến trình *turn* vào CS
- Không bảo đảm Progress
  - Nguyên nhân ?
    - “Mở cửa” cho người = “Đóng cửa” chính mình !



# Giải pháp phần mềm 3 : Peterson's Solution

- Kết hợp ý tưởng của 1 & 2, các tiến trình chia sẻ:
  - int turn; *//đến phiên ai*
  - int interest[2] = FALSE; *//interest[i] = T : Pi muốn vào CS*

Pi

NonCS;

```
j = 1 - i;  
interest[i] = TRUE;  
turn = j;  
while (turn==j && interest[j]==TRUE);
```

CS;

```
interest[i] = FALSE;
```

NonCS;



# Giải pháp phần mềm 3 : Peterson

$P_j$

NonCS;

```
i = 1 - j;  
interest[j] = TRUE;  
turn = i;  
while (turn==i && interest[i]==TRUE);
```

CS;

```
interest[j] = FALSE;
```

NonCS;



# Nhận xét giải pháp phần mềm 3: Peterson

- Là giải pháp phần mềm đáp ứng được cả 3 điều kiện
  - Mutual Exclusion :
    - $P_i$  chỉ có thể vào CS khi:  $interest[j] == F$  hay  $turn == i$
    - Nếu cả 2 muốn về thì do  $turn$  chỉ có thể nhận giá trị 0 hay 1 nên chỉ có 1 tiến trình vào CS
  - Progress
    - Sử dụng 2 biến  $interest[i]$  riêng biệt  $\Rightarrow$  trạng thái đối phương không khoá mình được
  - Bounded Wait :  $interest[i]$  và  $turn$  đều có thay đổi giá trị

- Không thể mở rộng cho N tiến trình



# Nhận xét chung về các giải pháp phần mềm trong nhóm Busy-Waiting

- Không cần sự hỗ trợ của hệ thống
- Dễ...sai, Khó mở rộng
- Giải pháp 1 nếu có thể được hỗ trợ **atomicity** thì sẽ tốt...
  - Nhờ đến phần cứng ?

cuu duong than cong . com

cuu duong than cong . com



# Nhóm Busy-Waiting - Các giải pháp phần cứng

- Các giải pháp Busy Waiting
  - Các giải pháp phần mềm
    - Giải pháp biến cờ hiệu
    - Giải pháp kiểm tra luân phiên
    - Giải pháp Peterson
  - Các giải pháp phần cứng
    - Cấm ngắt
    - Test&Set lock Instruction





# Nhóm Busy-Waiting - Giải pháp phần cứng 1: Cấm ngắt

NonCS;

Disable Interrupt;

cuu duong thanh cong . com  
CS;

Enable Interrupt;

cuu duong thanh cong . com  
NonCS;

- **Disable Interrupt** : Cấm mọi ngắt, kể cả ngắt đồng hồ
- **Enable Interrupt** : Cho phép ngắt



# Giải pháp phần cứng 1: Cấm ngắt

- Thiếu thận trọng
  - Nếu tiến trình bị khoá trong CS ?
    - System Halt
  - Cho phép tiến trình sử dụng một lệnh đặc quyền
    - Quá ...liều !
- Máy có N CPUs ?
  - Không bảo đảm được Mutual Exclusion



# Nhóm Busy-Waiting - Giải pháp phần cứng 2: chỉ thị TSL()

- CPU hỗ trợ primitive **Test and Set Lock**
  - Trả về giá trị hiện hành của 1 biến, và đặt lại giá trị True cho biến
  - Thực hiện một cách không thể phân chia

**TSL** (boolean &target)

{

**TSL = target;**

**target = TRUE;**

}



# Áp dụng TSL

int lock = 0

Pi

NonCS;

while (TSL(lock)); // wait

CS;

lock = 0;

NonCS;



# Nhận xét chung các giải pháp phần cứng trong nhóm Busy-Waiting

- Cần được sự hỗ trợ của cơ chế phần cứng
  - Không dễ, nhất là trên các máy có nhiều bộ xử lý
- Dễ mở rộng cho  $N$  tiến trình



# Nhận xét chung cho các giải pháp trong nhóm Busy Waiting

- Sử dụng CPU không hiệu quả
  - Liên tục kiểm tra điều kiện khi chờ vào CS
- Khắc phục
  - **Khoá** các tiến trình chưa đủ điều kiện vào CS, **nhường CPU** cho tiến trình khác
    - Phải nhờ đến **Scheduler**
    - Wait and See...



# Các giải pháp đồng bộ hoá

- Nhóm giải pháp Busy Waiting
  - Phần mềm
    - Sử dụng các biến cờ hiệu
    - Sử dụng việc kiểm tra luân phiên
    - Giải pháp của Peterson
  - Phần cứng
    - Cấm ngắt
    - Chỉ thị TSL
- Nhóm giải pháp Sleep & Wakeup
  - Semaphore
  - Monitor
  - Message





# Các giải pháp “Sleep & Wake up”

if (chưa có quyền) Sleep() ;

CS;

Wakeup( somebody);

- Từ bỏ CPU khi chưa được vào CS
- Khi CS trống, sẽ được đánh thức để vào CS
- Cần được Hệ điều hành hỗ trợ
  - Vì phải thay đổi trạng thái tiến trình



# Ý tưởng

- Hệ Điều hành hỗ trợ 2 primitive :
  - **Sleep()** : Tiến trình gọi sẽ nhận trạng thái Blocked
  - **WakeUp(P)**: Tiến trình P nhận trạng thái Ready
- Áp dụng
  - Sau khi kiểm tra điều kiện sẽ vào CS hay gọi Sleep() tùy vào kết quả kiểm tra
  - Tiến trình vừa sử dụng xong CS sẽ đánh thức các tiến trình bị Blocked trước đó



# Áp dụng Sleep() and Wakeup()

- int busy; // busy ==0 : CS trống
- int blocked; // đếm số tiến trình bị Blocked chờ vào CS

```
if (busy) {  
    blocked = blocked + 1;  
    Sleep();  
}  
else busy = 1;
```

