Chương V - Phần II

Đồng Bộ và Giải Quyết Tranh Chấp (Process Synchronization)

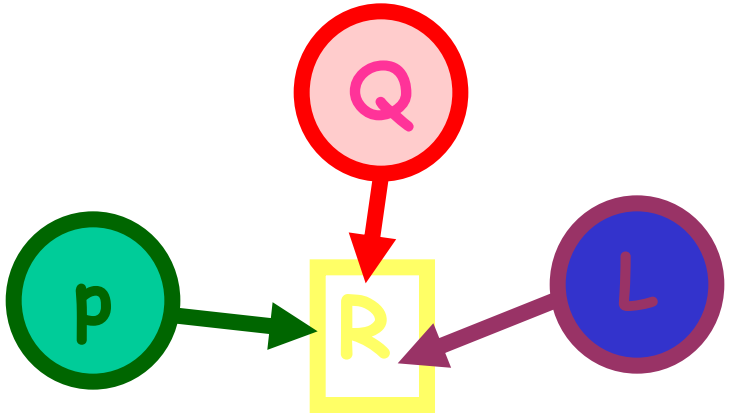
Nội dung

- Đặt vấn đề (tại sao phải đồng bộ và giải quyết tranh chấp ?)
- Vấn đề Critical section
- Các giải pháp phần mềm
  - Giải thuật Peterson, và giải thuật bakery
- Đồng bộ bằng hardware
- Semaphore
- Các bài toán đồng bộ
- Critical region
- Monitor

Đặt vấn đề

Khảo sát các process/thread thực thi đồng thời và chia sẻ dữ liệu (qua shared memory, file).

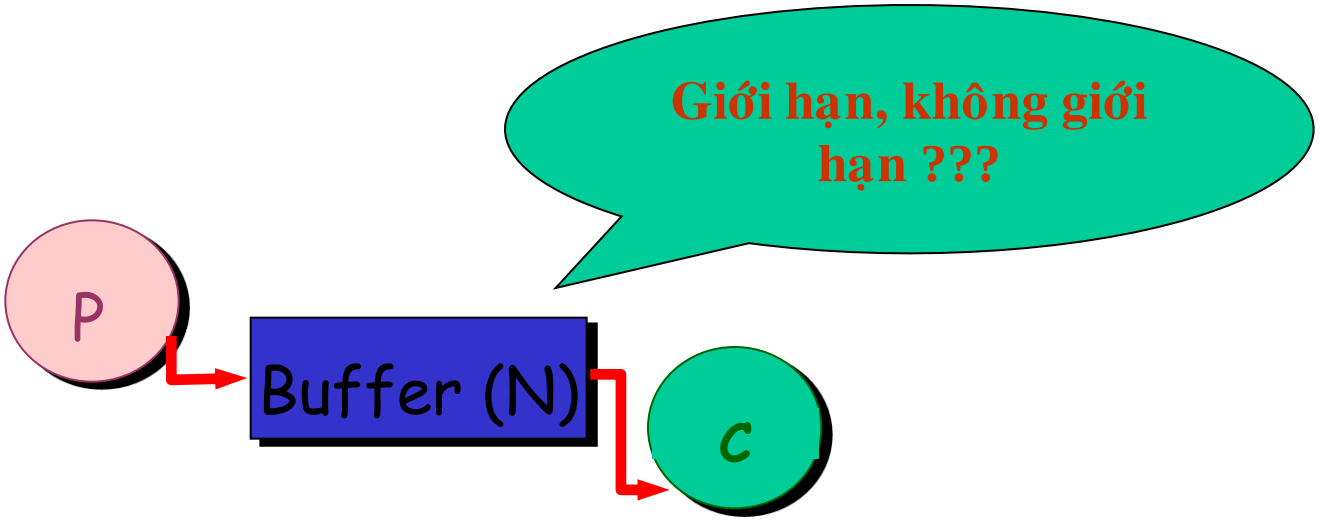
- Nếu không có sự kiểm soát khi truy cập các dữ liệu chia sẻ thì có thể đưa đến ra trường hợp *không nhất quán dữ liệu* (data inconsistency).
- Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế bảo đảm sự thực thi có trật tự của các process đồng thời.



Bài toán Producer-Consumer

Producer-Consumer

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



Đặt vấn đề

- Xét bài toán Producer-Consumer với bounded buffer
- 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;
```

Bounded buffer (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

Bounded buffer (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, các $register_i$ là các thanh ghi của CPU.

Bounded buffer (tt)

Mã máy của các lệnh tăng và giảm biến count có thể bị thực thi xen kẽ

- Giả sử count đang bằng 5. Chuỗi thực thi sau có thể xảy ra:

0:	producer	$register_1$:= count	{$register_1$ = 5}
1:	producer	$register_1$:= $register_1$ + 1	{$register_1$ = 6}
2:	consumer	$register_2$:= count	{$register_2$ = 5}
3:	consumer	$register_2$:= $register_2$ - 1	{$register_2$ = 4}
4:	producer	count := $register_1$	**{count = 6}**
5:	consumer	count := $register_2$	**{count = 4}**

Các lệnh count++, count-- phải là *đơn nguyên* (atomic), nghĩa là thực hiện như một lệnh đơn, không bị ngắt nửa chừng.

Bounded buffer (tt)

- *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 tại mỗi thời điểm chỉ có một process được thao tác lên dữ liệu chia sẻ. Do đó, cần có cơ chế đồng bộ hoạt động của các process này.

Vấn đề Critical Section

- Giả sử có *n* process cùng truy xuất đồng thời dữ liệu chia sẻ
- Cấu trúc của mỗi process Pi - Mỗi process có đoạn code như sau :

