Chương 4
ĐỒNG BỘ TIẾN TRÌNH

Đồng bộ hóa tiến trình

- Nền tảng
- Vấn đề “Đoạn tới hạn”
- Giải pháp của Peterson
- Đồng bộ nhờ phần cứng
- Semaphores
- 3 bài toán đồng bộ điển hình
- Monitors

Nền tảng

- Truy nhập đồng thời đến dữ liệu chia sẻ có thể gây ra sự không nhất quán về dữ liệu

  → Cần các kĩ thuật để việc thực thi tuần tự của các tiến trình cộng tác

- Vấn đề cộng tác tiến trình (phần1)
  - Bài toán "Sản xuất – Tiêu dùng"
  - Bộ nhớ chia sẻ (có giới hạn)
  - Biến count lưu số các items trong buffer

Mã nguồn cho Producer và Consumer

- Producer

```
while (true)
{
//produce an item
while (count == BUFFER_SIZE);
// do nothing
buffer [in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
count++;
}
```

- Consumer

```
while (1)
{
while (count == 0)
; // do nothing
nextConsumed= buffer[out];
out = (out + 1) %
BUFFER_SIZE;
count--;
/*consume the item in
}
```

Chạy đua...

- count++ có thể được thực hiện như sau
  - register1 = count
  - register1 = register1 + 1
  - count = register1
- count - - có thể được thực hiện như sau
  - register2 = count
  - register2 = register2 - 1
  - count = register2

Giả sử ban đầu “count = 5”:

S0: producer thực thi
register1 = count {register1 = 5}
S1: producer thực thi
register1 = register1+ 1 {register1 = 6}
S2: consumer thực thi
register2 = count {register2 = 5}
S3: consumer thực thi
register2 = register2-1 {register2 = 4}
S4: producer thực thi
count = register1 {count = 6 }
S5: consumer thực thi
count = register2 {count = 4}

...Chạy đua

- Chạy đua: Tình huống nhiều tiến trình cùng truy cập và xử lý dữ liệu dùng chung đồng thời. Giá trị cuối cùng của dữ liệu phụ thuộc vào tiến trình sẽ kết thúc cuối.
- Để tránh việc chạy đua, các tiến trình đồng thời cần phải được đồng bộ hóa

Vấn đề đoạn tới hạn (Critical section)

- *n* tiến trình cùng muốn truy cập đến dữ liệu chia sẻ
- Mỗi tiến trình có một đoạn mã gọi là "đoạn tới hạn" (miền găng), trong "đoạn tới hạn" dữ liệu chia sẻ mới được truy cập tới.
- Vấn đề - đảm bảo rằng khi một tiến trình đang trong đoạn tới hạn của nó, các tiến trình khác không được phép thực thi đoạn tới hạn của chúng.

Giải pháp cho vấn đề "đoạn tới hạn"

1. Loại trừ lẫn nhau (độc quyền truy xuất). Nếu tiến trình P_i đang trong đoạn tới hạn, không tiến trình nào khác được phép ở trong đoạn tới hạn.

2. Phát triển. Nếu không tiến trình nào ở trong đoạn tới hạn và có một số tiến trình muốn vào đoạn tới hạn, việc lựa chọn một tiến trình vào đoạn tới hạn sẽ không bị trì hoãn mãi.

3. Chờ đợi có cận. Phải có một cận về số lần mà các tiến trình khác được phép vào đoạn tới hạn sau khi một tiến trình đã yêu cầu vào đoạn tới hạn và trước khi yêu cầu đó được đáp ứng.

