



Đồng Bộ Quá Trình

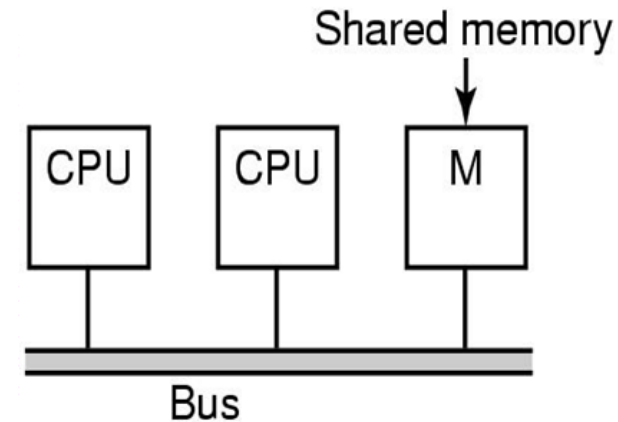


Nội dung

- Khái niệm cơ bản
- Tranh chấp “Critical section”
- Các giải pháp
 - Sử dụng lệnh máy thông thường
 - Giải thuật Peterson, và giải thuật bakery
 - Sử dụng lệnh cấm ngắt hoặc lệnh máy đặc biệt
 - Semaphore
 - Monitor

Bài toán đồng bộ

- Khảo sát các process/thread **thực thi đồng thời** và **chia sẻ dữ liệu** (ghi shared memory) trong hệ thống
 - **uniprocessor**, hoặc
 - **shared memory multiprocessor**
- Nếu không có sự kiểm soát khi truy cập các dữ liệu chia sẻ thì chúng có thể rơi vào tình trạng **không nhất quán** (inconsistent).
- Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế bảo đảm sự thực thi có trật tự của các process đồng thời.



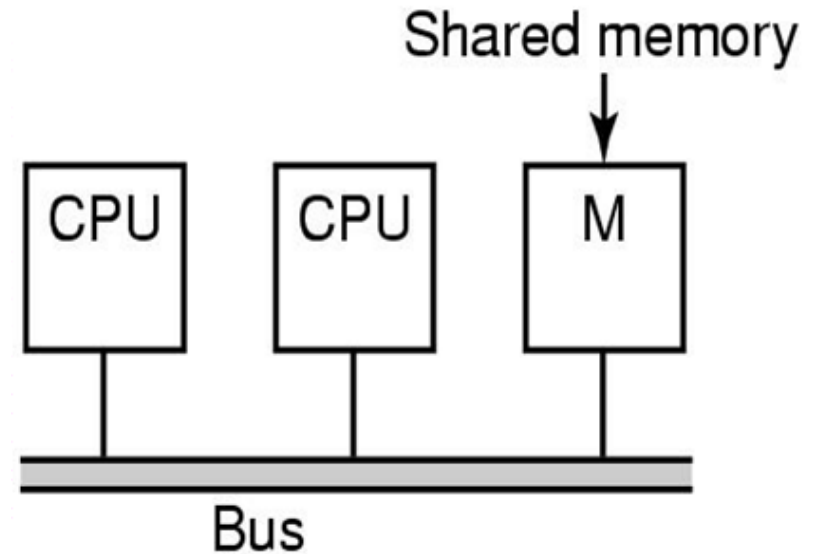


Bài toán đồng bộ (tt.)

- Hai lớp bài toán đồng bộ:
 - **Hợp tác** (cooperation)
 - Bài toán producer-consumer: bounded buffer
 - **Tranh giành** (contention)
 - Bài toán loại trừ tương hỗ: đồng bộ nhiều quá trình sử dụng một tài nguyên không chia sẻ đồng thời được (như printer)
 - Bài toán Dining Philosophers

Đồng thời vs. song song

- Trên uniprocessor hay trên shared memory multiprocessor, các quá trình chạy đồng thời
- Trên shared memory multiprocessor, các quá trình có thể chạy song song





Bài toán Producer-consumer

- Ví dụ Bounded buffer, thêm biến đếm count

```
#define BUFFER_SIZE 10          /* 10 buffers */  
typedef struct {  
    . . .  
} item;  
item buffer[BUFFER_SIZE];  
int in = 0, out = 0, count = 0;
```

Bài toán Producer-consumer (tt.)

■ Quá trình Producer

item nextProduced;

while(1) {

while (count == BUFFER_SIZE); /* do nothing */

buffer[in] = nextProduced;

count++;

in = (in + 1) % BUFFER_SIZE;

}

■ Quá trình Consumer

item nextConsumed;

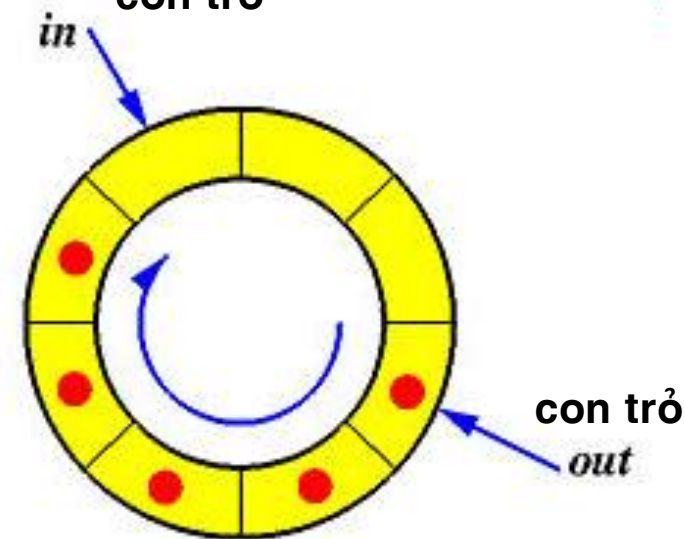
while(1) {

while (count == 0); /* do nothing */
nextConsumed = buffer[out];

count--;

out = (out + 1) % BUFFER_SIZE;

}



biến count được chia sẻ
giữa producer và consumer



Bài toán Producer-consumer (tt.)

