# Chương 3
# Đồng Bộ Quá Trình

Nguyễn Quang Hùng

E-mail: hungnq2@cse.hcmut.edu.vn

Tel: +84 8 8647256 (Ext. 5840)

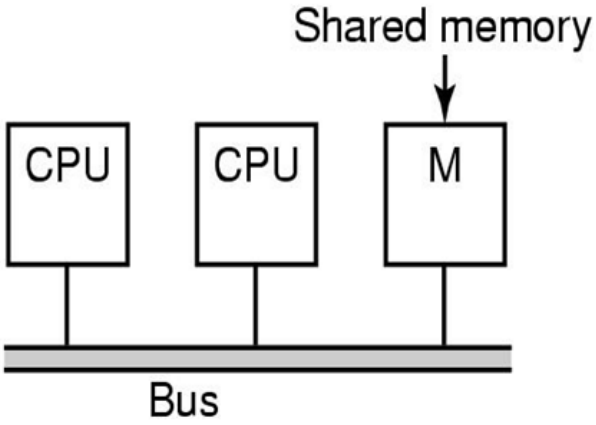
# Nội dung

- Khái niệm cơ bản
- Critical section
- Các giải pháp dùng lệnh máy thông thường
  - Giải thuật Peterson, và giải thuật bakery
- Các giải pháp dùng lệnh cấm ngắt hoặc lệnh máy đặc biệt
- Semaphore
- Semaphore và các bài toán đồng bộ
- Monitor

# Bài toán đồng bộ (1/2)

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



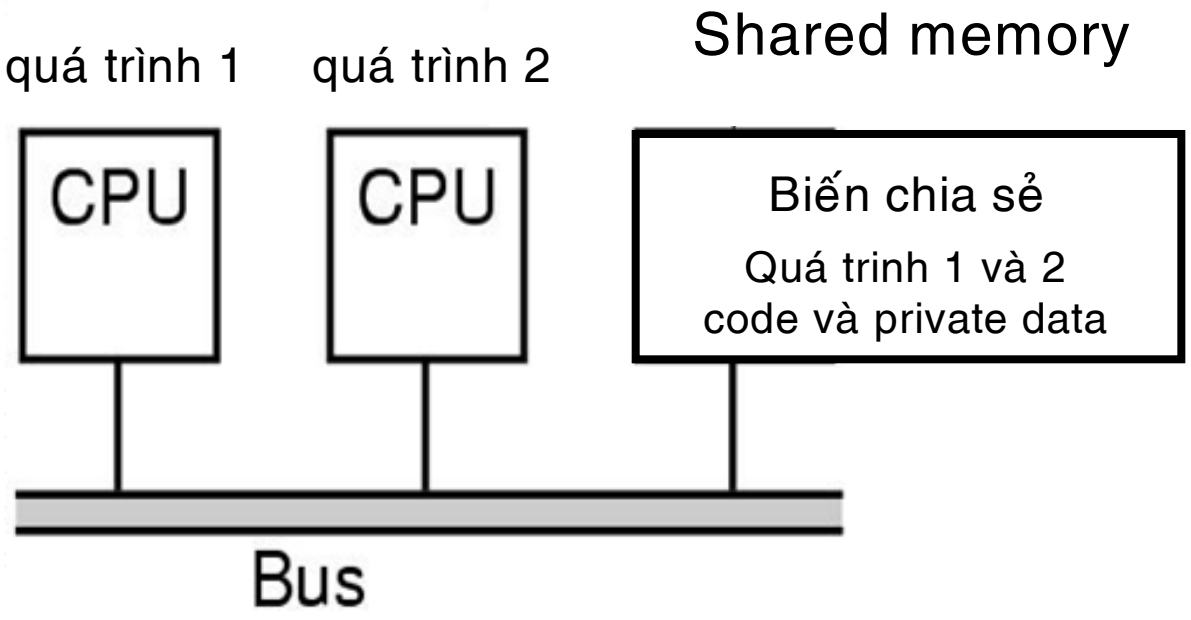

- Nếu không có sự kiểm soát khi truy cập các dữ liệu chia sẻ thì chúng có thể trở nên *không nhất quán*.
- Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế bảo đảm sự thực thi có trật tự của các process đồng thời.