```
Do {
    entry section      /* vào critical section */
      critical section  /* truy xuất dữ liệu chia xẻ */
    exit  section       /* rời critical section */
      remainder section  /* làm những việc khác */
} While (1)
```

- Trong mỗi process có 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*).

Vấn đề Critical Section

- Vấn đề Critical Section: phải bảo đảm sự *loại trừ tương hỗ* (MUTual EXclusion, mutex), tức là khi một process đang thực thi trong vùng tranh chấp, không có process nào khác đồng thời thực thi các lệnh trong vùng tranh chấp.

Yêu cầu của lời giải cho Critical Section Problem

Lời giải phải thỏa bốn tính chất:

(1) Độc quyền truy xuất (*Mutual exclusion)*: Khi một process P đang thực thi trong vùng tranh chấp (CS) của nó thì không có process Q nào khác đang thực thi trong CS của Q.

(2) Progress: Một tiến trình tạm dừng bên ngoài miền găng (CS) không được ngăn cản các tiến trình khác vào miền găng (CS) và việc lựa chọn P nào vào CS phải có hạn định

(3) Chờ đợi giới hạn (*Bounded waiting):* Mỗi process chỉ phải chờ để được vào vùng tranh chấp trong một khoảng thời gian có hạn định nào đó. Không xảy ra tình trạng *đói tài nguyên* (starvation).

(4)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

Phân loại giải pháp

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

Các giải pháp "Busy waiting"

While (chưa có quyền) do nothing() ;

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**

Các giải pháp "Sleep & Wake up"

if (chưa có quyền) Sleep() ;

CS;

Wakeup(somebody);

- **Từ bỏ CPU khi chưa được vào miền găng**
- **Cần được Hệ điều hành hỗ trợ**

Các giải pháp "Busy waiting"

Giải thuật 1

- 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);
```

- Thoả mãn mutual exclusion (1)
- Nhưng không thoả mãn yêu cầu về progress (2) và bounded waiting (3) vì tính chất strict alternation của giải thuật

Giải thuật 1 (tt)

```
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);
```

Ví dụ:
P0 có RS (remainder section) rất lớn còn P1 có RS nhỏ???

P0 (I/O) P1 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;      /* Pi "sẵn sàng" vào CS */
        while ( flag[ j ] );   /* Pi "nhường" Pj       */
            critical section
        flag[ i ] = false;
            remainder section
} while (1);
```

- Bảo đảm được mutual exclusion. Chứng minh?
- Không thỏa mãn progress. Vì sao?

Giải thuật 3 (Peterson)

- 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  */
        turn = j;                /*  Nhường process j    */
        while (flag[ j ] and turn == j);
                critical section
        flag[ i ] = false;
                remainder section
} while (1);
```

- Thoả mãn được cả 3 yêu cầu (chứng minh?)
  $\Rightarrow$ giải quyết bài toán critical section cho 2 process.

Giải thuật Peterson-2 process

Process P_0

```
do {
        /*  0 wants in  */
       flag[0] = true;
       /*  0 gives a chance to 1  */
       turn = 1;
       while (flag[1] && turn == 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  */
       turn = 0;
       while (flag[0] && turn == 0);
          critical section;
        /*  1 no longer wants in  */
       flag[1] = false;
          remainder section;
} while(1);
```

Giải thuật 3: Tính đúng đắn

Giải thuật 3 thỏa mutual exclusion, progress, và bounded waiting

- Mutual exclusion được bảo đảm bởi vì

  P0 và P1 đều ở trong CS nếu và chỉ nếu flag[0] = flag[1] = true và *turn = i* cho mỗi Pi (không thể xảy ra)

- Chứng minh thỏa yêu cầu về progress và bounded waiting
  - Pi không thể vào CS nếu và chỉ nếu bị kẹt tại vòng lặp while() với điều kiện *flag[j] = true và turn = j* .
  - Nếu Pj không muốn vào CS thì flag[j] = false và do đó Pi có thể vào CS.