- Các lệnh tăng/giảm biến count tương đương trong **ngôn ngữ máy** là:

Producer **count++:**

$register_1 = count$

$register_1 = register_1 + 1$

$count = register_1$

Consumer **count--:**

$register_2 = count$

$register_2 = register_2 - 1$

$count = register_2$

Trong đó, $register_i$ là thanh ghi của CPU.

Đồng bộ và lệnh đơn nguyên

- Mã máy của các lệnh tăng và giảm biến count có thể thực thi xen kẽ
- Giả sử count đang bằng 5. Chuỗi thực thi sau có thể xảy ra:

1:	producer	register ₁ := count	{register ₁ = 5}
	producer	register ₁ := register ₁ + 1	{register ₁ = 6}
2:	consumer	register ₂ := count	{register ₂ = 5}
	consumer	register ₂ := register ₂ - 1	{register ₂ = 4}
3:	producer	count := register ₁	{count = 6}
4:	consumer	count := register ₂	{count = 4}

Cả hai process thao tác đồng thời lên biến chung *count*. Trị của biến chung này không nhất quán dưới các thao tác của hai process.

Giải pháp: các lệnh count++, count-- phải là *đơn nguyên* (atomic), nghĩa là thực hiện như một lệnh đơn, không thực thi đan xen nhau.



Race condition

- *Race condition*: nhiều process truy xuất và thao tác đồng thời lên dữ liệu chia sẻ (*như biến count*); kết quả cuối cùng của việc truy xuất đồng thời này phụ thuộc thứ tự thực thi của các lệnh thao tác dữ liệu.
- Để dữ liệu chia sẻ được nhất quán, cần bảo đảm sao cho các process *lần lượt* thao tác lên dữ liệu chia sẻ. Do đó, cần có cơ chế **đồng bộ** hoạt động của các process này.



Khái niệm “Critical Section”

- Giả sử có n process đồng thời truy xuất dữ liệu chia sẻ.
- Không phải tất cả các đoạn code đều cần được giải quyết vấn đề race condition mà chỉ những đoạn code có chứa các thao tác lên dữ liệu chia sẻ. Đoạn code này được gọi là *vùng tranh chấp* (critical section, *CS*).
- **Bài toán loại trừ tương hỗ**: phải bảo đảm sự *loại trừ tương hỗ* (mutual exclusion, *mutex*), tức là khi một process P đang thực thi trong CS của P , không có process Q nào khác đồng thời thực thi các lệnh trong CS của Q .

Cấu trúc tổng quát của quá trình trong bài toán loại trừ tương hỗ

- Giả sử mỗi process thực thi bình thường (i.e., nonzero speed) và không có sự tương quan giữa tốc độ thực thi của các process
- Cấu trúc tổng quát của một process:

do {

entry section

critical section

exit section

remainder section

} while(1);

Một số giả định

- Có thể có nhiều CPU nhưng phần cứng không cho phép nhiều tác vụ truy cập một vị trí trong bộ nhớ cùng lúc (simultaneous)
- Không ràng buộc về thứ tự thực thi của các process
- Các process có thể chia sẻ một số biến chung nhằm đồng bộ hoạt động của chúng
- Giải pháp cần phải đặc tả **entry section** và **exit section**



Giải bài toán loại trừ tương hỗ

Lời giải phải thỏa 3 tính chất:

1. Mutual exclusion: Khi một process P đang thực thi trong vùng tranh chấp (CS) thì không có process Q nào khác đang thực thi trong CS.
2. Progress: Nếu không có quá trình nào đang thực thi trong vùng tranh chấp (CS) và có ít nhất một quá trình muốn vào vùng tranh chấp, thì chỉ có những quá trình đang không thực thi trong vùng remainder (RS) mới có quyền quyết định lựa chọn quá trình kế tiếp vào vùng tranh chấp và quyết định đó không được phép trì hoãn vô hạn định
3. Bounded waiting (*lockout-freedom*): Khi một quá trình muốn vào vùng tranh chấp (CS), thì từ khi yêu cầu đến khi được đáp ứng là khoảng thời gian có hạn (bounded or limit)



Phân loại giải pháp cho loại trừ tương hỗ

- Giải pháp dùng lệnh máy thông thường
- Giải pháp dùng lệnh cấm ngắt hay lệnh máy đặc biệt
 - Lệnh Disable interrupt
 - Lệnh máy đặc biệt như
 - TestAndSet



Giải pháp dùng lệnh máy thông thường

- Giải pháp cho 2 process đồng thời
 - Giải thuật 1 và 2 (Dekker1 &2)
 - Giải thuật Peterson cho 2 process
- Giải pháp cho n process
 - Giải thuật bakery



Giải thuật 1 (Dekker1)

- Biến chia sẻ

int turn; */* khởi đầu **turn = 0** */*

if **turn** = i then (P_i được phép vào critical section, với $i = 0$ hay 1)

- Process P_i

do {

while (turn != i);

critical section

turn = j;

remainder section

} while (1);

- Giải thuật thoả mãn mutual exclusion (1), nhưng **không** thoả mãn tính chất progress (2) vì tính chất strict alternation của giải thuật



Giải thuật 1(tt.)

(viết lại)

Process P0:

do

while (**turn** **!=** 0);

critical section

turn **:=** 1;

remainder section

while (1);

Process P1:

do

while (**turn** **!=** 1);

critical section

turn **:=** 0;

remainder section

while (1);

Giải thuật không thỏa mãn tính chất progress (2):

Nếu $turn = 0$, P0 được vào CS và sau đó thực thi $turn = 1$ và vào vùng RS; giả sử P0 “ở lâu” trong đó.

Lúc đó P1 vào CS và sau đó gán $turn = 0$, kể đó P1 vào và xong RS, vào entry section, đợi vào CS một lần nữa; nhưng vì $turn = 0$ nên P1 phải chờ P0.

Giải thuật 2 (Dekker2)

- Biến chia sẻ

boolean flag[2]; /* khởi đầu **flag[0] = flag[1] = false** */
if **flag[i] = true** then P_i “sẵn sàng” vào critical section.