# Bài toán đồng bộ (2/2)

- Hai lớp bài toán đồng bộ:
  - Hợp tác
    - Bài toán producer-consumer: bounded buffer
  - Cấp phát tài nguyên
    - Bài toán loại trừ tương hỗ: đồng bộ nhiều quá trình sử dụng một tài nguyên không chia sẻ đồng thời được
    - Bài toán Dining Philosophers

# Đồng thời ⊃ 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 (1/3)

- Ví dụ Bounded buffer, thêm biến đếm count

```
#define BUFFER_SIZE 8     /* 8 buffers */
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0, count = 0;
```

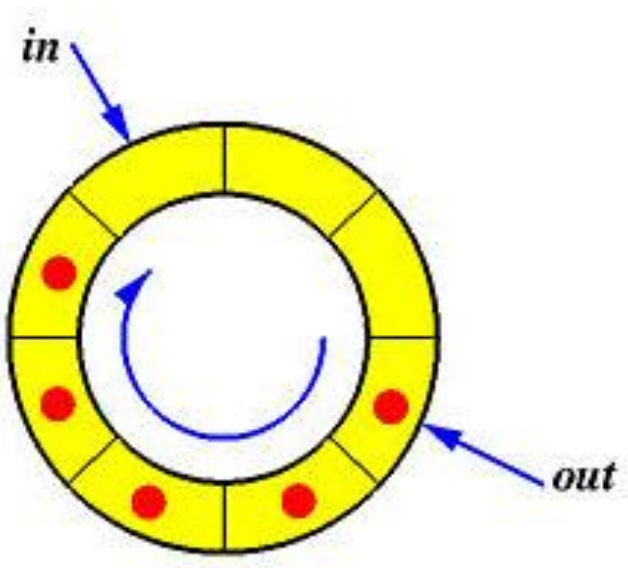
# Bài toán Producer-consumer (2/3)

- Quá trình Producer

```
item nextProduced;
while(1) {
    while (count == BUFFER_SIZE);
    buffer[in] = nextProduced;
    count++;
    in = (in + 1) % BUFFER_SIZE;
}
```

- Quá trình Consumer

```
item nextConsumed;
while(1) {
    while (count == 0);
    nextConsumed = buffer[out];
    count--;
    out = (out + 1) % BUFFER_SIZE;}
```




biến count được chia sẻ giữa producer và consumer

# Bài toán Producer-consumer (3/3)

- 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.
- Giải quyết vấn đề race condition cho 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
- 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

**Vấn đề**

- Thiết kế entry section và exit section

# Định nghĩa lời giải của bài toán loại trừ tươnghỗ

*Lời giải* phải thỏa ba tính chất

1. *Mutual exclusion*
2. *Progress*
   - (Progress cho entry section) Nếu ít nhất một process đang trong entry section và không có process nào đang trong critical section, thì một process vào critical section tại một thời điểm sau đó.
   - (Progress cho exit section) Nếu ít nhất một process đang trong exit section, thì một process vào remainder section tại một thời điểm sau đó.
3. *Starvation freedom* (*lockout-freedom*)
   - (cho entry section) quá trình vào entry section sẽ vào CS
   - (cho exit section) quá trình vào exit section sẽ vào remainder section.

# Phân loại giải pháp cho bài toán loại trừ tương hỗ

- Giải pháp dùng lệnh máy thông thường
  - Giải thuật Peterson cho 2 process
  - Giải thuật Bakery cho n process
- Giải pháp dùng lệnh cấm ngắt hay lệnh máy đặc biệt
  - Lệnh cấm ngắt (disabling interrupts)
  - Lệnh máy đặc biệt như:
    - testAndSet( lock )
    - swap ( &a, &b )

# Giải pháp dùng lệnh máy thông thường

- Giải pháp cho 2 process đồng thời
  - Dẫn nhập từ hai giải thuật 1 & giải thuật 2
  - Giải thuật Peterson cho 2 process
    - Là một lời giải trọn vẹn thỏa mãn tất cả điều kiện của bài toán tương hỗ cho 2 process
- Giải pháp cho n process
  - Giải thuật Bakery
    - Là một lời giải trọn vẹn thỏa mãn tất cả điều kiện của bài toán tương hỗ cho 2 process

# Giải thuật 1 (1/2)

- Biến chia sẻ

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

  nếu **turn =** i thì $P_i$ 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 (2/2)

(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 (Progress cho entry section):

Nếu turn = 0, P0 được vào CS và sau đó gán turn = 1 và vào remainder section (RS); giả sử P0 "ở lâu" trong đó.