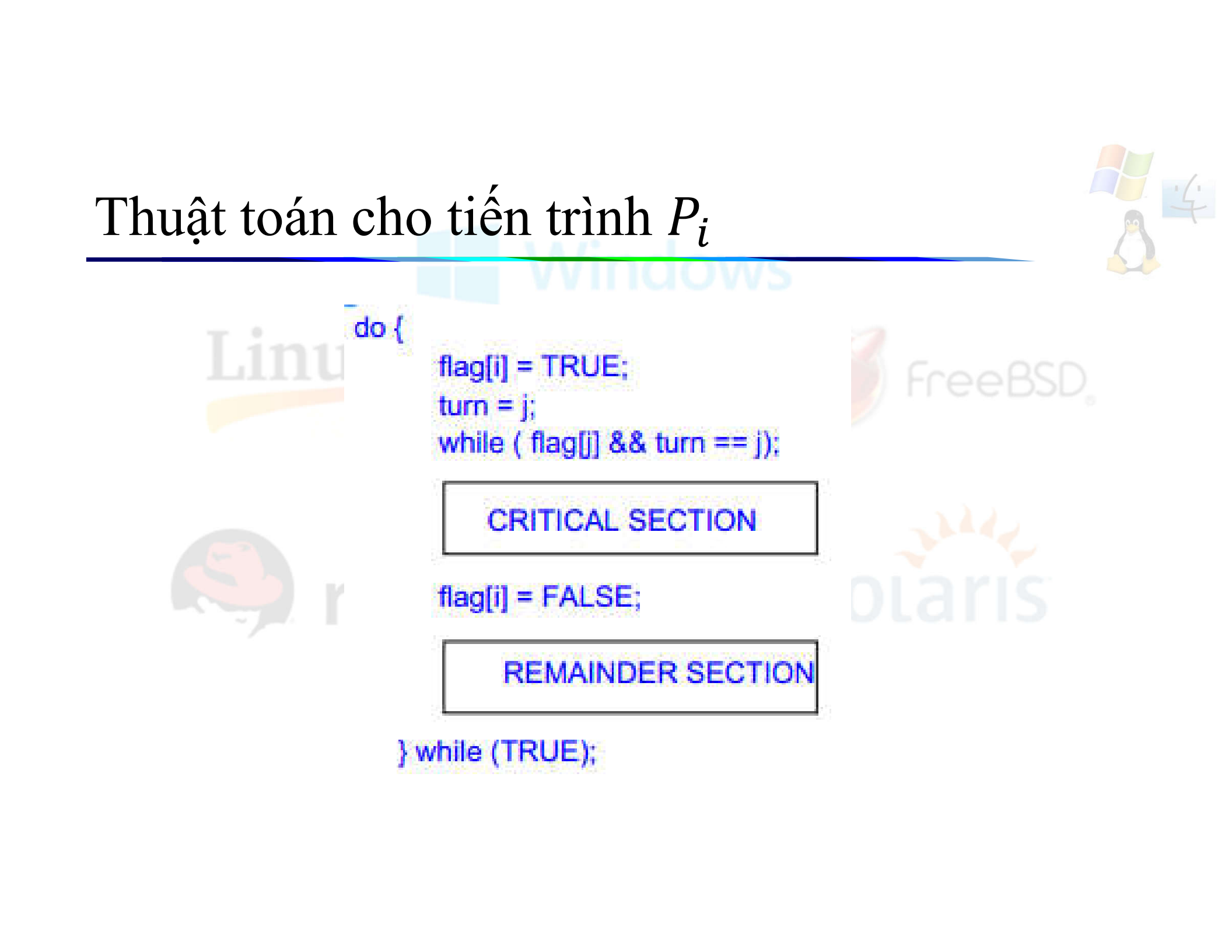
Giả thiết rằng mỗi tiến trình thực thi với tốc độ khác không

Không có giả thiết về mối tương quan tốc độ của n tiến trình

Giải pháp của Peterson

- Giải pháp đồng bộ cho hai tiến trình
- Giả sử hai lệnh LOAD và STORE là các lệnh nguyên tử (thao tác không thể phân chia)
- Các tiến trình chia sẻ các biến sau:
  - int turn;
  - Boolean flag[2];
- Biến turn cho biết tiến trình nào được vào đoạn tới hạn.
- Mảng flag chỉ xem liệu một tiến trình có sẵn sàng vào đoạn tới hạn hay không.
  - flag[i] = true là tiến trình P_i sẵn sàng vào đoạn tới hạn

Thuật toán cho tiến trình P_i



```
do {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

            CRITICAL SECTION

        flag[i] = FALSE;

            REMAINDER SECTION

} while (TRUE);
```

Đồng bộ hóa nhờ phần cứng

- Nhiều hệ thống cung cấp sự hỗ trợ của phần cứng cho vấn đề đoạn tới hạn
- Các hệ đơn xử lý – có thể bỏ chức năng ngắt
  - Mã được thực thi mà không có sự chiếm đoạt
  - Thông thường không hiệu quả trên các hệ thống đa xử lý
    - Các hệ điều hành với chiến lược này thường không khả cỡ
- Các máy tính hiện đại cung cấp các lệnh phần cứng nguyên tử
    - Nguyên tử = không phân chia
  - Kiểm tra từ nhớ và thiết lập giá trị
  - Tráo đổi nội dung của hai từ nhớ