**CS;**

```
busy = 0;  
if(blocked) {  
    WakeUp(P);  
    blocked = blocked - 1;  
}
```



# Vấn đề với Sleep & WakeUp

P1

```
if (busy) {  
    blocked = blocked + 1;  
    Sleep();  
}  
else busy = 1;
```

CS:

```
busy = 0;  
if(blocked) {  
    WakeUp(P);  
    blocked = blocked - 1;  
}
```

P2

```
if (busy) {  
    blocked = blocked + 1;  
    Sleep();  
}  
else busy = 1;
```

CS:

```
busy = 0;  
if(blocked) {  
    WakeUp(P);  
    blocked = blocked - 1;  
}
```

P1  
blocked

WakeUp  
bị “lạc”

## ■ Nguyên nhân :

- Việc kiểm tra điều kiện và động tác từ bỏ CPU có thể bị ngắt quãng

# Cài đặt các giải pháp Sleep & WakeUp ?

- Hệ điều hành cần hỗ trợ các cơ chế cao hơn
  - Dựa trên Sleep&WakeUp
  - Kết hợp các yếu tố kiểm tra
  - Thi hành không thể phân chia
- Nhóm giải pháp Sleep & Wakeup
  - Semaphore
  - Monitor
  - Message

cuu duong than cong . com



# Giải pháp Sleep & Wakeup 1:

## Semaphore

- Được đề nghị bởi Dijkstra năm 1965
- Các đặc tính : Semaphore s;
  - Có 1 giá trị
  - Chỉ được thao tác bởi 2 primitives:
    - Down(s)
    - Up(s)
  - Các primitive Down và Up được thực hiện không thể phân chia

```
Semaphore s; // s >= 0
```



# Cài đặt Semaphore (Sleep & Wakeup)

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

Giá trị bên trong của semaphore

Danh sách các tiến trình đang bị block đợi semaphore nhận giá trị dương

- Semaphore được xem như là một resource
  - Các tiến trình “yêu cầu” semaphore : gọi Down(s)
    - Nếu không hoàn tất được Down(s): chưa được cấp resource
      - Blocked, được đưa vào s.L
- Cần có sự hỗ trợ của HĐH
  - Sleep() & Wakeup()





# Cài đặt Semaphore (Sleep & Wakeup)

Down (S)

```
{  
    S.value --;  
    if S.value < 0  
    {  
        Add(P, S.L);  
        Sleep();  
    }  
}
```

Up(S)

```
{  
    S.value ++;  
    if S.value ≤ 0  
    {  
        Remove(P, S.L);  
        Wakeup(P);  
    }  
}
```



# Sử dụng Semaphore

- Tổ chức “độc quyền truy xuất”  $P_i$

Semaphore  $s = 1$

Down (s)  
CS;  
Up(s)

- Tổ chức “hò hẹn”

Semaphore  $s = 0$

$P_1$  :  
Job1;  
Up(s)

$P_2$  :  
Down (s);  
Job2;

# Nhận xét Semaphores

- Là một cơ chế tốt để thực hiện đồng bộ
  - Dễ dùng cho N tiến trình
- Nhưng ý nghĩa sử dụng không rõ ràng
  - MutualExclusion : Down & Up
  - Rendez-vous : Down & Up
  - Chỉ phân biệt qua mô hình
- Khó sử dụng đúng
  - Nhầm lẫn



# Giải pháp Sleep & Wakeup 2: Monitor

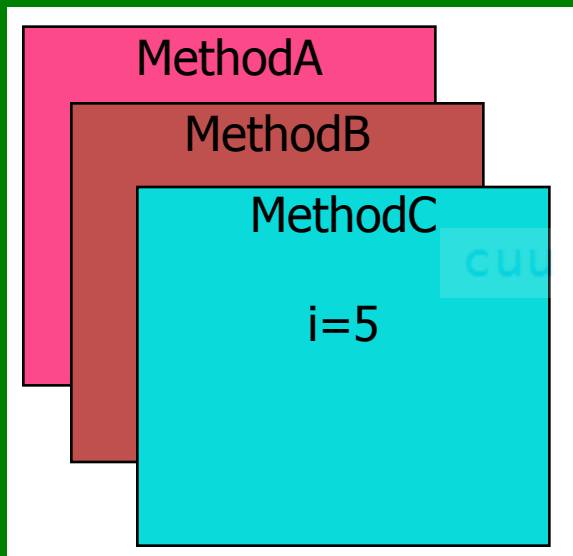
- Đề xuất bởi Hoare(1974) & Brinch (1975)
- Là cơ chế đồng bộ hoá do NNLT cung cấp
  - Hỗ trợ cùng các chức năng như Semaphore
  - Dễ sử dụng và kiểm soát hơn Semaphore
    - Bảo đảm Mutual Exclusion một cách tự động
    - Sử dụng biến điều kiện để thực hiện Synchronization



# Monitor : Ngữ nghĩa và tính chất(1)

## Monitor M

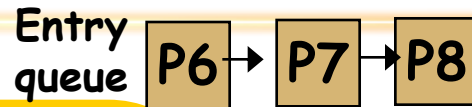
Share variable:  $i, j$ ;



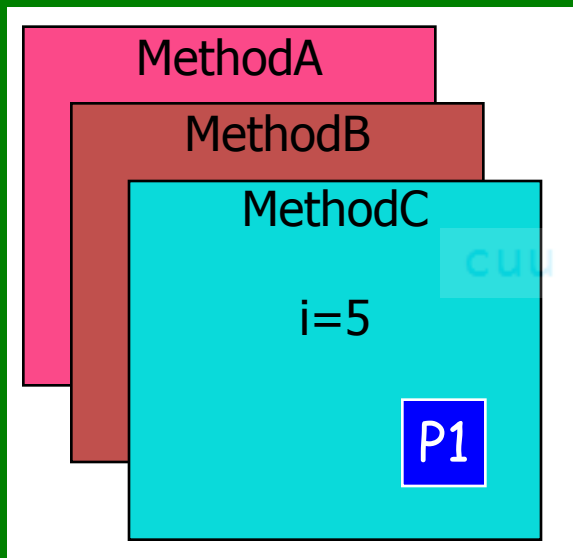
- Là một module chương trình định nghĩa
  - Các CTDL, **đối tượng** dùng chung
  - Các **phương thức** xử lý các đối tượng này
  - Bảo đảm tính **encapsulation**
- Các tiến trình muốn truy xuất dữ liệu bên trong monitor phải dùng các phương thức của monitor :
  - $P1 : M.C() \text{ // } i=5$
  - $P2: M.B() \text{ // printf}(j)$