Lúc đó P1 vào CS và sau đó gán turn = 0, kế 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

- Biến chia sẻ

  boolean flag[ 2 ]; /* khởi đầu flag[ 0 ] = flag[ 1 ] = false */

  Nếu flag[ i ] = true thì $P_i$ "sẵn sàng" vào critical section.

- Process $P_i$

```
do {
    flag[ i ] = true;
    while ( flag[ j ] ); /* Pi "nhường" Pj */
    critical section
    flag[ i ] = false;
    remainder section
} while (1);
```

# Giải thuật 2 (tiếp)

(copy lại)

- Process $P_0$

```
do {
        flag[ 0 ] = true;
        while (flag[ 1 ]);
        critical section
        flag[ 0 ] = false;
        remainder section
} while (1);
```

- Process $P_1$

```
do {
        flag[ 1 ] = true;
        while (flag[ 0 ]);
        critical section
        flag[ 1 ] = false;
        remainder section
} while (1);
```

- Bảo đảm được mutual exclusion. Chứng minh?
- Không thỏa mãn progress. Vì sao? Nếu đồng thời
  P0 gán flag[ 0 ] = true
  P1 gán flag[ 1 ] = true
  → P0 và P1 loop mãi mãi trong vòng lặp while

# Giải thuật Peterson cho 2 process (1/2)

- Giải thuật Peterson yêu cầu hai biến chia sẻ chung cho cả hai process là:

```
int turn;
boolean flags[2];
```

- Process $P_i$ , với i = 0 hay 1

```
do {
        flag[ i ] = TRUE;      /*  Process i sẵn sàng  */
        turn = j;              /*  Nhường process j    */
        while (flag[ j ] and turn == j );
        critical section
        flag[ i ] = FALSE;
        remainder section
} while (1);
```

# Giải thuật Peterson cho 2 process (2/2)

(viết lại)

Process $P_0$

```
do {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1] &&
           turn == 1);
    critical section
    flag[0] = FALSE;
    remainder section
} while(1);
```

Process $P_1$

```
do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] &&
           turn == 0);
    critical section
    flag[1] = FALSE;
    remainder section
} while(1);
```

# Giải thuật Peterson cho 2 process: Tính đúng đắn

- Mutual exclusion được bảo đảm
  - Chứng minh bằng phản chứng:

    Nếu $P_0$ và $P_1$ cùng ở trong CS thì flag[0] = flag[1] = true, suy ra từ điều kiện của vòng lặp while sẽ có turn = 0 (trong $P_0$) và turn = 1 (trong $P_1$). Điều không thể xảy ra.
- Chứng minh thỏa yêu cầu về progress và lock-out freedom
  - Xem tài liệu

# Giải thuật Bakery (1/3)

- *n* process
- 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 con số:
  - Nếu $i < j$ thì $P_i$ được vào trước.
- Khi ra khỏi CS, $P_i$ gán 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 nếu $a = c$ và $b < d$

  $\max(a_0,\ldots,a_k)$ là con số b sao cho $b \geq a_i$ với mọi $i = 0,\ldots, k$

# Giải thuật bakery (2/3)

```
/*  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);
```

# Giải thuật bakery (3/3)

- Giải thuật bakery bảo đảm
  - Mutual exclusion
  - Progress
  - Lockout freedom

# Nhận xét

- Các giải thuật vừa được trình bày chỉ 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.

# Dùng lệnh cấm ngắt

```
Process Pi:
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 (1/2)

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

```
boolean TestAndSet(boolean &target)
{
    boolean rv = target;
    target = true;    /* set */
    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);
```

Nếu TestAndSet không đơn nguyên, tình huống xấu nào có thể xảy ra cho giải thuật trên?

# Lệnh TestAndSet (2/2)

- Mutual exclusion được bảo đảm: nếu $P_i$ vào CS, các process $P_j$ khác đều đang busy waiting hoặc trong remainder section

- Khi $P_i$ ra khỏi CS, sự chọn lựa process $P_j$ vào CS kế tiếp là tùy ý $\Rightarrow$ *starvation*

# Lệnh Swap (1/2)

- Các processor (ví dụ Pentium) thông thường cung cấp một lệnh máy đơn nguyên là Swap(a, b) có tác dụng hoán chuyển trị của a và b.

  Swap(a, b) cũng có ưu nhược điểm như TestAndSet

# Lệnh Swap (2/2)

- Lệnh đơn nguyên

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

Áp dụng

- Biến chia sẻ

  **bool  lock = false;**

- 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 (1/2)

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 (2/2)

- 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 được thực thi lên lock mà trả về 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 cyclic order

# Semaphore

Semaphore là công cụ đồng bộ cung cấp bởi OS.

- *semaphore*, ngoài thao tác khởi động biến cùng với trị ban đầu, chỉ có thể được truy xuất qua hai tác vụ *đơn nguyên* (*atomic operations*)
  - 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 phục hồi 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 (1/3)

- 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, trạng thái running → waiting
- wakeup(P): phục hồi process P đang blocked, trạng thái waiting → ready

# Hiện thực semaphore (2/3)

- Các tác vụ semaphore được hiện thực như sau