- Process P_i

```
do {      flag[ i ] = true;      /*  $P_i$  “sẵn sàng” vào CS */
        while ( flag[ j ] );    /*  $P_i$  “nhường”  $P_j$  */
        critical section
        flag[ i ] = false;
        remainder section
    } while (1);
```

- Bảo đảm được mutual exclusion. Chứng minh?
- Không thỏa mãn **bounded wait(3)**. Vì sao? Trường hợp sau có thể xảy ra:

P_0 gán **flag[0] = true**

P_1 gán **flag[1] = true**

P_0 và P_1 loop mãi mãi trong vòng lặp while



Giải thuật Peterson (1981)

- Biến chia sẻ: kết hợp cả giải thuật 1 và 2
- Process P_i , với $i = 0$ hay 1

do {

```
flag[ i ] = true;    /* Process i sẵn sàng */  
favor = j;         /* Nhường process j */  
while (flag[ j ] and favor == j );  
critical section  
flag[ i ] = false;  
remainder section  
} while (1);
```

Giải thuật Peterson cho 2 process (tt.)

(viết lại)

Process P_0

do {

```
/* 0 wants in */
flag[0] = true;
/* 0 gives a chance to 1 */
favor = 1;
while (flag[1] &&
        favor == 1);
```

critical section

```
/* 0 no longer wants in */
flag[0] = false;
```

remainder section

} **while**(1);

Process P_1

do {

```
/* 1 wants in */
flag[1] = true;
/* 1 gives a chance to 0 */
favor = 0;
while (flag[0] &&
        favor == 0);
```

critical section

```
/* 1 no longer wants in */
flag[1] = false;
```

remainder section

} **while**(1);



Giải thuật Peterson cho 2 process: Tính đúng đắn

Giải thuật Peterson cho 2 process thỏa mutual exclusion, progress, và lockout-freedom

- Mutual exclusion được bảo đảm bởi vì
 - P0 và P1 đều ở trong CS nếu và chỉ nếu $\text{flag}[0] = \text{flag}[1] = \text{true}$ và $\text{turn} = i$ cho mỗi P_i (không thể xảy ra)
- Chứng minh thỏa yêu cầu về **progress(2)** và **bounded wait(3)**.
 - Xem tài liệu



Giải thuật bakery

- Trước khi vào CS, process P_i nhận một con số. Process nào giữ con số nhỏ nhất thì được vào CS
- Trường hợp P_i và P_j cùng nhận được một chỉ số:
 - Nếu $i < j$ thì P_i được vào trước.
- Khi ra khỏi CS, P_i đặt lại số của mình bằng 0
- Cách cấp số cho các process thường tạo các số tăng dần, ví dụ 1, 2, 3, 3, 3, 3, 4, 5,...
- Kí hiệu
 - $(a,b) < (c,d)$ nếu $a < c$ hoặc if $a = c$ và $b < d$
 - $\max(a_0, \dots, a_k)$ là con số b sao cho $b \geq a_i$ với mọi $i = 0, \dots, k$



Giải thuật bakery (tt.)

```
/* shared variable */
boolean    choosing[ n ];    /* initially, choosing[ i ] = false */
int        num[ n ];        /* initially, num[ i ] = 0 */

do {
    choosing[ i ] = true;
    num[ i ]      = max(num[0], num[1],..., num[n - 1]) + 1;
    choosing[ i ] = false;
    for (j = 0; j < n; j++) {
        while (choosing[ j ]);
        while ((num[ j ] != 0) && (num[ j ], j) < (num[ i ], i));
    }
    critical section
    num[ i ] = 0;
    remainder section
} while (1);
```



Đánh giá

- Các giải pháp dùng lệnh máy thông thường
 - Các process khi yêu cầu được vào vùng tranh chấp đều phải liên tục kiểm tra điều kiện (**busy waiting**), tốn thời gian xử lý của CPU.
 - Nếu thời gian xử lý trong vùng tranh chấp lớn, một giải pháp hiệu quả nên có cơ chế block các process cần đợi
- Các giải pháp dùng lệnh cấm ngắt hay dùng các lệnh máy đặc biệt → slide tiếp theo



Dùng lệnh cấm ngắt

Process P_i :

do {

`disable_interrupts();`

critical section

`enable_interrupts();`

remainder section

} while (1);

- Trong hệ thống **uniprocessor**: mutual exclusion được bảo đảm.
 - Nhưng nếu system clock được cập nhật do interrupt thì ...
- Trên hệ thống **multiprocessor**: mutual exclusion không được đảm bảo vì
 - Chỉ cấm ngắt tại CPU thực thi lệnh `disable interrupts`
 - Các CPU khác vẫn có thể truy cập bộ nhớ chia sẻ



Dùng các lệnh máy đặc biệt

- Ý tưởng

- Việc truy xuất vào vào một địa chỉ của bộ nhớ vốn đã có tính loại trừ tương hỗ (chỉ có một thao tác truy xuất tại một thời điểm)

- Mở rộng

- Thiết kế một lệnh **máy đơn nguyên** có thể thực hiện hai thao tác trên cùng một ô nhớ (vd: read và write)
- Việc thực thi các lệnh máy như trên luôn bảo đảm mutual exclusion (ngay cả với multiprocessor)

- Các lệnh máy đặc biệt có thể đảm bảo mutual exclusion nhưng cần kết hợp với một số cơ chế khác để thoả mãn progress, tránh starvation và deadlock.



Lệnh TestAndSet

- Đọc và ghi một biến chia sẻ bằng một lệnh đơn nguyên

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

Áp dụng TestAndSet

- Shared data:
boolean lock = false;
- Process P_i :
do {
 while (TestAndSet(lock));
 critical section
 lock = false;
 remainder section
} while (1);



Lệnh TestAndSet (tt.)