Giải thuật 3: Tính đúng đắn (tt)

- Nếu Pj đã bật flag[j] = true và đang chờ tại while() thì có chỉ hai trường hợp là turn = i hoặc turn = j
- Nếu turn = i thì Pi vào CS. Nếu turn = j thì Pj vào CS nhưng sẽ bật flag[j] = false khi thoát ra $\Rightarrow$ cho phép Pi vào CS
- Nhưng nếu Pj có đủ thời gian bật flag[j] = true thì Pj cũng phải gán turn = i
- Vì Pi không thay đổi trị của biến turn khi đang kẹt trong vòng lặp while(), Pi sẽ chờ để vào CS nhiều nhất là sau một lần Pj vào CS (bounded waiting)

Giải thuật bakery: *n* process

- Trước khi vào CS, process Pi nhận một con số. Process nào giữ con số nhỏ nhất thì được vào CS
- Trường hợp Pi và Pj cùng nhận được một chỉ số:
  - Nếu i < j thì Pi được vào trước. (Đối xứng)
- Khi ra khỏi CS, Pi đặt lại số của mình bằng 0
- Cơ chế cấp số cho các process thường tạo các số theo cơ chế 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 $b < d$

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

Giải thuật bakery: *n* process (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; // không chiếm quyền vào vùng CS
            remainder section
} while (1);
```

Từ software đến hardware

- Khuyết điểm của các giải pháp software
  - Các process khi yêu cầu được vào vùng tranh chấp đều phải liên tục kiểm tra điều kiện (busy waiting), tốn nhiều 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 phần cứng (hardware)
  - Cấm ngắt (disable interrupts)
  - Dùng các lệnh đặc biệt

Cấm ngắt

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

```
Process Pi:
do {
        disable_interrupts();
            critical section
        enable_interrupts();
            remainder section
} while (1);
```

Lệnh *TestAndSet*

- Đọc và ghi một biến trong một thao tác *atomic* (không chia cắt được).

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

- 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, quá trình chọn lựa process P_j vào CS kế tiếp là tùy ý $\Rightarrow$ không bảo đảm điều kiện bounded waiting. Do đó có thể xảy ra *starvation* (bị bỏ đói)

- Các processor (ví dụ Pentium) thông thường cung cấp một lệnh đơn 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

Swap và mutual exclusion

- Biến chia sẻ **lock** được khởi tạo giá trị 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**

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

- Biến chia sẻ (khởi tạo là **false**)

  bool lock;

  bool key;

- Process P_i

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

Không thỏa mãn bounded waiting

Giải thuật dùng TestAndSet thoả mãn 3 yêu cầu (1)

- Cấu trúc dữ liệu dùng chung (khởi tạo là false)

  bool waiting[n];

  bool lock;

- Mutual exclusion: Pi 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: chứng minh tương tự như mutual exclusion
- Bounded waiting: waiting in the cyclic order

Giải thuật dùng TestAndSet thoả mãn 3 yêu cầu (2)

```
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)
```

Các giải pháp "Sleep & Wake up"

```
int busy;          // =1 nếu CS đang bị chiếm
Int blocked;       // số P đang bị khóa
do
{
    if (busy==1){
                              blocked = blocked +1;
                              sleep();
                    }
    else
                     busy =1;
          CS;
    busy = 0;
    if(blocked !=0){
                              wakeup(process);
                              blocked = blocked -1;
                       }
          RS;
} while(1);
```

Trường hợp:
-A vào CS
-B kích hoạt và tăng blocked
-A kích hoạt lại
-B kích hoạt lại
-?????

Semaphore

Là công cụ đồng bộ cung cấp bởi OS mà <u>không đòi hỏi busy waiting</u>

- *Semaphore* S là một **biến số nguyên**.
- Ngoài thao tác khởi động biến thì chỉ có thể được truy xuất qua hai tác vụ có tính đơn nguyên (atomic) và loại trừ (mutual exclusion)

  wait(S) hay còn gọi là P(S): giảm giá trị semaphore (S=S-1) . Kế đó nếu giá trị này âm thì process thực hiện lệnh wait() bị blocked.

  signal(S) hay còn gọi là V(S): tăng giá trị semaphore (S=S+1) . Kế đó nếu giá trị này không dương, một process đang blocked bởi một lệnh wait() sẽ được hồi phục để thực thi.
- 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.

Semaphore

- P(S) hay wait(S) sử dụng để *giành tài nguyên* và giảm biến đếm S=S-1
- V(S) hay signal(S) sẽ *giải phóng tài nguyên* và tăng biến đếm S= S+1
- Nếu P được thực hiện trên biến đếm <= 0 , tiến trình phải đợi V hay chờ đợi sự giải phóng tài nguyên

Hiện thực semaphore

- Định nghĩa semaphore là một record

```
typedef  struct {
        int  value;
        struct  process *L;/* process queue */
} semaphore;
```

- Giả sử hệ điều hành cung cấp hai tác vụ (system call):

  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 một process phải chờ trên semaphore S, nó sẽ bị blocked và được đặt trong hàng đợi semaphore
  - Hàng đợi này là ***danh sách liên kết các PCB***
- Tác vụ signal() thường sử dụng **cơ chế FIFO** khi chọn một process từ hàng đợi và đưa 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

Ví dụ sử dụng semaphore 1 : Hiện thực mutex với semaphore

- Dùng cho *n* process
- Khởi tạo S.value = 1

  Chỉ duy nhất một process được vào CS (mutual exclusion)
- Để cho phép *k* process vào 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);
```

Ví dụ sử dụng semaphore 2 :Đồng bộ process bằng semaphore

- 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

- 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$|$

- Atomic và mutual exclusion: không được xảy ra trường hợp **2 process cùng đang ở trong thân lệnh *wait*(S) và *signal*(S) (cùng semaphore S) tại một thời điểm** (ngay cả với hệ thống ***multiprocessor***)

  $\Rightarrow$ do đó, đoạn mã định nghĩa các lệnh wait(S) và signal(S) cũng chính là vùng tranh chấp

Nhận xét (tt)

- Vùng tranh chấp của các tác vụ wait(S) và signal(S) thông thường rất nhỏ: khoảng 10 lệnh.