```
void wait(semaphore S) { /* P(S) – giảm trị Semaphore S */
        S.value--;
        if (S.value < 0) {
                add this process to S.L;
                block();
        }
}

void signal(semaphore S) { /* V(S) – tăng trị Semaphore S */
        S.value++;
        if (S.value <= 0) {
                remove a process P from S.L;
                wakeup(P);
        }
}
```

# Hiện thực semaphore (3/3)

- 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
- Khi trị của S < 0, tác vụ signal(S) chọn một process từ hàng đợi của S và đưa nó vào hàng đợi 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 */
```

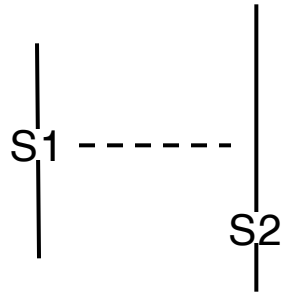
- Process Pi:

```
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 (1/2)

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

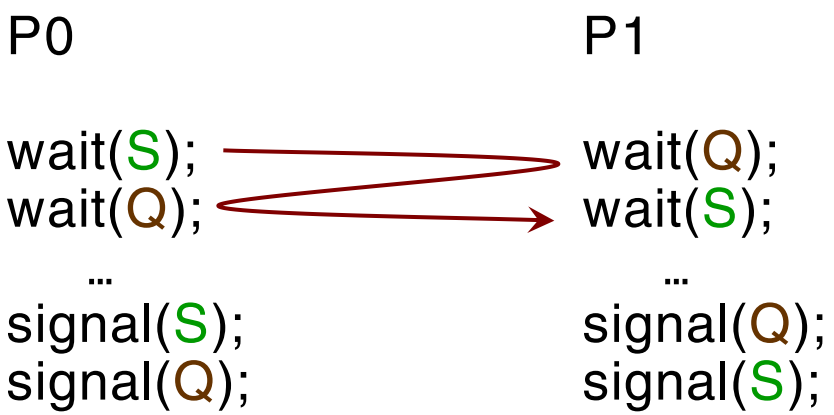
# Nhận xét về semaphore (2/2)

- Cấu trúc dữ liệu hiện thực semaphore là biến chia sẻ $\Rightarrow$ 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 (disabling interrupt). Nhưng phương pháp này không làm việc 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.