Lệnh TestAndSet

- Định nghĩa:

```
Boolean TestAndSet (Boolean *target)
{
Boolean rv = *target;
*target = TRUE;
return rv;
}
```

Giải pháp dùng TestAndSet

- Biến chia sẻ lock, được khởi tạo bằng FALSE
- Giải pháp:

```
do {
while ( TestAndSet (&lock )) ; /* do nothing
    // critical section
lock = FALSE;
    // remainder section
} while ( TRUE);
```

Lệnh Swap

- Định nghĩa:

```
void Swap (Boolean *a, boolean *b)
{
Boolean temp = *a;
*a = *b;
*b = temp:
}
```

Giải pháp dùng Swap

- Biến chia sẻ lock, được khởi tạo bằng FALSE; mỗi tiến trình có một biến Boolean cục bộ
- Giải pháp:

```
do {
    key = TRUE;
        while ( key == TRUE)
            Swap (&lock, &key );
                // critical section
            lock = FALSE;
                // remainder section
} while ( TRUE);
```

Semaphore

- Công cụ đồng bộ không đòi hỏi “chờ - bận”
- Semaphore S – biến nguyên
- Hai thao tác “nguyên tử” để sửa đổi S:

```
wait (S) {
    while S <= 0;
        // no-op
    S--;
}
```

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

Semaphore –Công cụ đồng bộ tổng quát

- Hai loại semaphore
  - Semaphore đếm
    - Giá trị của semaphore là số các tài nguyên available
  - Semaphore nhị phân
    - Semaphore chỉ nhận hai giá trị 0 và 1
    - Còn được gọi là mutex lock
- Loại trừ lẫn nhau dùng semaphore

```
Semaphore S; // initialized to 1
        wait (S);
                Critical Section
        signal (S);
                remainder section
```

Cài đặt Semaphore

- Cần đảm bảo rằng không có hai tiến trình nào có thể cùng thực thi wait() và signal() trên cùng 1 semaphore tại cùng 1 thời điểm
- Thực thi semaphore bản thân nó cũng là bài toán đoạn tới hạn
  - Mã cho wait() và signal() được đặt trong đoạn tới hạn
  - Có thể cài đặt cơ chế "chờ - bận" hoặc không
- Chờ - bận
  - Mã nguồn đơn giản
  - Chấp nhận được trong TH đoạn tới hạn ít khi được truy nhập tới

Cài đặt Semaphore không chờ-bận

- Mỗi semaphore liên kết với một hàng đợi.
- Mỗi phần tử của hàng đợi có hai phần
  - Giá trị (kiểu nguyên)
  - Con trỏ đến bản ghi tiếp theo trong danh sách
- Hai thao tác:
  - Block –đặt tiến trình gọi thao tác này vào hàng đợi semaphore.
  - Wakeup – gỡ bỏ một trong số các tiến trình trong hàng đợi và đặt nó vào hàng đợi sẵn sàng

Cài đặt Semaphore không chờ - bận

- Implementation of wait:

```
Wait (S)
{
        value--;
        if (value < 0) {
        //add this process to waiting
        //queue
        block();
}
```

- Implementation of signal:

```
Signal (S)
{
    value++;
    if (value <= 0) {
            //remove a process P from
            //the waiting queue
    wakeup(P);
}
```

Bế tắc và Chết đói

- Bế tắc– hai hay nhiều tiến trình chờ vô hạn trên một sự kiện gây ra bởi một trong các tiến trình đang chờ
- S và Q là hai semaphores được khởi tạo bằng 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Chết đói –bị block vô hạn. Một tiến trình có thể bị gỡ khỏi hàng đợi của semaphore.

Các bài toán đồng bộ kinh điển

- Bộ đệm giới hạn
- Bài toán đọc và ghi
- Bài toán “bữa ăn của các triết gia”

Bài toán "Bộ đệm giới hạn"

- N bộ đệm, mỗi bộ đệm có item
- Semaphore mutex được khởi tạo bằng 1
- Semaphore full được khởi tạo bằng 0
- Semaphore empty được khởi tạo bằng N.