- Giải pháp cho vùng tranh chấp wait(S) và signal(S)
  - Uniprocessor: có thể dùng cơ chế cấm ngắt (disable interrupt). Nhưng phương pháp này không làm việc trên hệ thống multiprocessor.
  - Multiprocessor: có thể dùng các giải pháp software (như giải thuật Dekker, Peterson) hoặc giải pháp hardware (TestAndSet, Swap).

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

Deadlock và starvation

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

```
P0                      P1

wait(S);                wait(Q);
wait(Q);                wait(S);
    :                       :
signal(S);              signal(Q);
signal(Q);              signal(S);
```

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) Một tiến trình có thể không bao giờ được lấy ra khỏi hàng đợi mà nó bị treo trong hàng đợi đó.

Các loại semaphore

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

Các bài toán đồng bộ (kinh điển)

- Bounded Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

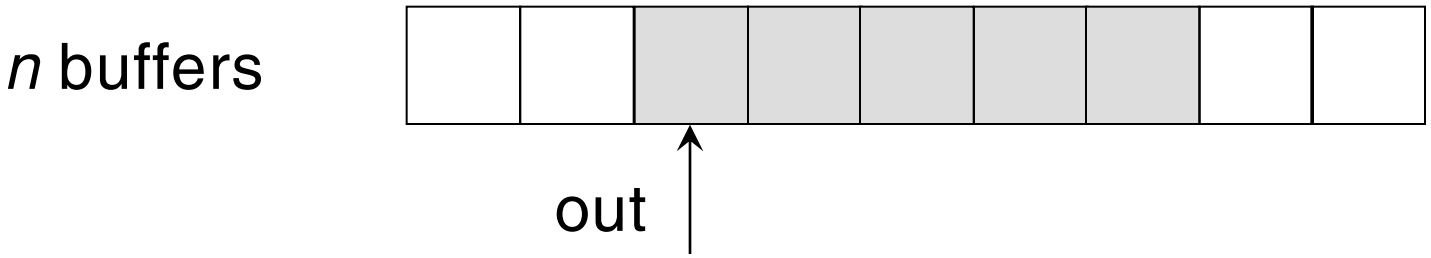
Các bài toán đồng bộ

- 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;
```



Bounded buffer

producer

```
do {
        …
        nextp = new_item();
        …
        wait(empty);
        wait(mutex);//1
        …
        insert_to_buffer(nextp);
        …
        signal(mutex);//3
        signal(full);
} while (1);
```

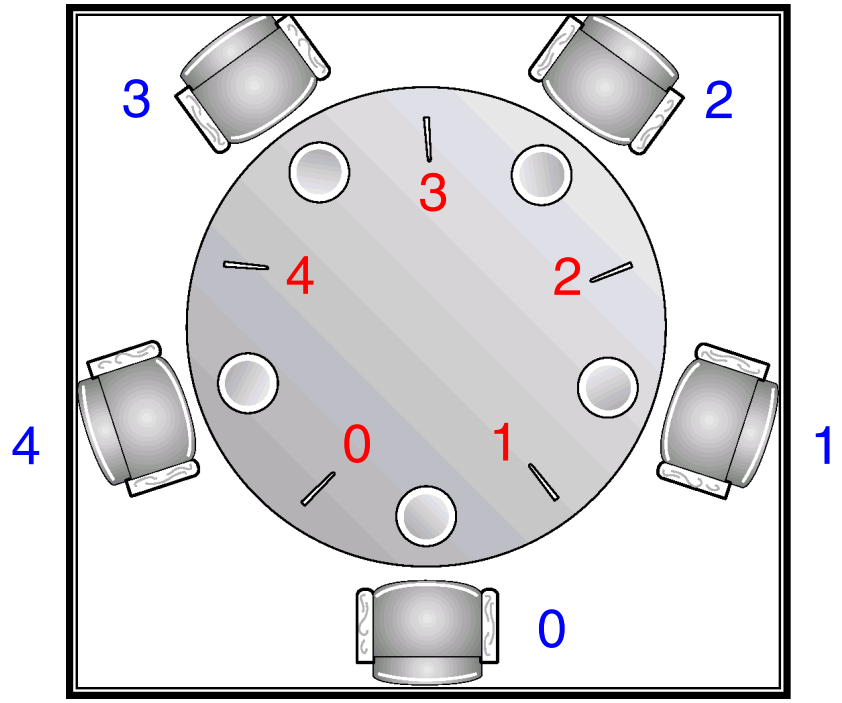
consumer