- Vd 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 (1/2)

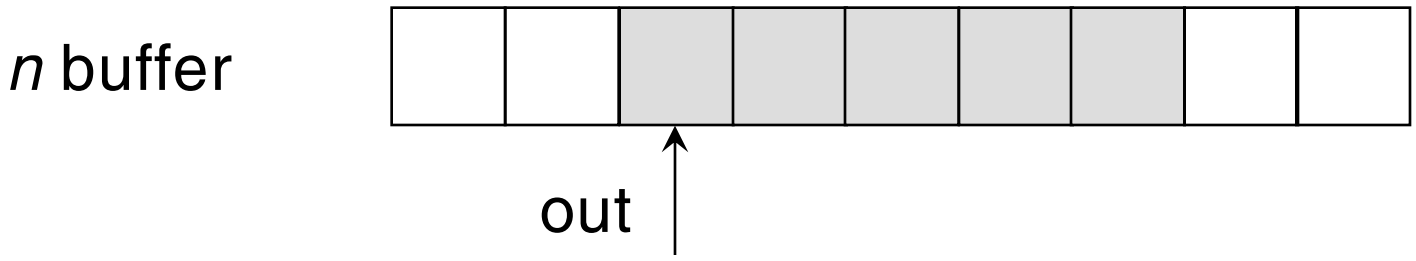
- Giải thuật cho bài toán bounded buffer
  - Dữ liệu chia sẻ:

    semaphore full, empty, mutex;
  - Khởi tạo trị:

    full = 0; /* số buffer đầy */

    empty = n; /* số buffer trống */

    mutex = 1;



# Semaphore và bài toán bounded buffer (2/2)

**producer**

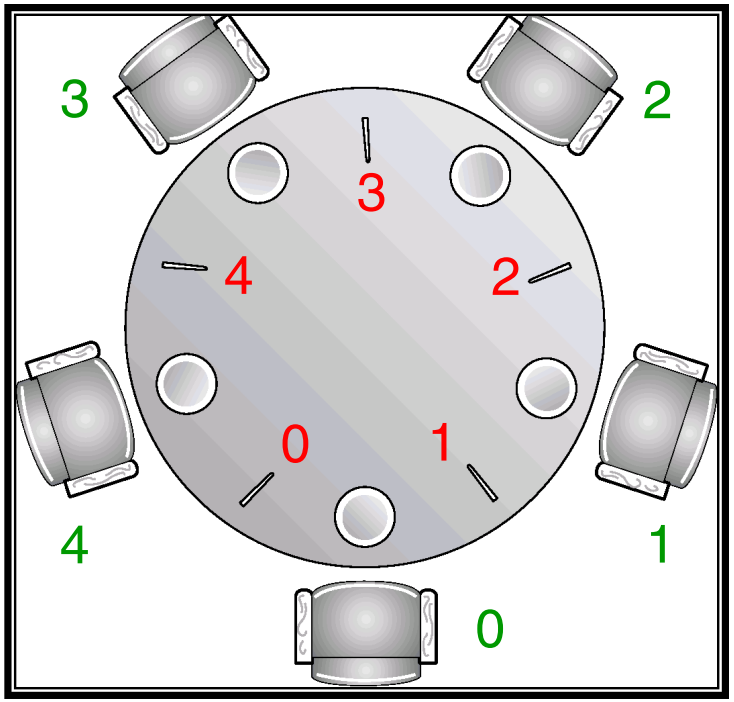
```
do {
        …
        nextp = new_item();
        …
        wait(empty);
        wait(mutex);
        …
        insert_to_buffer(nextp);
        …
        signal(mutex);
        signal(full);
} while (1);
```

**consumer**

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

# Bài toán "Dining Philosophers" (1/3)

- 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 cấp phát tài nguyên giữa các process sao cho không xảy ra deadlock và starvation



# Bài toán "Dining Philosophers" (2/3)

- Giải thuật

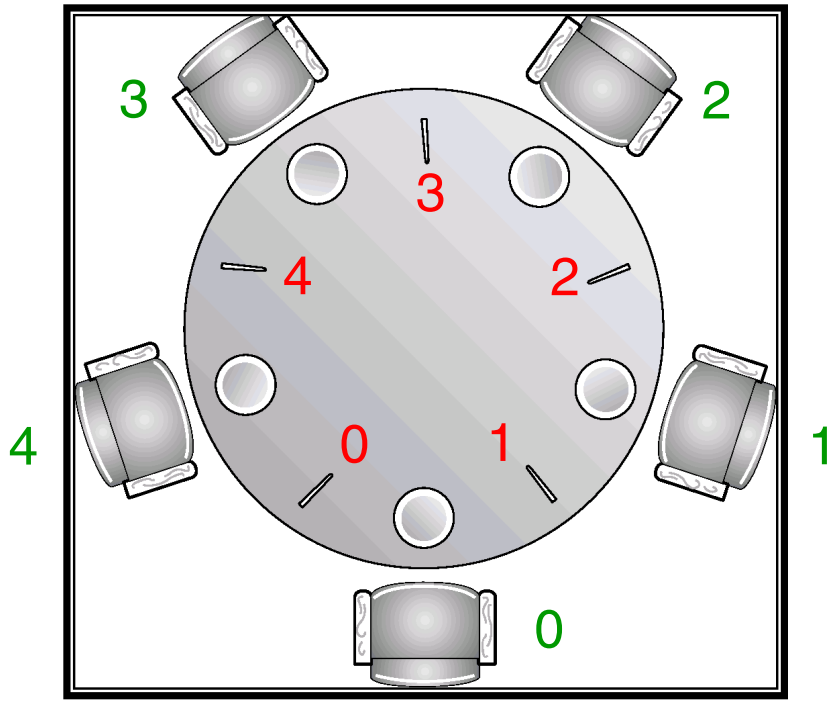
Dữ liệu chia sẻ:

```
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" (3/3)

- 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 ⇒ deadlock
- Một số giải pháp giải quyết được deadlock
  - Nhiều nhất 4 triết gia ngồi vào bàn
  - Triết gia cầm các đũa chỉ khi cả hai chiếc đũa đều sẵn sàng
  - 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 (1/4)

- Một file
- Nhiều reader và nhiều writer
- Bài toán Readers-Writers
  - Khi có 1 writer đang truy cập file, thì không có quá trình khác đồng thời truy cập file
  - Nhưng nhiều reader có thể cùng lúc truy cập file

# Bài toán Readers-Writers (2/4)

## 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 (3/4)

- 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.
- Nhận xét
  - 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 phục hồi một trong các reader hoặc writer đang đợi.

# Bài toán Readers-Writers (4/4)

- Câu hỏi: reader gọi wait(wrt) và reader gọi signal(wrt) có phải là một?

# Các vấn đề với semaphore

- Các tác vụ wait(S) và signal(S) nằm rải rác ở các process ⇒ Người lập trình khó nắm bắt được hiệu ứng của chúng.
- Nếu không sử dụng đúng ⇒ 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)
```

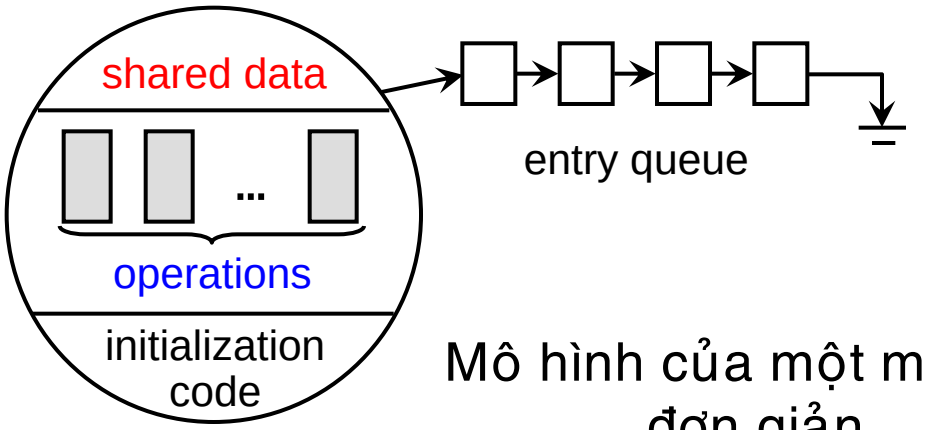
# Monitor (1/2)

- Construct 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 (2/2)

- 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
        }
}
```