- Mutual exclusion được bảo đảm: nếu P_i vào CS, các process P_j khác đều đang busy waiting
- Khi P_i ra khỏi CS, sự chọn lựa process P_j vào CS kế tiếp là tùy ý \Rightarrow *starvation* (bounded wait)
- Các processor (ví dụ Pentium) thông thường cung cấp một lệnh máy đơn nguyên là Swap(a, b) có tác dụng hoán chuyển nội dung của a và b.
- Swap(a, b) cũng có ưu nhược điểm như TestAndSet

Lệnh Swap

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

Áp dụng lệnh Swap

- Biến chia sẻ **lock** khởi tạo là false
- Mỗi process P_i có biến cục bộ **key**;
process P_i nào thấy giá trị **lock**
= false thì được vào CS.
 - Process P_i sẽ loại trừ các process
 P_j khác khi thiết lập **lock = true**

- Biến chia sẻ (khởi tạo là **false**)
bool lock;

Process P_i :

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

Không thỏa mãn starvation freedom

Giải thuật dùng TestAndSet: thoả mãn 3 yêu cầu

Biến chia sẻ,
khởi tạo là false

```
bool waiting[ n ];  
bool lock;
```

```
do {
```

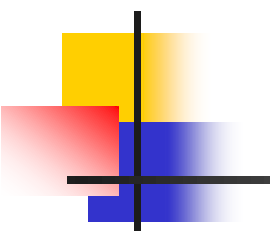
```
    waiting[ i ] = true;  
    key = true;  
    while (waiting[ i ] && key)  
        key = TestAndSet(lock);  
    waiting[ i ] = false;
```

critical section

```
    j = ( i + 1 ) % n;  
    while (( j != i ) && !waiting[ j ])  
        j = ( j + 1 ) % n;  
    if ( j == i )  
        lock = false;  
    else  
        waiting[ j ] = false;
```

remainder section

```
} while (1)
```



Giải thuật dùng TestAndSet: thoả mãn 3 yêu cầu (tt.)

- Mutual exclusion: P_i chỉ có thể vào CS nếu và chỉ nếu hoặc $waiting[i] = false$, hoặc $key = false$
 $key = false$ chỉ khi TestAndSet (hay Swap) được thực thi
 - Process đầu tiên thực thi TestAndSet mới có $key == false$; các process khác đều phải đợi $waiting[i] = false$ chỉ khi process khác rời khỏi CS
 - Chỉ có một $waiting[i]$ có giá trị false
- Progress
- Lockout-freedom: waiting in the cyclic order



Semaphore

Semaphore là công cụ đồng bộ cung cấp bởi OS.

- Một thuộc tính của *semaphore* là trị của nó, ngoài thao tác khởi động biến thì chỉ có thể được truy xuất qua hai tác vụ
 - *wait(S)* hay còn gọi là P(S): giảm trị semaphore, nếu trị này âm thì process gọi lệnh bị blocked.
 - *signal(S)* hay còn gọi là V(S): tăng trị semaphore, nếu trị này không dương, một process đang blocked bởi gọi lệnh *wait()* trước đó sẽ được hồi phục để thực thi.
- Semaphore giúp process tránh busy waiting: khi phải đợi thì process sẽ được đặt vào một blocked queue, trong đó chứa các process đang chờ đợi cùng một sự kiện.
 - Nhưng có thể cần busy waiting để hiện thực chính semaphore



Hiện thực semaphore

- Hiện thực semaphore là một record

```
typedef struct {
```

```
    int value;
```

```
    struct process *L; /* process queue */
```

```
} semaphore;
```

cùng với các tác vụ lên nó

- Giả sử hệ điều hành cung cấp hai tác vụ:
 - **block()**: tạm treo process nào thực thi lệnh này
 - **wakeup(P)**: hồi phục quá trình thực thi của process P đang blocked



Hiện thực semaphore (tt.)

- Các tác vụ semaphore được hiện thực như sau

```
void wait(semaphore S) {  
    S.value--;  
    if (S.value < 0) {  
        add this process to S.L;  
        block();  
    }  
}  
  
void signal(semaphore S) {  
    S.value++;  
    if (S.value <= 0) {  
        remove a process P from S.L;  
        wakeup(P);  
    }  
}
```



Hiện thực semaphore (tt.)

- Khi trị của $S \leq 0$, thì process gọi `wait(S)` sẽ bị blocked và được đặt trong hàng đợi semaphore - thường là hàng đợi FIFO
 - Hàng đợi này là danh sách liên kết các PCB
- Nếu trị của $S < 0$, tác vụ `signal(S)` chọn một process từ S.L và đưa nó vào hàng đợi ready
- `block()` và `wakeup()` thay đổi trạng thái của process
 - **block**: chuyển từ running sang waiting
 - **wakeup**: chuyển từ waiting sang ready

Ứng dụng semaphore: hiện thực mutex

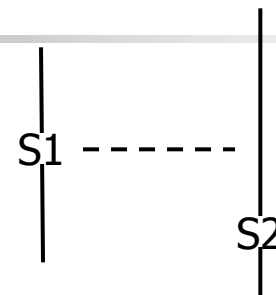
- Dùng cho n process
- Khởi tạo $S.value = 1$
Chỉ một process được thực thi trong CS (mutual exclusion)
- Để cho phép k process được thực thi trong CS, khởi tạo $S.value = k$

```
■ Shared data:  
semaphore mutex;  
/* initially mutex.value = 1  
*/  
■ Process  $P_i$ :  
  
do {  
    wait(mutex);  
    critical section  
    signal(mutex);  
    remainder section  
} while (1);
```



Ứng dụng semaphore: đồng bộ process

- Hai process: P1 và P2
- Yêu cầu: lệnh S1 trong P1 cần được thực thi **trước** lệnh S2 trong P2
- Định nghĩa semaphore synch để đồng bộ
- Khởi động semaphore:
synch.value = **0**



- Để đồng bộ hoạt động theo yêu cầu, P1 phải định nghĩa như sau:
S1;
signal(synch);
- Và P2 định nghĩa như sau:
wait(synch);
S2;



Nhận xét về semaphore

- Khi $S.value \geq 0$: số process có thể thực thi wait(S) mà không bị blocked = $S.value$
- Khi $S.value < 0$: số process đang đợi trên S là $|S.value|$



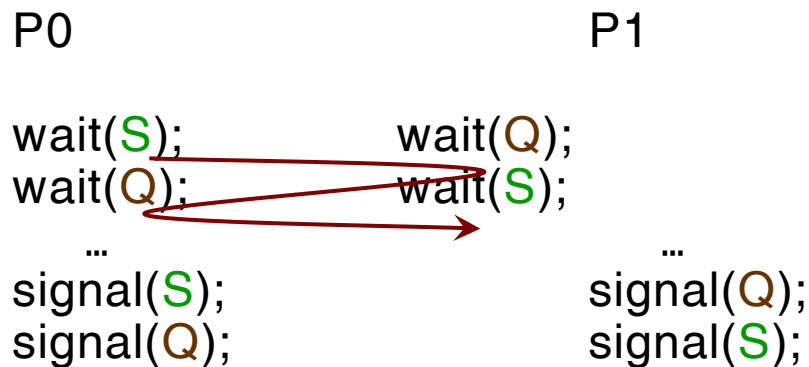
Nhận xét về semaphore (tt.)