```
do {
        wait(full)
        wait(mutex);//2
        …
        nextc = get_buffer_item(out);
        …
        signal(mutex);//4
        signal(empty);
        …
        consume_item(nextc);
        …
} while (1);
```

Bài toán "Dining Philosophers" (1)

- 5 triết gia ngồi ăn và suy nghĩ
- Mỗi người cần 2 chiếc đũa (chopstick) để ăn
- Trên bàn chỉ có 5 đũa

- Bài toán này minh họa sự khó khăn trong việc phân phối tài nguyên giữa các process sao cho không xảy ra deadlock và starvation



- Dữ liệu chia sẻ:
  semaphore chopstick[5];

- Khởi đầu các biến đều là 1

Bài toán "Dining Philosophers" (2)

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);
```

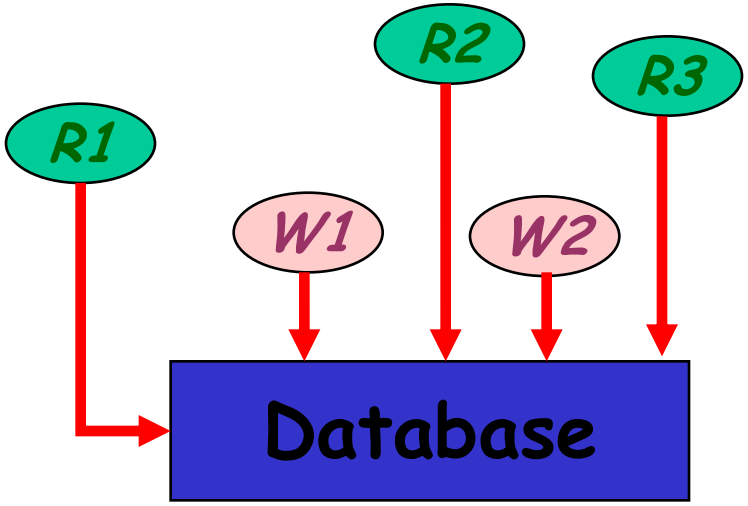
Bài toán "Dining Philosophers" (3)

- Giải pháp trên có thể gây ra deadlock
  - Khi tất cả triết gia đói bụng cùng lúc và đồng thời cầm chiếc đũa bên tay trái ⇒ deadlock
- Một số giải pháp khác giải quyết được deadlock
  - Cho phép nhiều nhất 4 triết gia ngồi vào cùng một lúc
  - 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 (1)

Readers - Writers

- **W không được cập nhật dữ liệu khi có một R đang truy xuất CSDL .**
- **Tại một thời điểm , chỉ cho phép một W được sửa đổi nội dung CSDL.**



Bài toán Readers-Writers (2)

- Bộ đọc trước bộ ghi (first reader-writer)
- Dữ liệu chia sẻ

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

- Writer process

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

- 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)

- 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

- Semaphore cung cấp một công cụ mạnh mẽ để bảo đảm mutual exclusion và phối hợp đồng bộ các process
- Tuy nhiên, nếu các tác vụ wait(S) và signal(S) nằm rải rác ở rất nhiều processes ⇒ khó nắm bắt được hiệu ứng của các tác vụ này. Nếu không sử dụng đúng ⇒ có thể xảy ra tình trạng 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)
```

Critical Region (CR)

- Là một cấu trúc ngôn ngữ cấp cao (high-level language construct, được dịch sang mã máy bởi một compiler), thuận tiện hơn cho người lập trình.

- Một biến chia sẻ *v* kiểu dữ liệu *T*, khai báo như sau

  v: shared T;

- Biến chia sẻ *v* chỉ có thể được truy xuất qua phát biểu sau

  region v when B do S; /* B là một biểu thức Boolean */

  Ý nghĩa: trong khi S được thực thi, không có quá trình khác có thể truy xuất biến v.

CR và bài toán bounded buffer

Dữ liệu chia sẽ:

```
struct  buffer
{
        int  pool[n];
        int  count,
             in,
             out;
}
```

Producer