# Monitor : Nghĩa và tính chất(2)



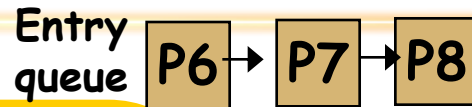
Share variable:  $i, j$ ;



- Tự động bảo đảm Mutual Exclusion
  - Tại 1 thời điểm chỉ có 1 tiến trình được thực hiện các phương thức của Monitor
  - Các tiến trình không thể vào Monitor sẽ được đưa vào Entry queue của Monitor
- Ví dụ
  - P1 : M.A();
  - P6 : M.B();
  - P7 : M.A();
  - P8 : M.C();






# Monitor : Ngữ nghĩa và tính chất(3)

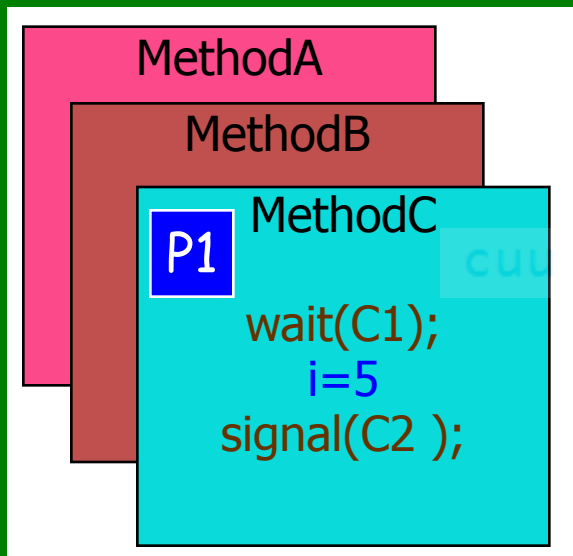


Share variable:  $i, j$ ;

Condition variable:

C1:  →  → 

C2:  → 



- Hỗ trợ Synchronization với các **condition variables**
  - **Wait(c)** : Tiến trình gọi hàm sẽ bị blocked
  - **Signal(c)**: Giải phóng 1 tiến trình đang bị blocked trên biến điều kiện **c**
  - **C.queue** : danh sách các tiến trình blocked trên **c**
- Trạng thái tiến trình sau khi gọi **Signal**?
  - Blocked. Nhường quyền vào monitor cho tiến trình được đánh thức
  - Tiếp tục xử lý hết chu kỳ, rồi blocked



# Sử dụng Monitor

## ■ Tổ chức “độc quyền truy xuất”

```
Monitor    M
<resource type> RC;
Function   AccessMutual
           CS; // access RC
```

```
Pi
M.AccessMutual(); //CS
```

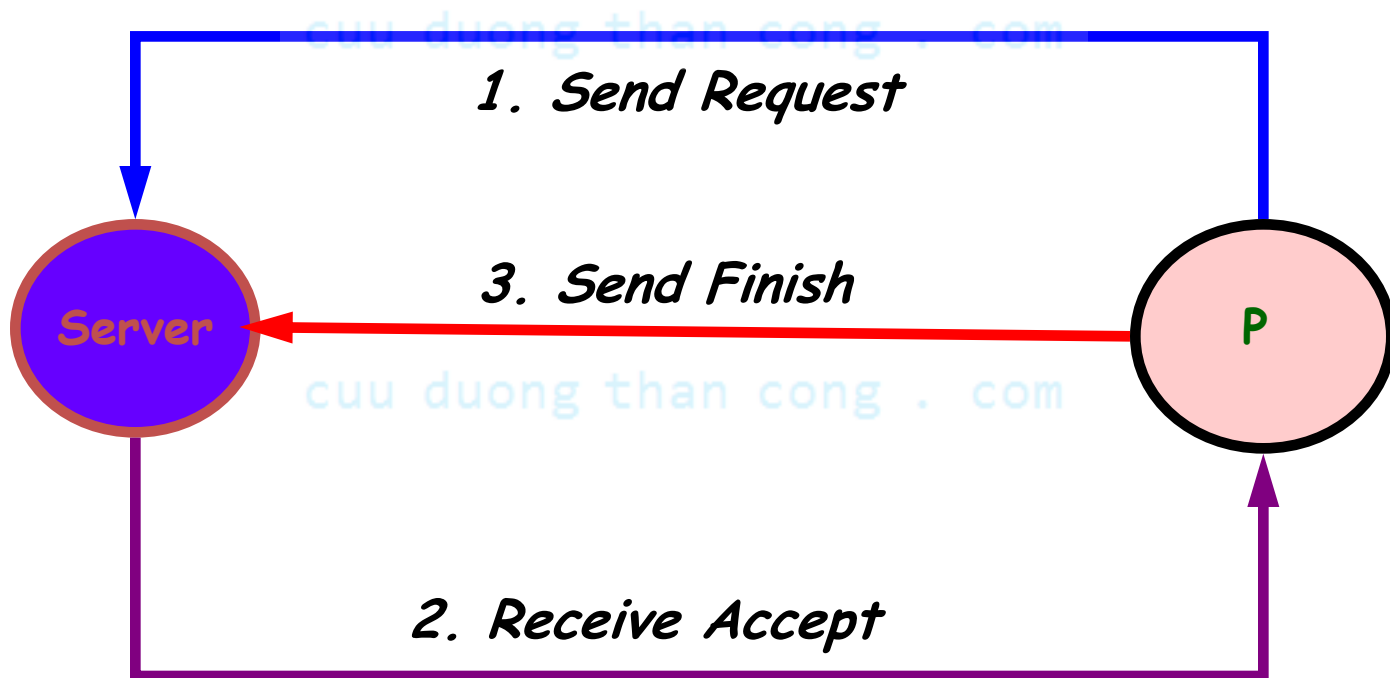
## ■ Tổ chức “hò hẹn”

```
Monitor    M
Condition  c;
Function   F1
           Job1;
           Signal(c);
Function   F2
           Wait(c);
           Job2;
```

```
P1 : M.F1(); → P2: M.F2();
```