- Cấu trúc dữ liệu hiện thực semaphore là biến chia sẻ ⇒ đoạn mã hiện thực các lệnh wait và signal là vùng tranh chấp
- Vùng tranh chấp của các tác vụ wait và signal thông thường rất nhỏ: khoảng 10 lệnh máy.
- Giải pháp cho vùng tranh chấp wait và signal
 - **Uniprocessor**: có thể dùng cấm ngắt (disable interrupt). Nhưng phương pháp này không thực hiện được trên hệ thống multiprocessor.
 - **Multiprocessor**: có thể dùng các giải pháp dùng lệnh máy thông thường (như giải thuật bakery) hoặc giải pháp dùng lệnh máy đặc biệt.

Vì CS rất ngắn nên chi phí cho busy waiting sẽ rất thấp.

Deadlock và starvation

- **Deadlock**: hai hay nhiều process chờ vô hạn định một sự kiện không bao giờ xảy ra, vd sự kiện do một trong các process đang đợi tạo ra.
- Ví dụ deadlock: Gọi S và Q là hai biến semaphore được khởi tạo = 1



P0 thực thi wait(S), rồi P1 thực thi wait(Q), rồi P0 thực thi wait(Q) bị blocked, P1 thực thi wait(S) bị blocked.

- **Starvation**: indefinite blocking, có thể xảy ra khi process vào hàng đợi và được lấy ra theo cơ chế LIFO.



Các loại semaphore

- *Counting semaphore*: một số nguyên có trị không giới hạn.
- *Binary semaphore*: có trị là 0 hay 1. Binary semaphore rất dễ hiện thực.
- Có thể hiện thực counting semaphore bằng binary semaphore.

Semaphore và bài toán bounded buffer

- Giải thuật cho bài toán bounded buffer

- Dữ liệu chia sẻ:

`semaphore full, empty, mutex;`

- Khởi tạo:

`full = 0; /* số buffers đầy */`

`empty = n; /* số buffers trống */`

`mutex = 1;`

n buffers



out

Semaphore và bài toán bounded buffer (tt.)

producer

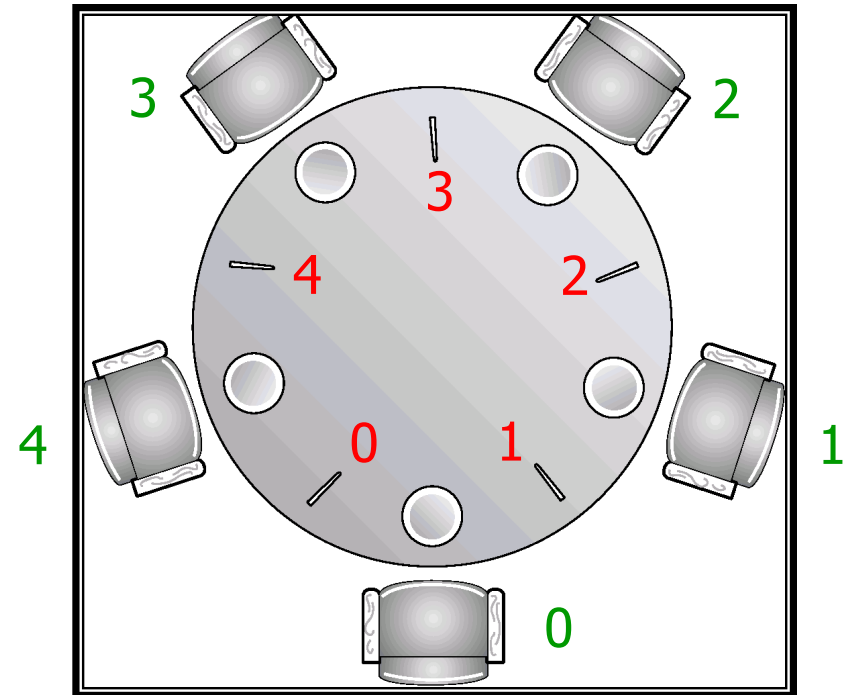
```
do {  
    ...  
    nextp = new_item();  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    insert_to_buffer(nextp);  
    ...  
    signal(mutex);  
    signal(full);  
} while (1);
```

consumer

```
do {  
    wait(full);  
    wait(mutex);  
    ...  
    nextc =  
    get_buffer_item(out);  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    consume_item(nextc);  
    ...  
} while (1);
```

Bài toán “Dining Philosophers”

- 5 triết gia ngồi ăn và suy nghĩ
 - Mỗi người cần 2 chiếc đũa (chopstick) để ăn
 - Trên bàn chỉ có 5 đũa
- “Dining philosophers” thuộc về lớp các bài toán **phân phối tài nguyên** giữa các process sao cho không xảy ra deadlock và starvation



Bài toán "Dining Philosophers" (tt.)