```
region buffer when (count < n) {
        pool[in] = nextp;
        in = (in + 1) % n;
        count++;
}
```

Consumer

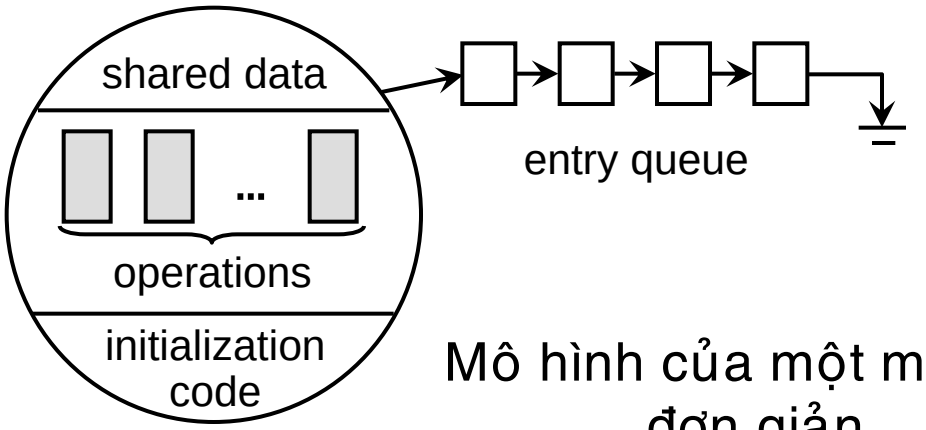
```
region buffer when (count > 0){
        nextc = pool[out];
        out = (out + 1) % n;
        count--;
}
```

Monitor (1)

- Cũng là một cấu trúc ngôn ngữ cấp cao tương tự CR, có chức năng như semaphore nhưng dễ điều khiển hơn
- Xuất hiện trong nhiều ngôn ngữ lập trình đồng thời như
  - Concurrent Pascal, Modula-3, Java,…
- Có thể hiện thực bằng semaphore

Monitor (2)

- Là một 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)



Mô hình của một monitor đơn giản

- Đặc tính của monitor
  - Local variable chỉ có thể truy xuất bởi các thủ tục của monitor