# Giải pháp Sleep & Wakeup 3: Message

- Được hỗ trợ bởi HĐH
- Đồng bộ hóa trên môi trường phân tán
- 2 primitive Send & Receive
  - Cài đặt theo mode blocking



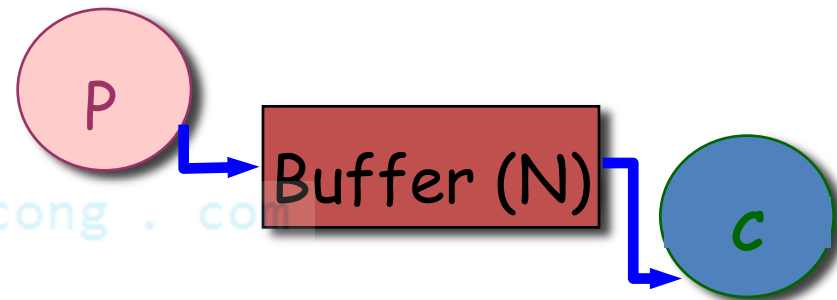
# Nội dung bài giảng

- Xử lý đồng hành và các vấn đề:
  - Vấn đề tranh đoạt điều khiển (Race Condition)
  - Vấn đề phối hợp xử lý
- Bài toán đồng bộ hóa
  - Yêu cầu độc quyền truy xuất (Mutual Exclusion)
  - Yêu cầu phối hợp xử lý (Synchronization)
- Các giải pháp đồng bộ hoá
  - Busy waiting
  - Sleep & Wakeup
- Các bài toán đồng bộ hoá kinh điển
  - Producer – Consumer
  - Readers – Writers
  - Dining Philosophers



# Bài toán đồng bộ kinh điển 1: Producer - Consumer (Bounded-Buffer Problem)

- Mô tả : 2 tiến trình P và C hoạt động đồng hành
  - P sản xuất hàng và đặt vào Buffer
  - C lấy hàng từ Buffer đi tiêu thụ
  - Buffer có kích thước giới hạn
- Tình huống
  - P và C đồng thời truy cập Buffer ?
  - P thêm hàng vào Buffer đầy ?
  - C lấy hàng từ Buffer trống ?



- P không được ghi dữ liệu vào buffer đã đầy (**Rendez-vous**)
- C không được đọc dữ liệu từ buffer đang trống (**Rendez-vous**)
- P và C không được thao tác trên buffer cùng lúc (**Mutual**

# Producer – Consumer : Giải pháp Semaphore

- Các biến dùng chung giữa P và C
  - `BufferSize = N;` // số chỗ trong bộ đệm
  - `semaphore mutex = 1 ;` // kiểm soát truy xuất độc quyền
  - `semaphore empty = BufferSize;` // số chỗ trống
  - `semaphore full = 0;` // số chỗ đầy
  - `int Buffer[BufferSize];` // bộ đệm dùng chung

cuu duong than cong . com



# Producer – Consumer : Giải pháp Semaphore

Producer()

```
{  
    int item;  
    while (TRUE)  
    {  
        produce_item(&item);  
        down(&empty);  
        down(&mutex)  
        enter_item(item, Buffer);  
        up(&mutex);  
        up(&full);  
    }  
}
```

Consumer()

```
{  
    int item;  
    while (TRUE)  
    {  
        down(&full);  
        down(&mutex);  
        remove_item(&item, Buffer);  
        up(&mutex);  
        up(&empty);  
        consume_item(item);  
    }  
}
```

# P&C - Giải pháp Semaphore: Thinking...

Producer()

```
{  
    int item;  
    while (TRUE)  
    {  
        produce_item(&item);  
        down(&mutex)  
        down(&empty);  
        enter_item(item, Buffer);  
        up(&mutex);  
        up(&full);  
    }  
}
```

Consumer()

```
{  
    int item;  
    while (TRUE)  
    {  
        down(&mutex);  
        down(&full);  
        remove_item(&item, Buffer);  
        up(&mutex);  
        up(&empty);  
        consume_item(item);  
    }  
}
```



# Producer – Consumer : Giải pháp Monitor

```
monitor ProducerConsumer
```

```
condition full, empty;
```

```
int Buffer[N], count;
```

```
procedure enter();
```

```
{
```

```
  if (count == N)
```

```
    wait(full);
```

```
  enter_item(item, Buffer);
```

```
  count ++;
```

```
  if (count == 1)
```

```
    signal(empty);
```

```
}
```

```
procedure remove();
```

```
{
```

```
  if (count == 0)
```

```
    wait(empty)
```

```
  remove_item(&item, Buffer);
```

```
  count --;
```

```
  if (count == N-1)
```

```
    signal(full);
```

```
}
```

```
count = 0;
```

```
end monitor;
```

# Producer – Consumer : Giải pháp Monitor

Producer()

```
{  
    int item;  
    while (TRUE)  
    {  
        produce_item(&item);  
        ProducerConsumer.enter;  
    }  
}
```

Consumer();

```
{  
    int item;  
    while (TRUE)  
    {  
        ProducerConsumer.remove;  
        consume_item(item);  
    }  
}
```

# Producer – Consumer : Giải pháp Message

Producer()

```
{
  int item;
  message m;

  while (TRUE)
  {
    produce_item(&item);
    receive(consumer, Request);
    create_message(&m, item);
    send(consumer, &m);
  }
}
```