Giải thuật

Dữ liệu chia sẻ:

`semaphore chopstick[5];`

Khởi đầu các biến đều là **1**

Triết gia thứ i :

do {

wait(chopstick[i])

wait(chopstick[$(i + 1) \% 5$])

...

eat

...

signal(chopstick[i]);

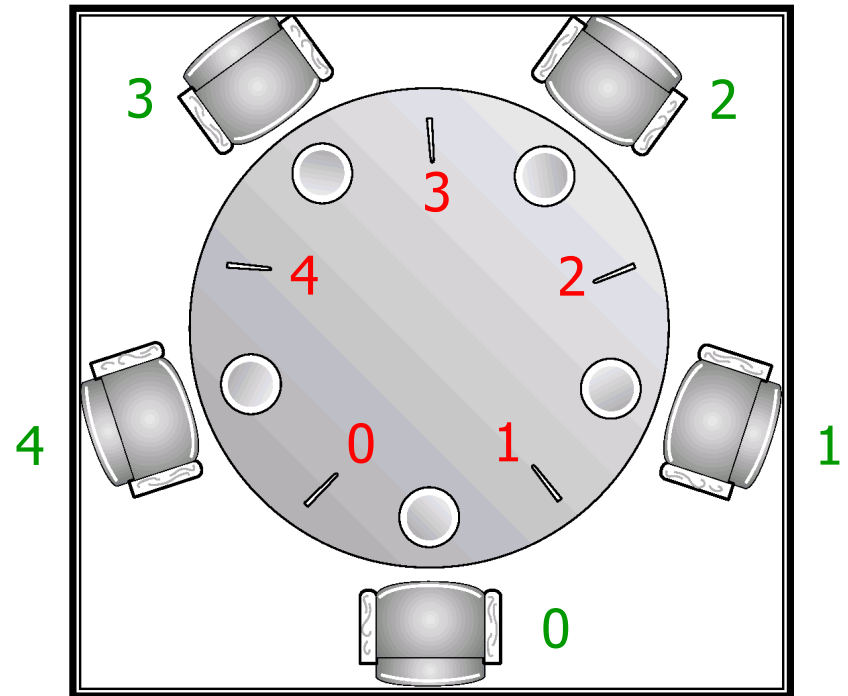
signal(chopstick[$(i + 1) \% 5$]);

...

think

...

} while (1);





Bài toán “Dining Philosophers” (tt.)

- Giải pháp trên có thể gây ra deadlock
 - Khi tất cả triết gia đồng thời cầm chiếc đũa bên tay trái rồi lấy đũa tay phải \Rightarrow **deadlock**
- Một số giải pháp giải quyết được deadlock
 - Cho phép nhiều nhất 4 triết gia ngồi vào bàn
 - Cho phép triết gia cầm các đũa chỉ khi cả hai chiếc đũa đều sẵn sàng (nghĩa là tác vụ cầm các đũa phải xảy ra trong CS)
 - Triết gia ngồi ở vị trí lẻ cầm đũa bên trái trước, sau đó mới đến đũa bên phải, trong khi đó triết gia ở vị trí chẵn cầm đũa bên phải trước, sau đó mới đến đũa bên trái

■ Starvation?

Bài toán Readers-Writers

Giải thuật

■ Dữ liệu chia sẻ

```
semaphore mutex = 1;  
semaphore wrt   = 1;  
int      readcount = 0;
```

■ Các writer process

```
wait(wrt);  
...  
writing is performed  
...  
signal(wrt);
```

■ Các reader process

```
wait(mutex);  
readcount++;  
if (readcount == 1)  
    wait(wrt);  
signal(mutex);  
...  
reading is performed  
...  
wait(mutex);  
readcount--;  
if (readcount == 0)  
    signal(wrt);  
signal(mutex);
```



Bài toán Readers-Writers (tt.)

- **mutex**: “bảo vệ” biến readcount
- **wrt**
 - Bảo đảm mutual exclusion đối với các writer
 - Được sử dụng bởi reader đầu tiên hoặc cuối cùng vào hay ra khỏi vùng tranh chấp.
- Nếu một writer đang ở trong CS và có n reader đang đợi thì một reader được xếp trong hàng đợi của wrt và $n - 1$ reader kia trong hàng đợi của mutex
- Khi writer thực thi signal(wrt), hệ thống có thể phục hồi thực thi của một trong các reader đang đợi hoặc writer đang đợi.



Các vấn đề với semaphore

- Nếu các tác vụ wait(S) và signal(S) nằm rải rác ở rất nhiều process \Rightarrow Người lập trình khó nắm bắt được hiệu ứng của chúng.
- Nếu không sử dụng đúng \Rightarrow có thể xảy ra deadlock hoặc starvation.
- Một process bị “die” có thể kéo theo các process khác cùng sử dụng biến semaphore.

```
signal(mutex)
...
critical section
...
wait(mutex)
```

```
wait(mutex)
...
critical section
...
wait(mutex)
```

```
signal(mutex)
...
critical section
...
signal(mutex)
```



Monitor

- Phần tử ngôn ngữ cấp cao
- Xuất hiện trong nhiều ngôn ngữ lập trình đồng thời như
 - Concurrent Pascal, Modula-3, Java,...
- Có thể hiện thực bằng semaphore

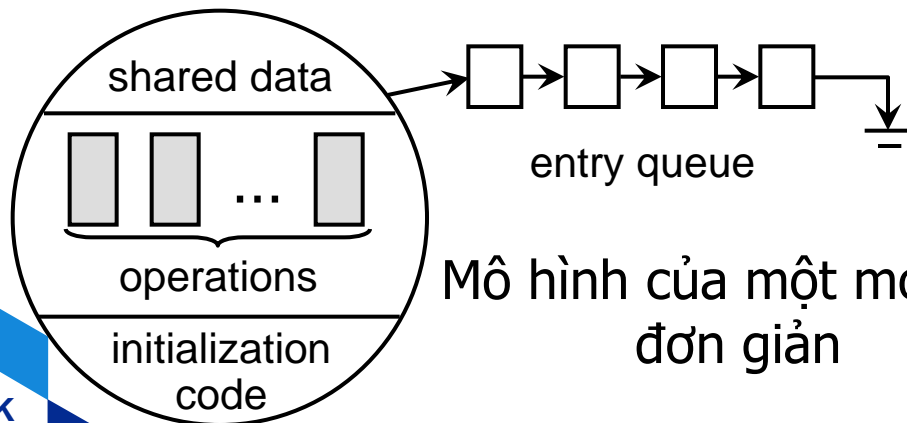
Monitor (tt.)