  - Process "vào monitor" bằng cách gọi một trong các thủ tục đó
  - Chỉ có một process có thể vào monitor tại một thời điểm ⇒ mutual exclusion được bảo đảm

Cấu trúc của monitor

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

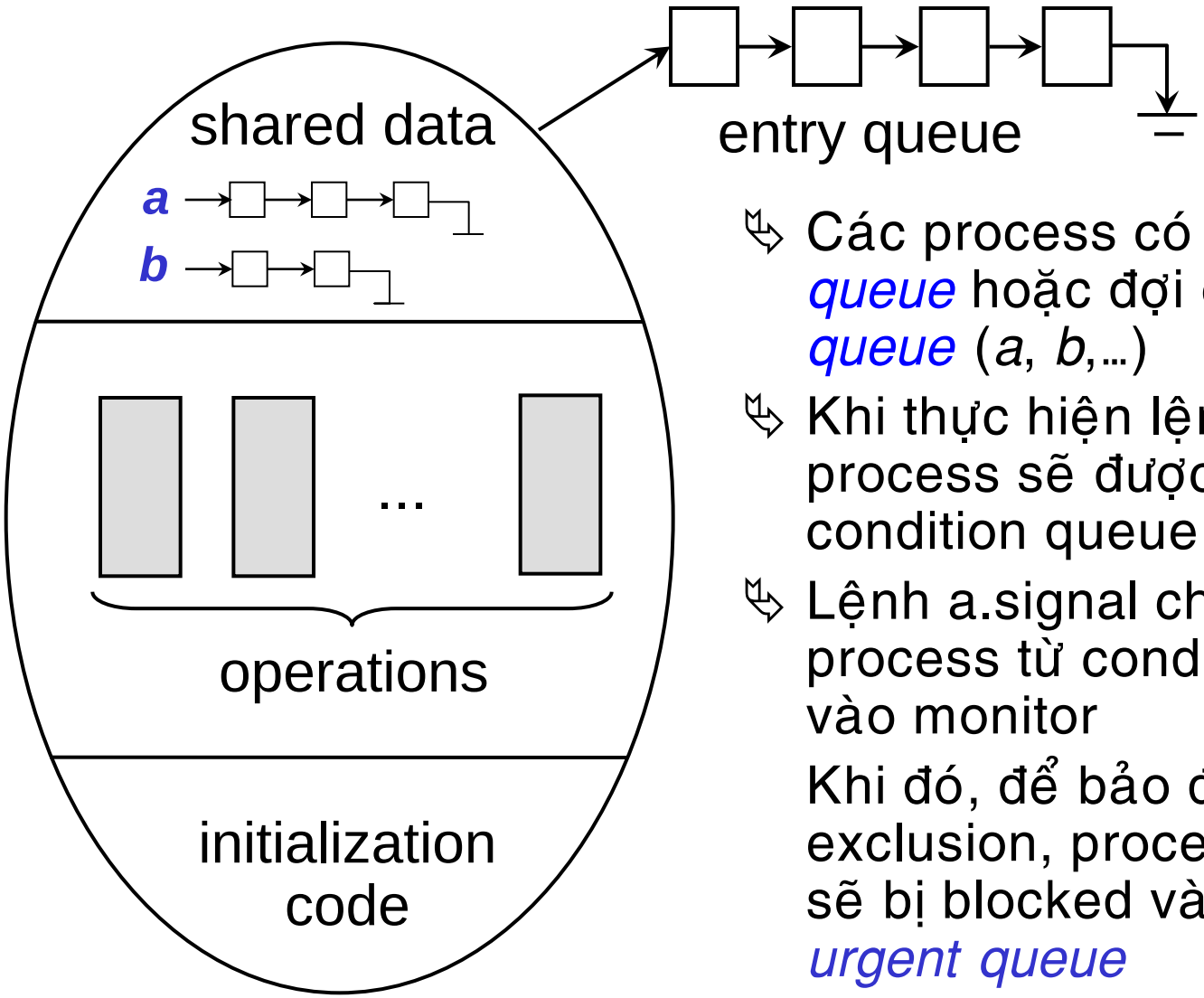
Condition variable

- 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

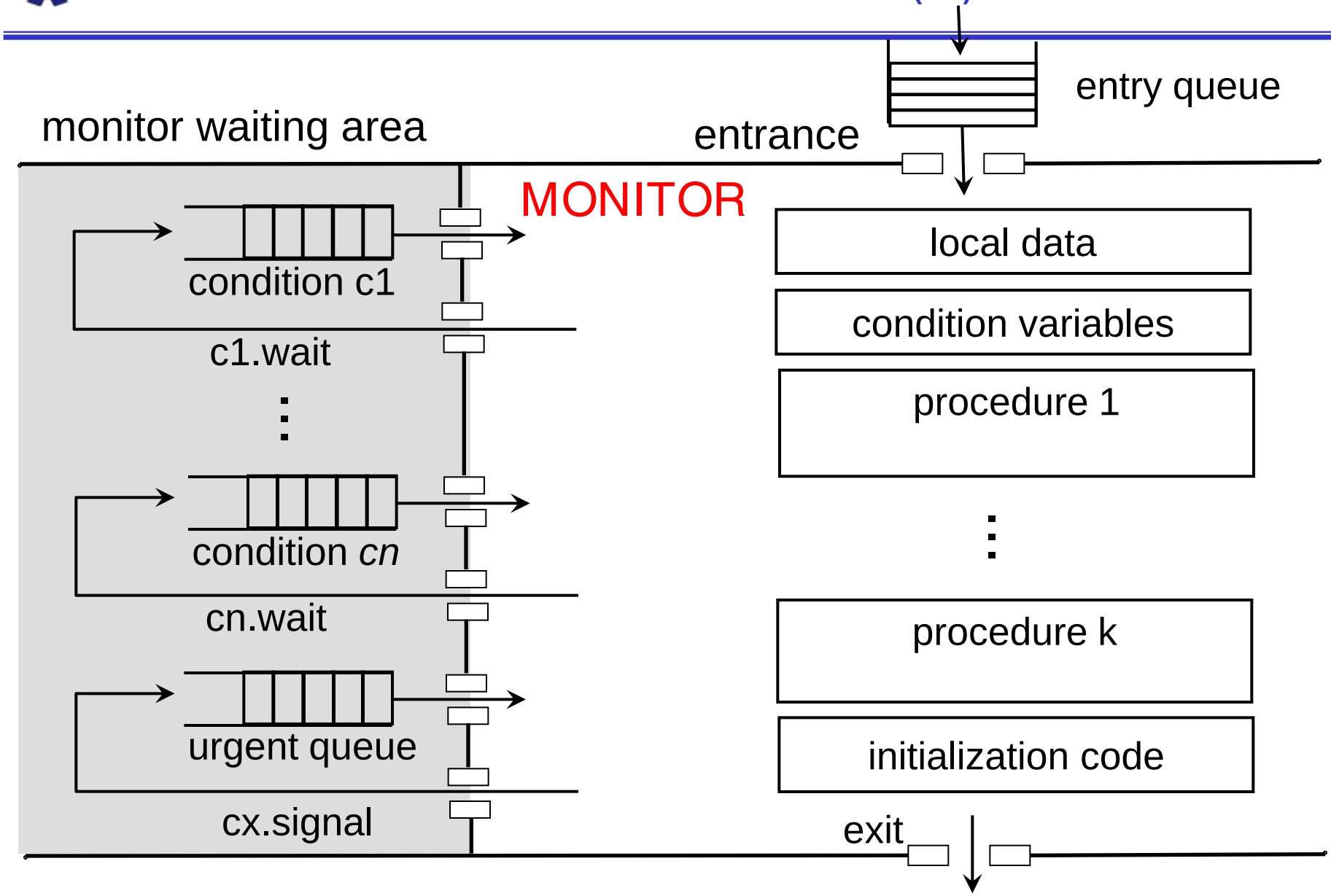
Monitor có condition variable



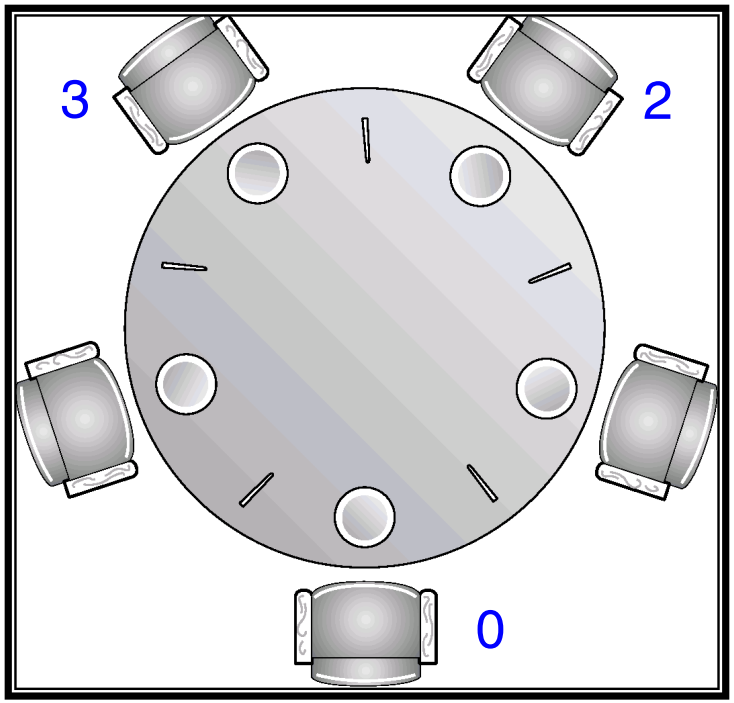
- Các process có thể đợi ở *entry queue* hoặc đợi ở các *condition queue* (*a*, *b*,…)
- Khi thực hiện lệnh a.wait, process sẽ được chuyển vào condition queue a
- Lệnh a.signal chuyển một process từ condition queue a vào monitor

  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];
```