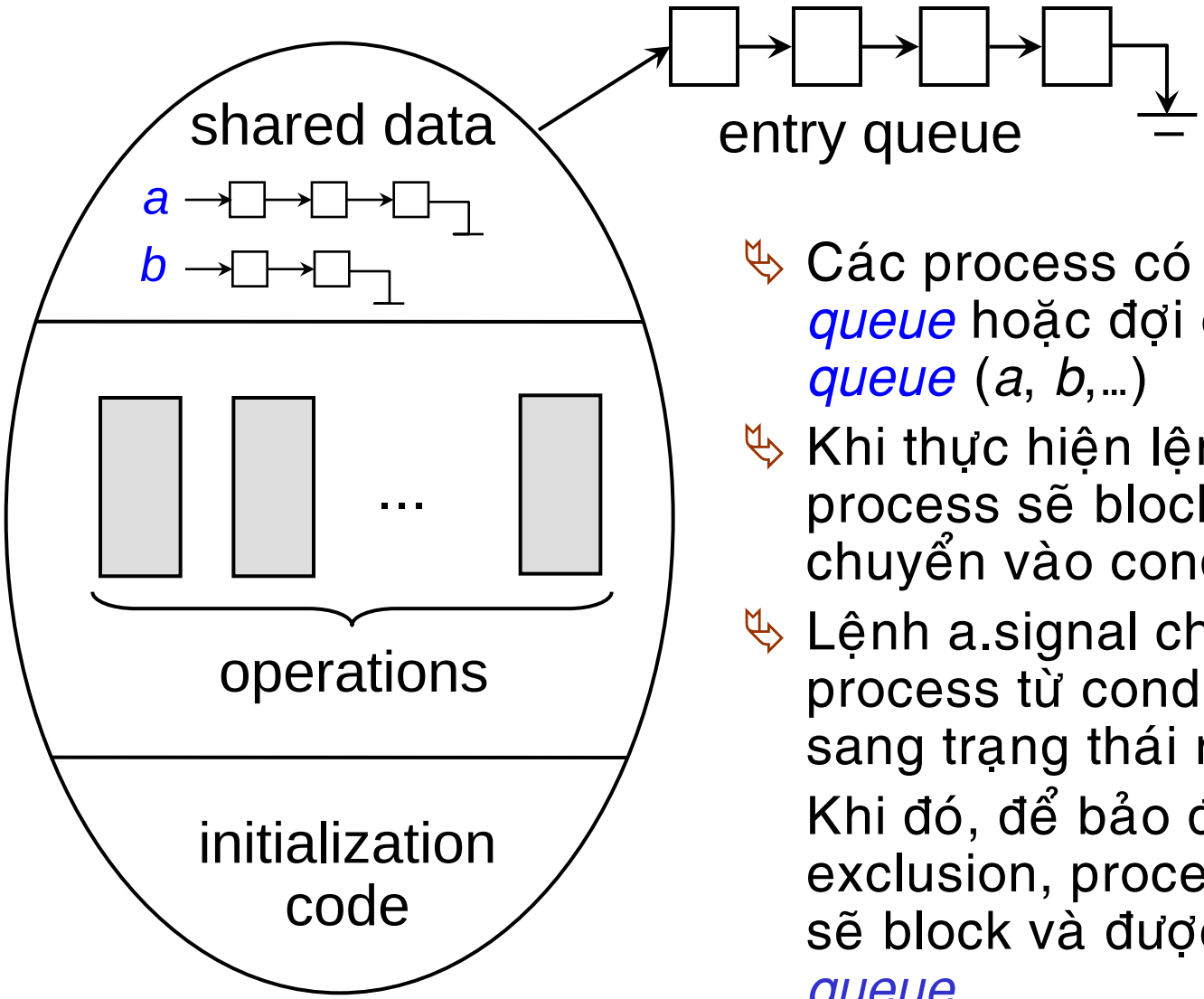
# Condition variable

- Để thực hiện concept "process đợi trong monitor", 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 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 process bị block trên biến điều kiện a.
    - Nếu có nhiều process: chỉ chọn một
    - Nếu không có process: không có tác dụng
  - Nhận xét: condition variable không phải là semaphore!

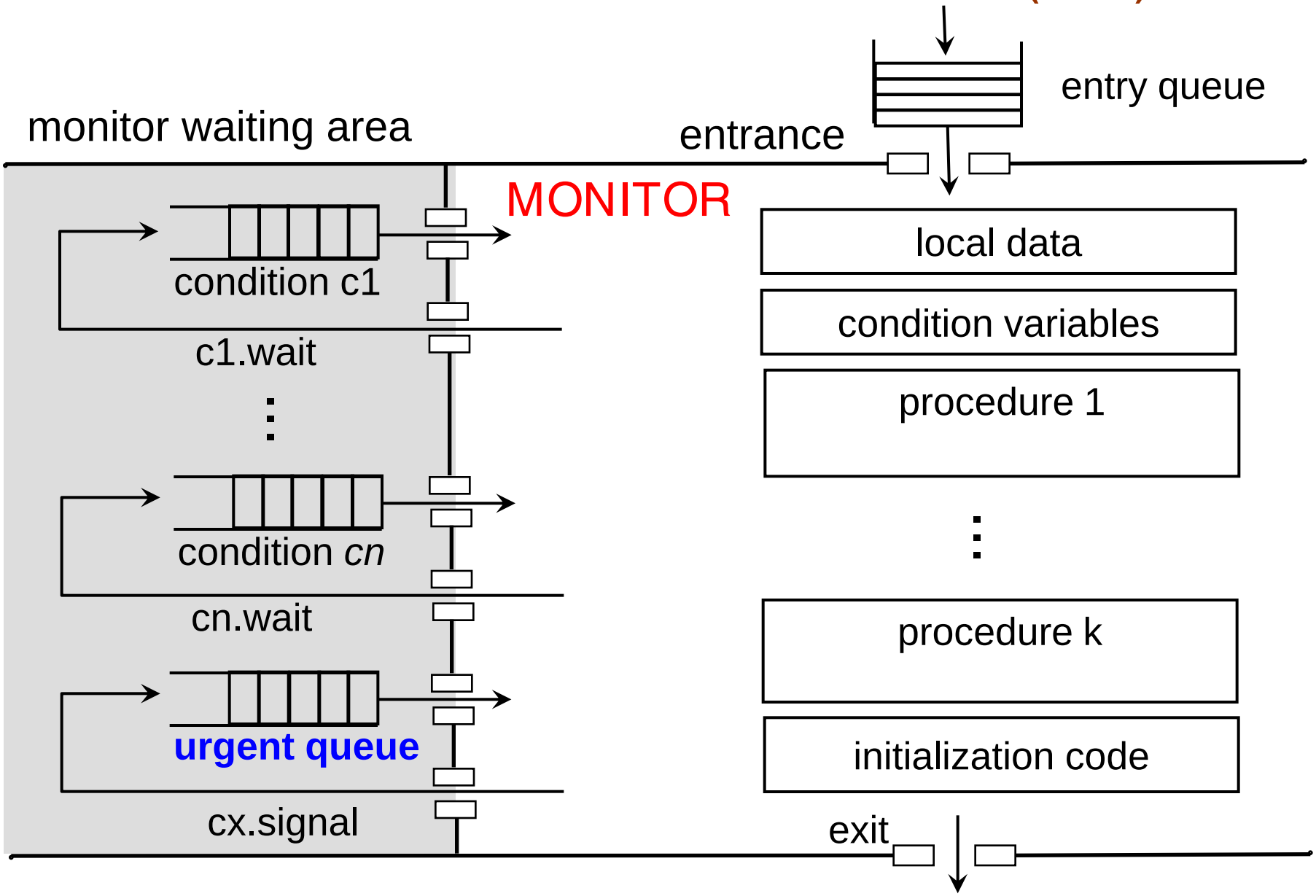
# Monitor có condition variable (1/2)