- Kiểu module phần mềm, bao gồm

- Một hoặc nhiều *thủ tục* (procedure)
- Một đoạn *code khởi tạo* (initialization code)
- Các *biến dữ liệu cục bộ* (local data variable)

- Ngữ nghĩa của monitor

- Shared variable chỉ có thể truy xuất bởi các thủ tục của monitor
- Process “vào monitor” bằng cách gọi một trong các thủ tục của monitor
- Các thủ tục của monitor loại trừ tương hỗ



Mô hình của một monitor đơn giản



Cấu trúc của monitor

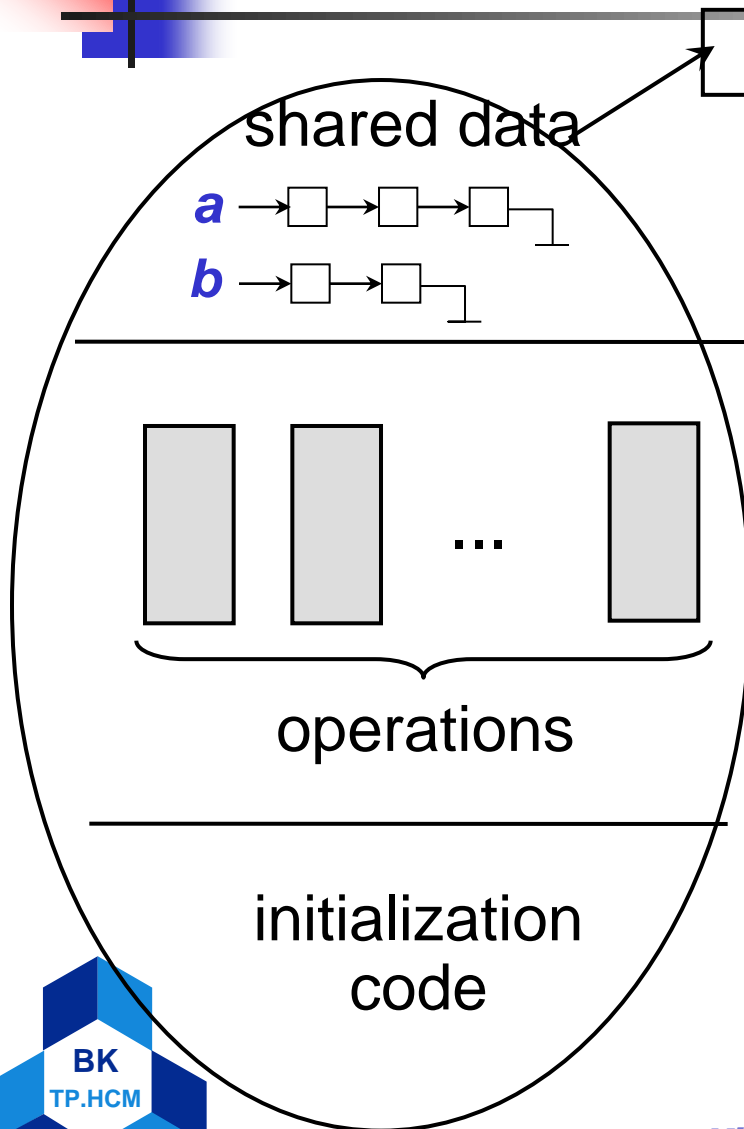
```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        . . .
    }
    procedure body P2 (...) {
        . . .
    }
    procedure body Pn (...) {
        . . .
    }
    {
        initialization code
    }
}
```



Condition variable

- Nhằm cho phép một process đợi “trong monitor”, phải khai báo *biến điều kiện* (condition variable)
condition a, b;
- Các biến điều kiện đều cục bộ và chỉ được truy cập bên trong monitor.
- Chỉ có thể thao tác lên biến điều kiện bằng hai thủ tục:
 - a.**wait**: process gọi tác vụ này sẽ bị “block trên biến điều kiện” a
 - process này chỉ có thể tiếp tục thực thi khi có process khác thực hiện tác vụ a.**signal**
 - a.**signal**: phục hồi quá trình thực thi của process bị block trên biến điều kiện a.
 - Nếu có nhiều process: chỉ chọn một
 - Nếu không có process: không có tác dụng