Bài toán "Bộ đệm giới hạn"

- Tiến trình "producer"

```
do {
        // produce an item
    wait (empty);
    wait (mutex);
        // add the item to the buffer
    signal (mutex);
    signal (full);
} while (true);
```

- Tiến trình "consumer"

```
do {
    wait (full);
    wait (mutex);
        // remove an item from buffer
    signal (mutex);
    signal (empty);
        // consume the removed item
} while (true);
```

Bài toán "Đọc - ghi"

- Một tập dữ liệu được chia sẻ giữa một số tiến trình đồng thời
  - Reader – chỉ đọc dữ liệu; không thực hiện bất kì một thao tác cập nhật.
  - Writer – có thể vừa đọc vừa ghi.
- Vấn đề
  - Cho phép nhiều reader đọc cùng một thời điểm. Chỉ có một writer có thể truy nhập đến dữ liệu chia sẻ tại cùng một thời điểm.
- Dữ liệu chia sẻ
  - Tập dữ liệu
  - Semaphore mutex được khởi tạo bằng 1.
  - Semaphore wrt được khởi tạo bằng 1.
  - Số nguyên readcount được khởi tạo bằng 0.

Bài toán "Đọc - ghi"

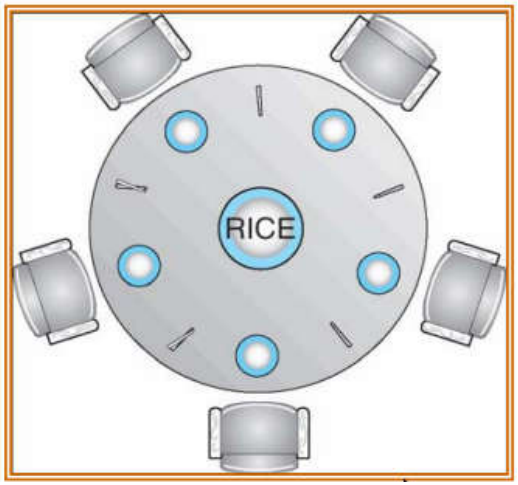
- Tiến trình ghi (writer)

```
do {
        wait (wrt) ;
            // writing is performed
        signal (wrt) ;
    } while (true)
```

- Tiến trình đọc (reader)

```
do {
        wait (mutex) ;
        readcount++ ;
        if (readercount == 1) wait (wrt) ;
        signal (mutex)
            // reading is performed
        wait (mutex) ;
        readcount --;
        if (redacount == 0) signal (wrt) ;
        signal (mutex) ;
    } while (true)
```

Bài toán “Bữa ăn của các triết gia”...



- Dữ liệu chia sẻ
- Cơm (tập dữ liệu)
- Semaphore chopstick [5] được khởi tạo bằng 1

... Bài toán "Bữa ăn của các triết gia"

- Mã lệnh cho Triết gia thứ I

```
Do {
    wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5] );
    // eat
    signal ( chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
    // think
} while (true) ;
```

Vấn đề của Semaphores

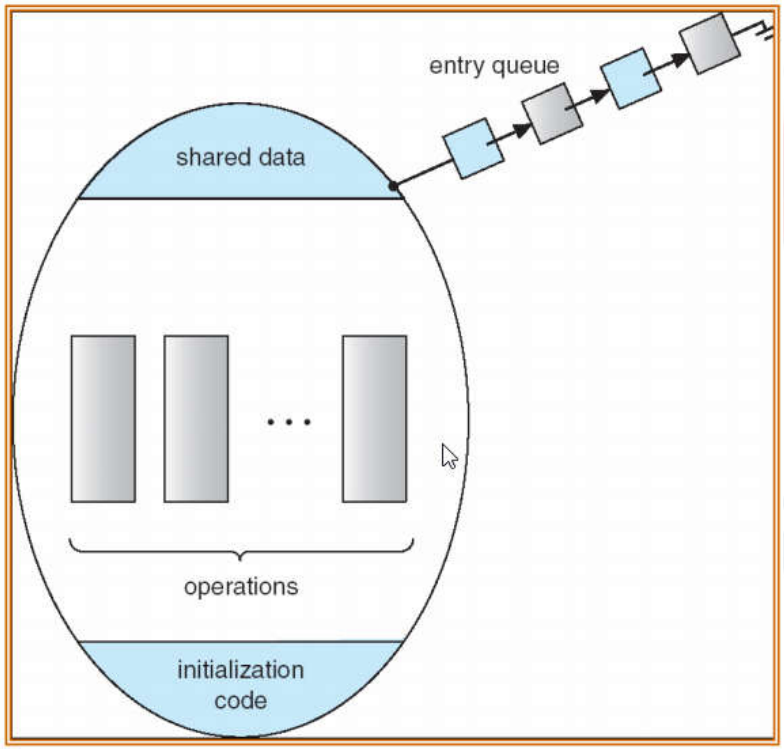
- Không sử dụng đúng các thao tác trên semaphore
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Bỏ qua wait (mutex) hay signal (mutex) (hoặc cả hai)