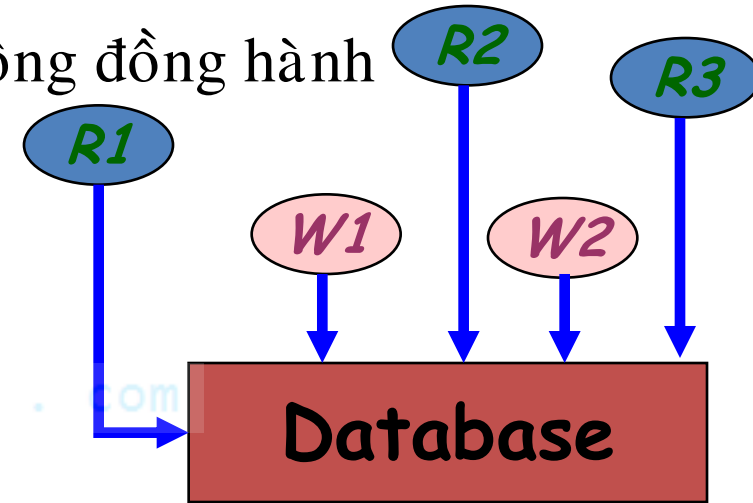
**Coi chừng  
Deadlock**

Consumer();

```
{
  int item;
  message m;
  for(0 to N)
    send(producer, Request);
  while (TRUE)
  {
    receive(producer, &m);
    remove_item(&m, &item);
    send(producer, Request);
    consumer_item(item);
  }
}
```

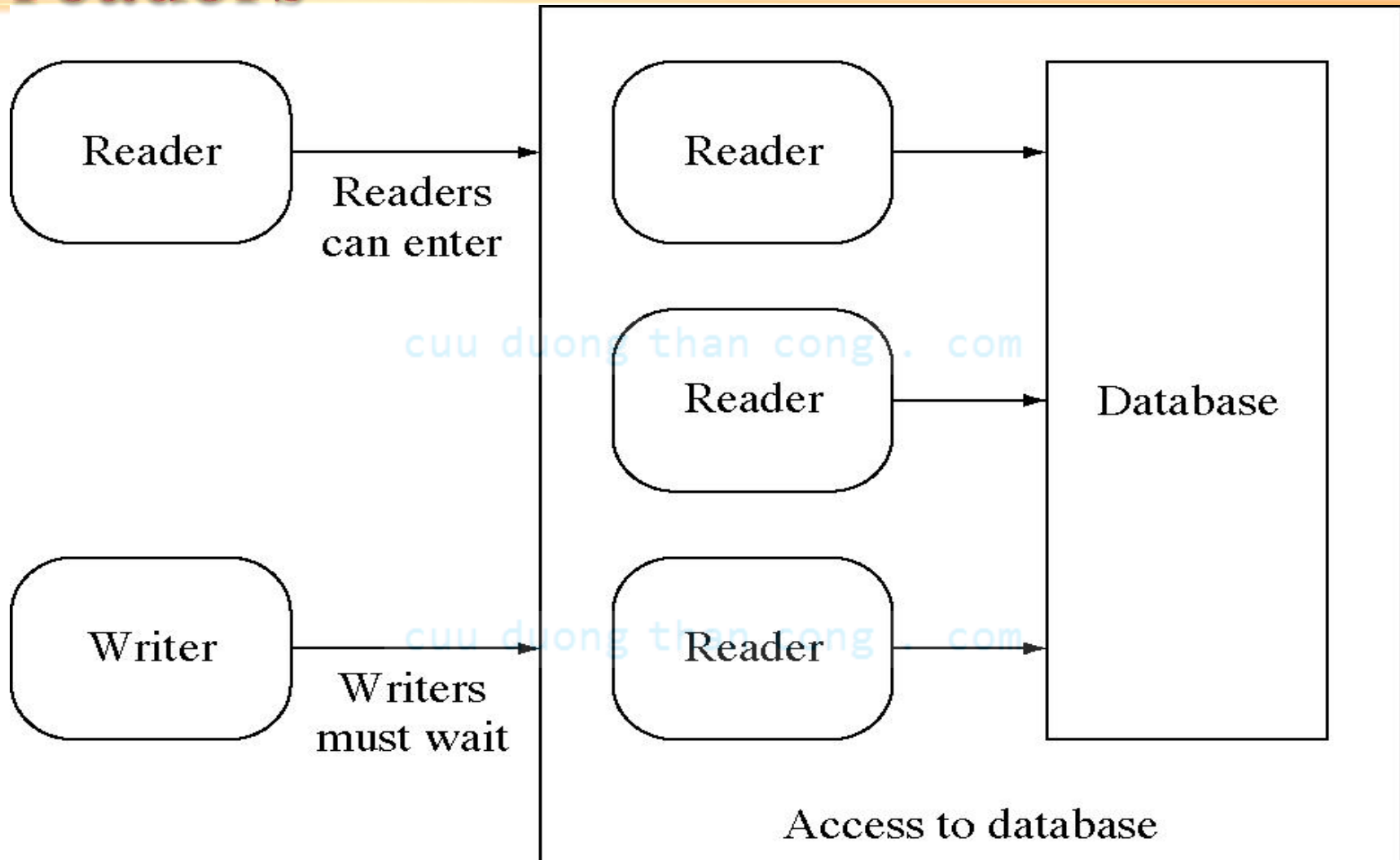
# Bài toán đồng bộ hoá kinh điển 2: Readers & Writers

- Mô tả : N tiến trình Ws và Rs hoạt động đồng hành
  - Rs và Ws chia sẻ CSDL
  - W cập nhật nội dung CSDL
  - Rs truy cập nội dung CSDL
- Tình huống
  - Các Rs cùng truy cập CSDL ?
  - W đang cập nhật CSDL thì các Rs truy cập CSDL ?
  - Các Rs đang truy cập CSDL thì W muốn cập nhật CSDL ?