Monitor có condition variable



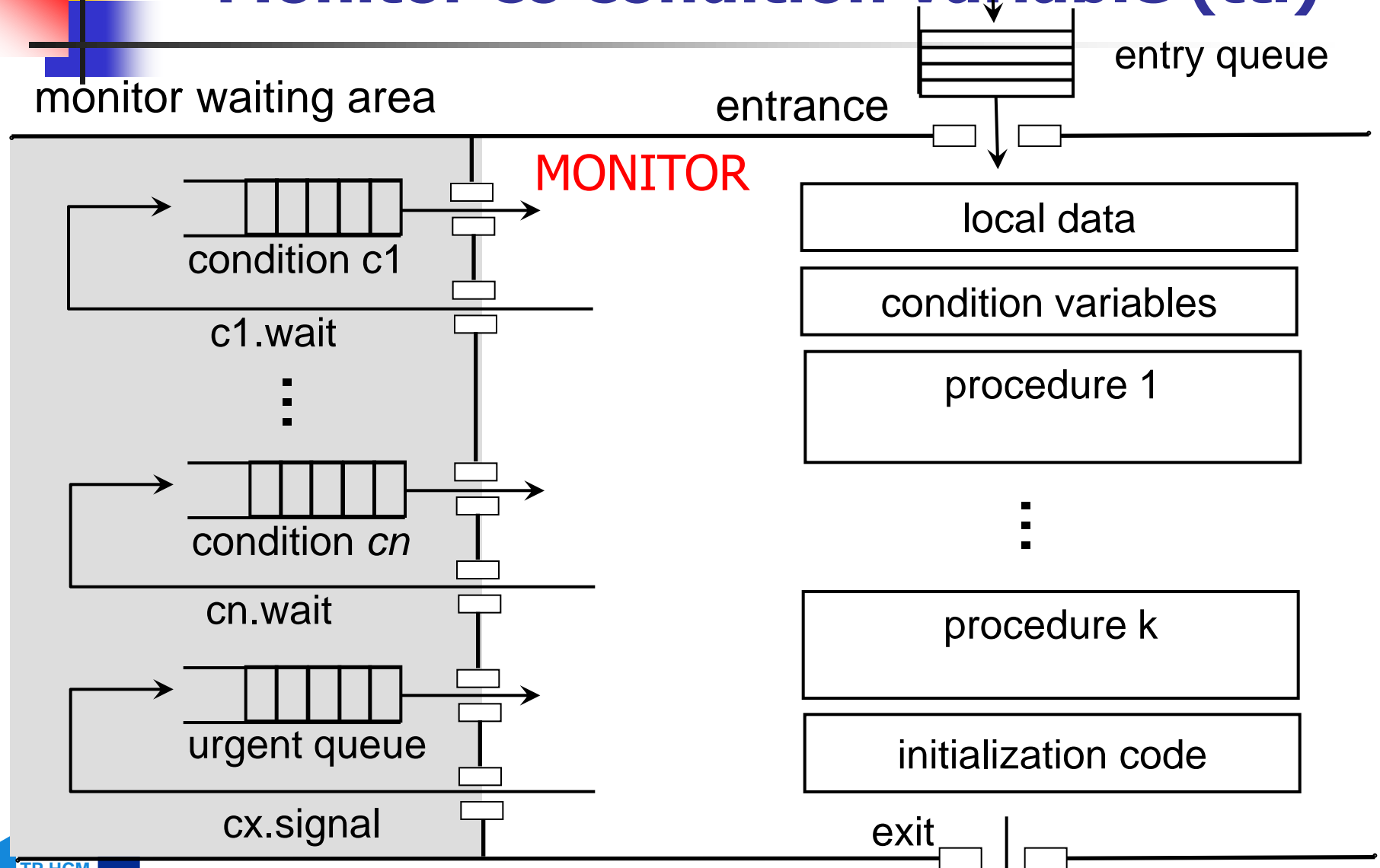
↪ Các process có thể đợi ở *entry queue* hoặc đợi ở các *condition queue* (*a*, *b*,...)

↪ Khi thực hiện lệnh *a.wait*, process sẽ được chuyển vào condition queue *a*

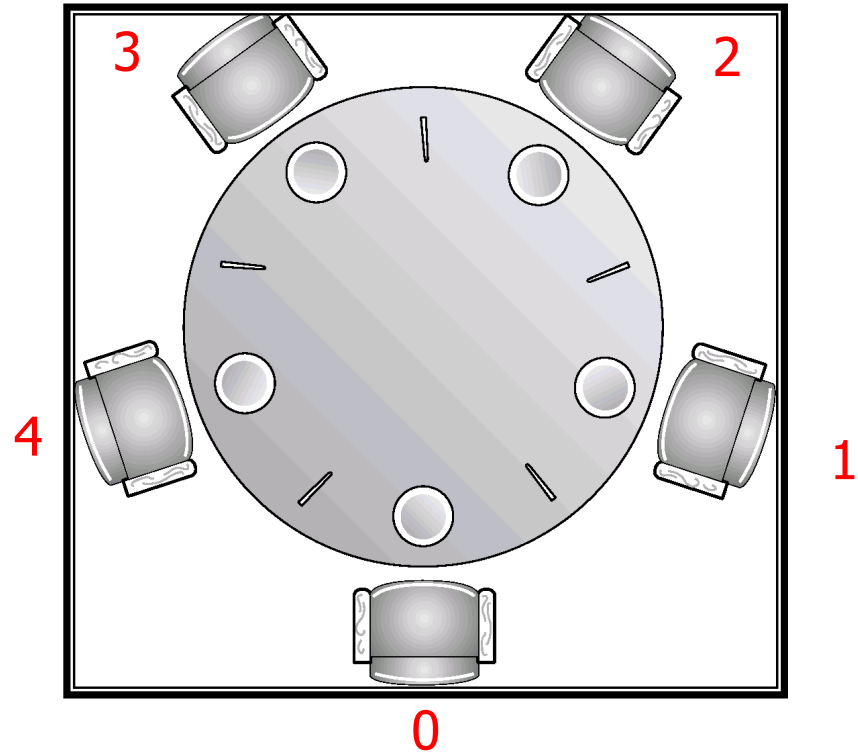
↪ Lệnh *a.signal* chuyển một process từ condition queue *a* vào monitor

- Khi đó, để bảo đảm mutual exclusion, process gọi *a.signal* sẽ bị blocked và được đưa vào *urgent queue*

Monitor có condition variable (tt.)



Monitor và dining philosophers



```
monitor dp {  
    enum {THINKING, HUNGRY, EATING} state[5];  
    condition self[5];  
}
```




Monitor và dining philosophers (tt.)

```
void pickup(int i) {  
    state[ i ] = HUNGRY;  
    test( i );  
    if (state[ i ] != EATING)  
        self[ i ].wait();  
}  
void putdown(int i) {  
    state[ i ] = THINKING; // test left and right neighbors  
    test((i + 4) % 5);      // left neighbor  
    test((i + 1) % 5);      // right ...  
}
```



Monitor và dining philosophers (tt.)

```
void test(int i) {  
    if ( (state[(i + 4) % 5] != EATING) &&  
        (state[ i ] == HUNGRY) &&  
        (state[(i + 1) % 5] != EATING) ) {  
        state[ i ] = EATING;  
        self[ i ].signal();  
    }  
}  
  
void init() {  
    for (int i = 0; i < 5; i++)  
        state[ i ] = THINKING;  
}
```



Monitor và dining philosophers (tt.)

- Trước khi ăn, mỗi triết gia phải gọi hàm pickup(), ăn xong rồi thì phải gọi hàm putdown()

đói
dp.pickup(i);
ăn
dp.putdown(i);
suy nghĩ

- Giải thuật
 - không gây deadlock nhưng có thể gây starvation.
 - không thực sự phân bố vì điều khiển tập trung.