Monitors...

- Trừu tượng mức cao cung cấp phương thức hữu hiệu cho đồng bộ tiến trình
- Chỉ có một tiến trình chỉ có thể active trong monitor tại một thời gian

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
    …
    procedure Pn(…) {……}
    Initialization code ( ….) { …}
    …
}
```

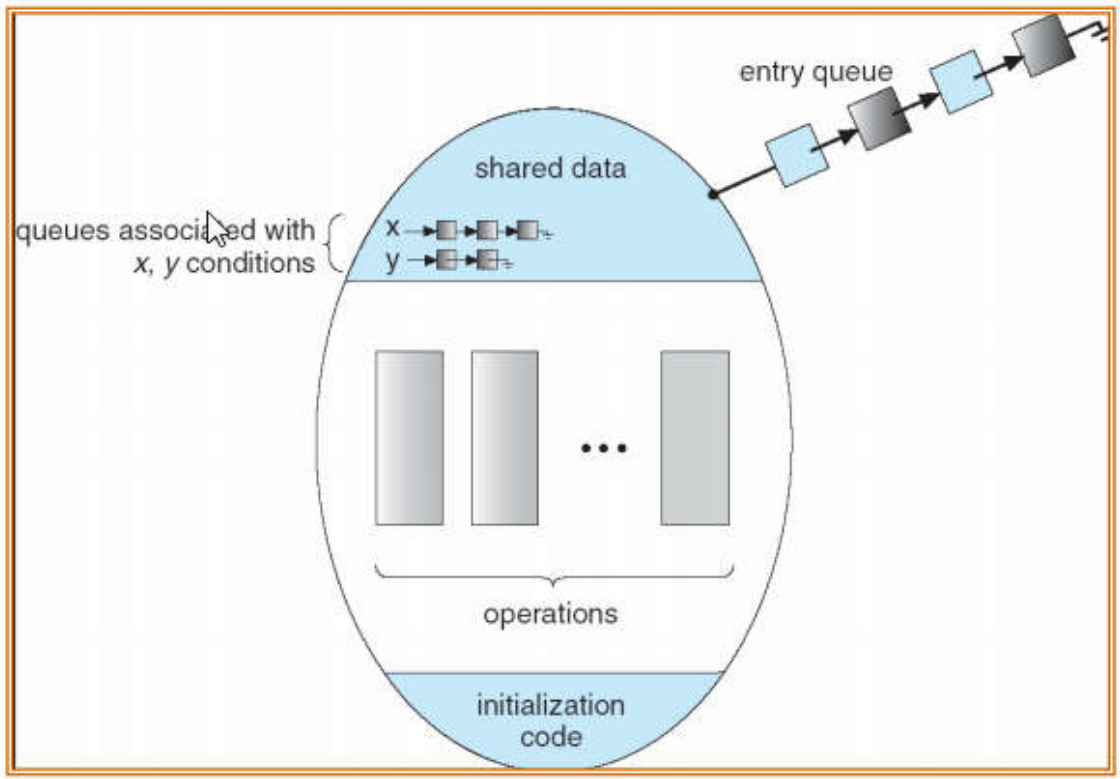
... Monitor



Các biến điều kiện

- condition x, y;
- Hai thao tác trên biến điều kiện:
  - x.wait () – tiến trình gọi thao tác này sẽ bị block.
  - x.signal () – khôi phục thực thi của một trong số các tiến trình (nếucó) đã gọi thao tác x.wait ()

Monitor với các biến điều kiện



Giải pháp cho bài toán "Bữa ăn của các triết gia"

```
monitor DP
{
enum{ THINKING; HUNGRY,
EATING) state [5] ;
condition self [5];
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);
test((i+ 1) % 5);
}
```

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