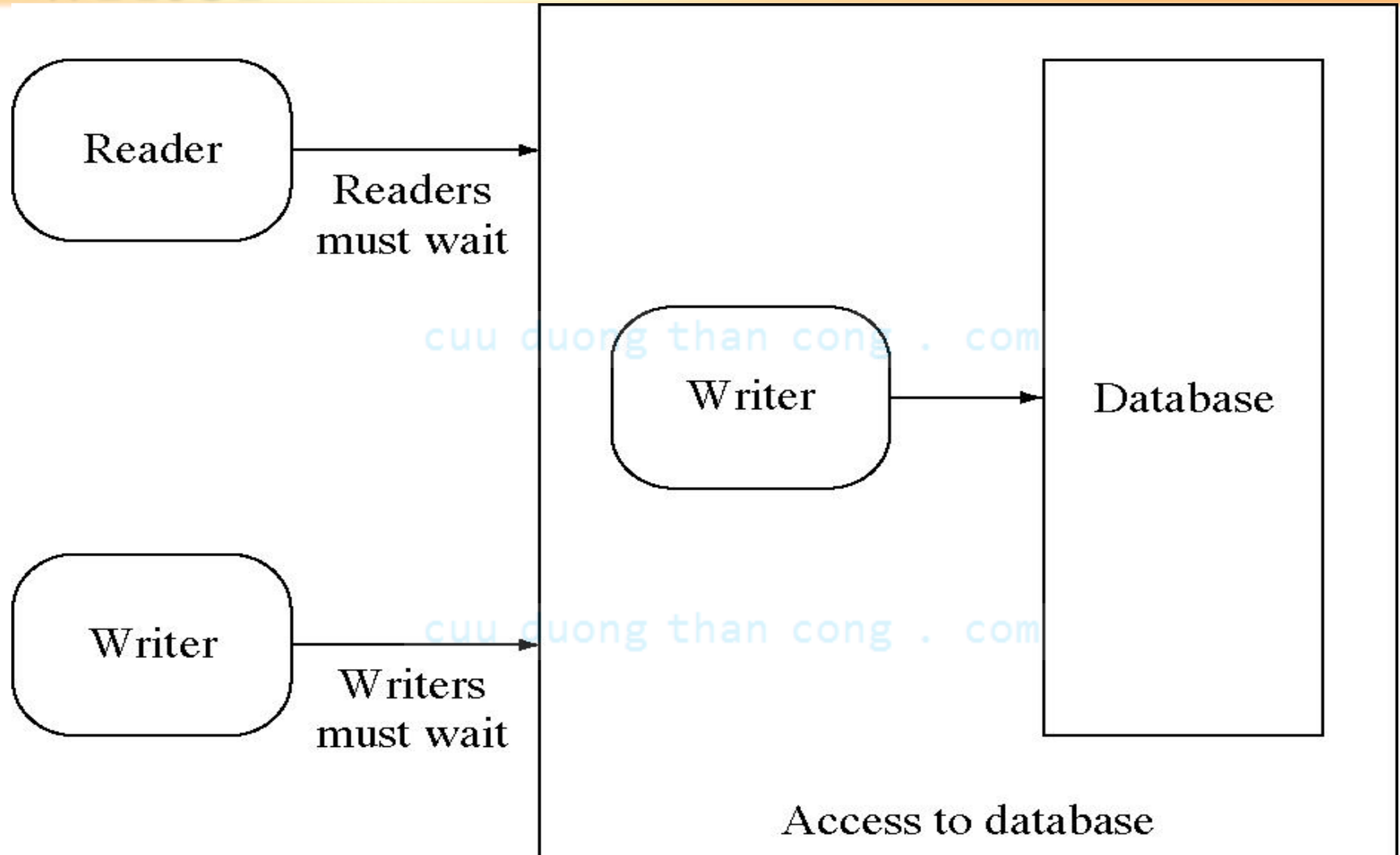


- W không được cập nhật dữ liệu khi có ít nhất một R đang truy xuất CSDL (ME)
- Rs không được truy cập CSDL khi một W đang cập nhật nội dung CSDL (ME)
- Tại một thời điểm, chỉ cho phép một W được sửa đổi nội dung CSDL (ME)

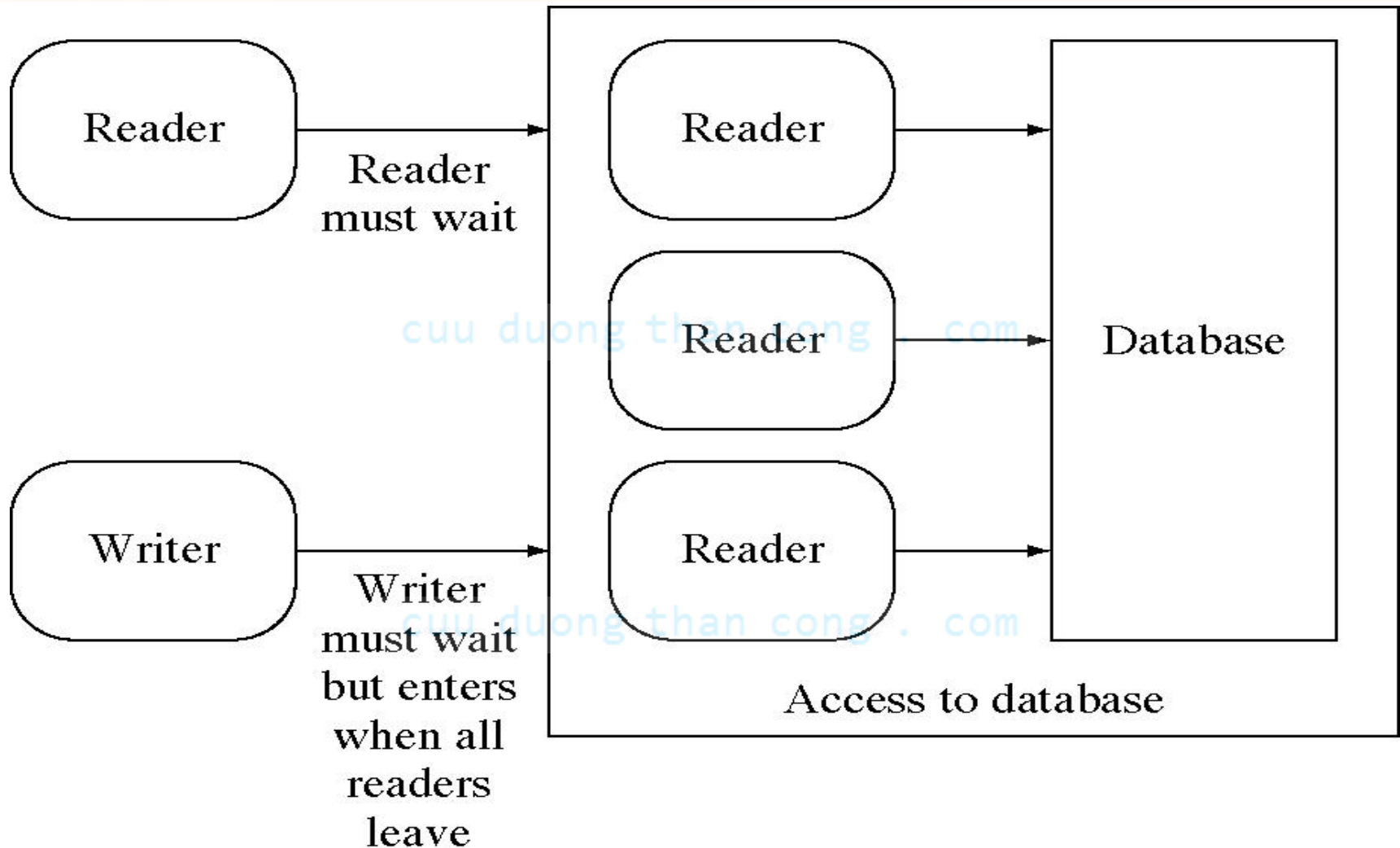
# Readers-Writers với “active readers”