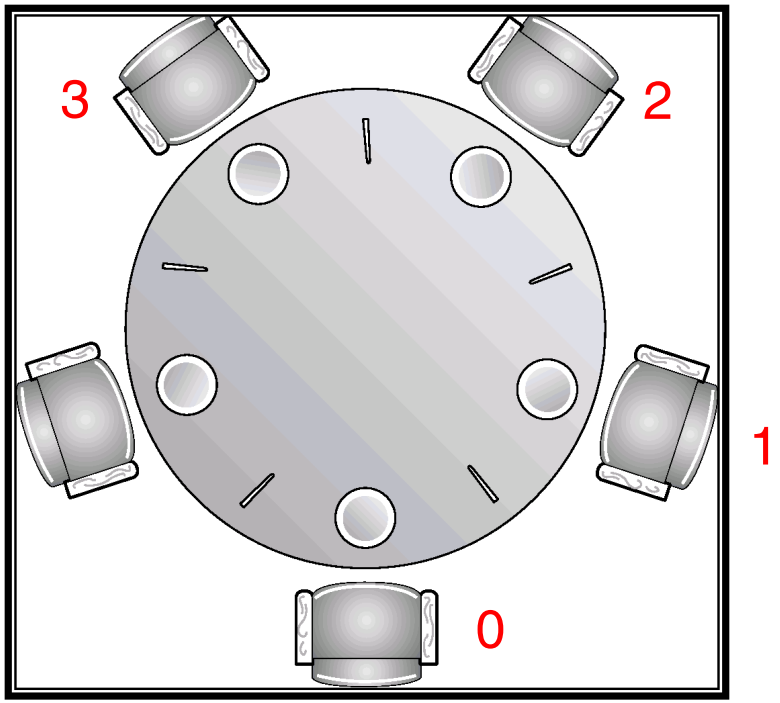

- Các process có thể đợi ở *entry queue* hoặc đợi ở các *condition queue* (*a*, *b*,…)
- Khi thực hiện lệnh a.wait, process sẽ block và được chuyển vào condition queue a.
- Lệnh a.signal chuyển một process từ condition queue a sang trạng thái ready

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

# Monitor có condition variable (2/2)



# Monitor và dining philosophers (1/5)




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

# Monitor và dining philosophers (2/5)

```
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);      // left neighbor
        test((i + 1) % 5);      // right …
}
```

# Monitor và dining philosophers (3/5)

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

# Monitor và dining philosophers (4/5)

- 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
```

- Câu hỏi: Nếu một triết gia phải đợi thì sẽ được phục hồi do ai gọi signal lên condition variable và từ đâu?

# Monitor và dining philosophers (5/5)

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