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 …
}
```

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;
        }
}
```

Dining philosophers (tt)

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

```
dp.pickup(i);
ăn
dp.putdown(i);
```

- Giải thuật không deadlock nhưng có thể gây starvation.

Bài Tập

Bài 1. Xét giải pháp đồng bộ hoá sau :

```
while (TRUE) {
int j = 1-i;
flag[i]= TRUE;          turn = j;
while (turn == j && flag[j]==TRUE);
critical-section ();
flag[j] = FALSE;
    Noncritical-section ();
}
```

Đây có phải là một giải pháp bảo đảm 3 điều kiện không ?

Bài tập

Bài 1 : Xét giải pháp phần mềm do Dekker đề nghị để tổ chức truy xất độc quyền cho hai tiến trình . Hai tiến trình P0, P1 chia sẻ các biến sau :

var flag : array [0..1] of boolean; (khởi động là false)
turn : 0..1;

Cấu trúc một tiến trình Pi (i =0 hay 1, và j là tiến trình còn lại) như sau :

```
repeat
flag[i] := true;
while flag[j] do
if turn = j then
      begin              flag[i]:= false;
                         while turn = j do ;
                         flag[i]:= true;
      end;
   critical_section();
turn:= j;
flag[i]:= false;
non_critical_section();
until false;
```

Giải pháp này có phải là một giải pháp đúng thỏa mãn 4 yêu cầu không ?

Bài tập

Giả sử một máy tính không có chỉ thị TSL, nhưng có chỉ thị Swap có khả năng hoán đổi nội dung của hai từ nhớ chỉ bằng một thao tác không thể phân chia :

```
procedure Swap() var a,b: boolean);
var        temp : boolean;
begin
   temp := a;
   a:= b;
   b:= temp;
end;
```

Sử dụng chỉ thị này có thể tổ chức truy xuất độc quyền không ? Nếu có, xây dựng cấu trúc chương trình tương ứng

Bài tập – semanphore

Xét hai tiến trình xử lý đoạn chương trình sau :

process P1 { A1 ; A2 } process P2 { B1 ; B2 }

Đồng bộ hoá hoạt động của hai tiến trình này sao cho cả A1 và B1 đều hoàn tất trước khi A2 hay B2 bắt đầu .

Bài tập – semanphore

Sử dụng semaphore để viết lại chương trình sau theo mô hình xử lý đồng hành:

A = x1 + x2; B = A*x3; C= A + x4; D= B + C; E = D*x5 + C;

Giả sử có 5 process mỗi process sẽ thực hiện 1 biểu thức.

Bài tập – semanphore

- Xét hai tiến trình sau :

```
process A  {      while (TRUE)   na = na +1;      }
process B  {      while (TRUE)   nb = nb +1;      }
```

  - Đồng bộ hoá xử lý của hai tiến trình trên, sử dụng hai semaphore tổng quát, sao cho tại bất kỳ thời điểm nào cũng có nb < na <= nb +10
  - Nếu giảm điều kiện chỉ là na <= nb +10, giải pháp của bạn sẽ được sửa chữa như thế nào ?

Bài tập – semanphore

Một biến X được chia sẻ bởi hai tiến trình cùng thực hiện đoạn code sau :

```
do
    X = X +1;
    if ( X == 20) X = 0;
while ( TRUE );
```

Bắt đầu với giá trị X = 0, chứng tỏ rằng giá trị X có thể vượt quá 20. Cần sửa chữa đoạn chương trình trên như thế nào để bảo đảm X không vượt quá 20 ?

Bài tập – semanphore

Tổng quát hoá câu hỏi 8) cho các tiến trình xử lý đoạn chương trình sau :

process P1 { for (i = 1; i <= 100; i ++) Ai }

process P2 { for (j = 1; j <= 100; j ++) Bj }

Đồng bộ hoá hoạt động của hai tiến trình này sao cho cả với *k* bất kỳ ($2 \leq k \leq 100$), Ak chỉ có thể bắt đầu khi B(k-1) đã kết thúc, và Bk chỉ có thể bắt đầu khi A(k-1) đã kết thúc.