# Readers-writers với một “active writer”



# Ưu tiên ai hơn đây ?





# Readers & Writers

- W độc quyền truy xuất CSDL
- W hiện tại kết thúc cập nhật CSDL : ai vào ?
  - Cho W khác vào, các Rs phải đợi
    - Ưu tiên Writer, Reader có thể starvation
  - Cho các Rs vào, Ws khác phải đợi
    - Ưu tiên Reader, Writer có thể starvation

cuu duong than cong . com



# Readers & Writers : Giải pháp Semaphore

- Các biến dùng chung giữa Rs và Ws
  - semaphore db = 1;      // Kiểm tra truy xuất CSDL

cuu duong than cong . com

cuu duong than cong . com



# R&W : Giải pháp Semaphore (1)

Reader()

```
{  
    down(&db);  
    read-db(Database);  
    up(&db);  
}
```

Writer()

```
{  
    down(&db);  
    write-db(Database);  
    up(&db);  
}
```

- Chuyện gì xảy ra ?
  - Chỉ có 1 Reader được đọc CSDL tại 1 thời điểm !

# R&W : Giải pháp Semaphore (2)

Reader()

```
{  
    if (rc == 0)  
        down(&db);  
    rc = rc + 1;  
    read-db(Database);  
    rc = rc - 1;  
    if (rc == 0)  
        up(&db);  
}
```

Writer()

```
{  
    down(&db);  
    write-db(Database);  
    up(&db);  
}
```

■ Đúng chưa ?

■ rc là biến dùng chung giữa các Reader...

■ CS đó ☹

# Readers & Writers : Giải pháp Semaphore

- Các biến dùng chung giữa Rs và Ws
  - semaphore db = 1; // Kiểm tra truy xuất CSDL
- Các biến dùng chung giữa Rs
  - int rc; // Số lượng tiến trình Reader
  - semaphore mutex = 1; // Kiểm tra truy xuất rc



# R&W : Giải pháp Semaphore (3)

Reader()

```
{  
    down(&mutex);  
    if (rc == 0)  
        down(&db);  
    rc = rc + 1;  
    up(mutex);  
    read-db(Database);  
    down(mutex);  
    rc = rc - 1;  
    if (rc == 0)  
        up(&db);  
    up(mutex);  
}
```

Writer()

```
{  
    down(&db);  
    write-db(Database);  
    up(&db);  
}
```

**Ai được ưu tiên ?**

# R&W : Giải pháp Semaphore (Thinking...)

Reader()

```
{  
    down(&mutex);  
    rc = rc + 1;  
    up(mutex);  
    if (rc == 1)  
        down(&db);  
    read-db(Database);  
    down(mutex);  
    rc = rc - 1;  
    up(mutex);  
    if (rc == 0)  
        up(&db);  
}
```

Writer()

```
{  
    down(&db);  
    write-db(Database);  
    up(&db);  
}
```

??? hê, hê, hê 😊



# R&W: Giải pháp Monitor

```
monitor ReaderWriter
```

```
? Database;
```

```
procedure R1();
```

```
{
```

```
}
```

```
procedure R...();
```

```
{
```

```
}
```

```
procedure W1();
```

```
{
```

```
}
```

```
procedure W...();
```

```
{
```

```
}
```

monitor **ReaderWriter**

condition OKWrite, OKRead;

int rc = 0;

Boolean busy = false;

procedure BeginRead()

{

if (busy)

**wait(OKRead);**

rc++;

**signal(OKRead);**

}

procedure FinishRead()

{

rc--;

if (rc == 0)

**signal(OKWrite);**

}

procedure BeginWrite()

{

if (busy || rc != 0)

**wait(OKWrite);**

busy = true;

}

procedure FinishWrite()

{

busy = false;

if (OKRead.Queue)

**signal(OKRead);**

else

**signal(OKWrite);**

}

end monitor;

# Reader&Writer : Giải pháp Monitor

Reader()

```
{  
    RW.BeginRead();  
    Read-db(Database);  
    RW.FinishRead();  
}
```

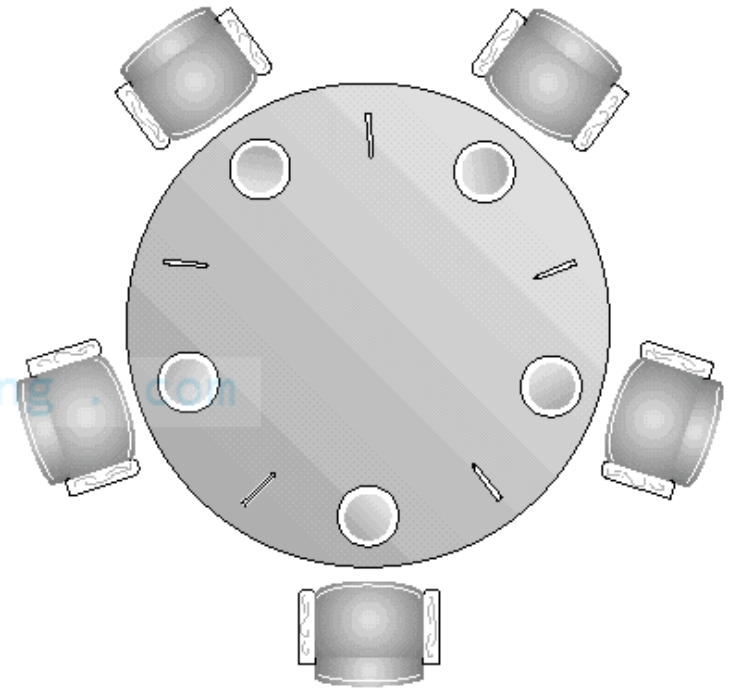
Writer();

```
{  
    RW.BeginWrite();  
    Write-db(Database);  
    RW.FinishWrite();  
}
```

# Bài toán đồng bộ hoá kinh điển 3:

## Bữa ăn của các Triết gia (Dining Philosophers)

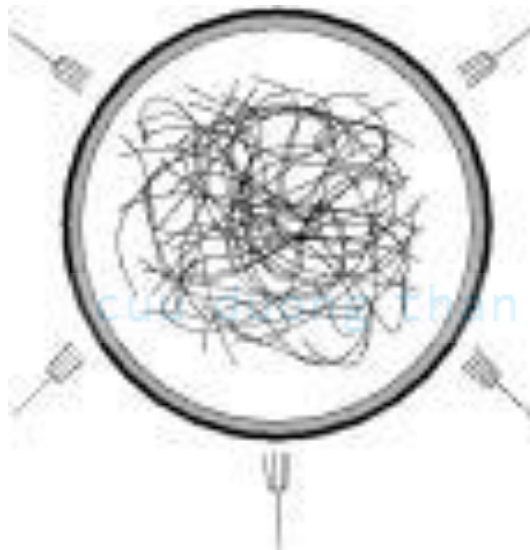
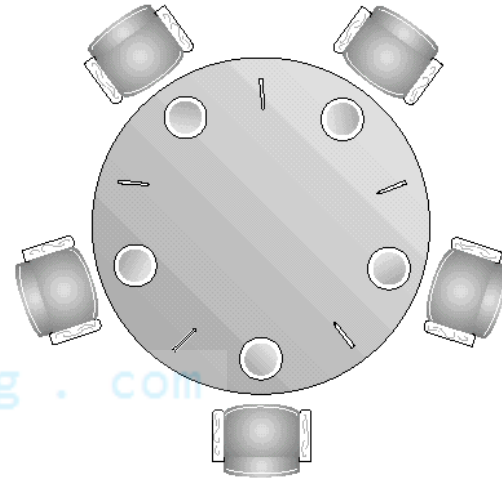
- Năm triết gia ngồi chung quanh bàn ăn món spaghetti (yum..yum)
  - Trên bàn có 5 cái nĩa được đặt giữa 5 cái đĩa (xem hình)
  - Để ăn món spaghetti mỗi người cần có 2 cái nĩa
- Triết gia thứ i:
  - Thinking...
  - Eating...



Chuyện gì có thể xảy ra ?

# Dining Philosophers : Tình huống nguy hiểm

- 2 triết gia “giành giật” cùng 1 cái nĩa
  - Tranh chấp
- Cần đồng bộ hoá hoạt động của các triết gia



# Dining Philosophers : Giải pháp đồng bộ

```
semaphore fork[5] = 1;
```

```
Philosopher (i)
```

```
{
```

```
while(true)
```

```
{
```

```
    down(fork[i]);
```

```
    down(fork[i+1 mod 5]);
```

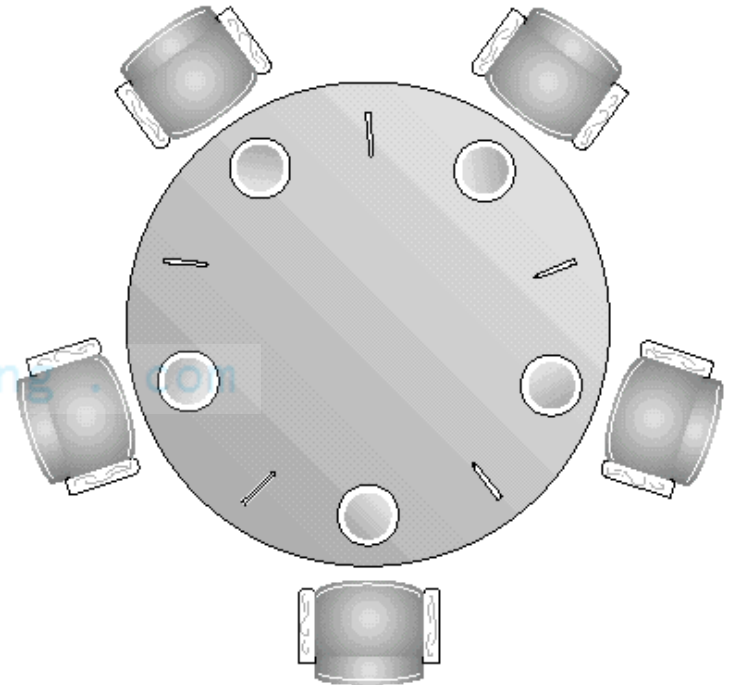
```
    eat;
```

```
    up(fork[i]);
```

```
    up(fork[i+1 mod 5]);
```

```
    think;
```

```
}
```



**Deadlock**

# Dining Philosophers : Thách thức

- Cần đồng bộ sao cho:
  - Không có deadlock
  - Không có starvation

cuu duong than cong . com

cuu duong than cong . com

