COMPUTER ENGINEERING

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

HỆ ĐIỀU HÀNH
Chương 5 – Đồng bộ (3)

8/23/2023

Ôn tập chương 5 (2)

- Khi nào thì xảy ra tranh chấp race condition?
- Vấn đề Critical Section là gì?
- Yêu cầu của lời giải cho CS problem?
- Có mấy loại giải pháp? Kể tên?

Mục tiêu chương 5 (3)

- Biết được các giải pháp đồng bộ tiến trình theo kiểu “Sleep & Wake up” bao gồm:
  - Semaphore
  - Critical Region
  - Monitor
- Áp dụng các giải pháp này vào các bài toán đồng bộ kinh điển

Nội dung chương 5 (2)

- Các giải pháp "Sleep & Wake up"
  - Semaphore
  - Các bài toán đồng bộ kinh điển
  - Critical Region
  - Monitor
- Áp dụng các giải pháp này vào các bài toán đồng bộ kinh điển

Các giải pháp "Sleep & Wake up"

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

Semaphore

- Là công cụ đồng bộ cung cấp bởi OS mà không đòi hỏi busy waiting
- Semaphore S là một biến số nguyên.
- Ngoài thao tác khởi động biến thì chỉ có thể được truy xuất qua hai tác vụ cố tính đơn nguyên (atomic) và loại trừ (mutual exclusive):
  - 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 (tt)

- 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

Semaphore (tt)

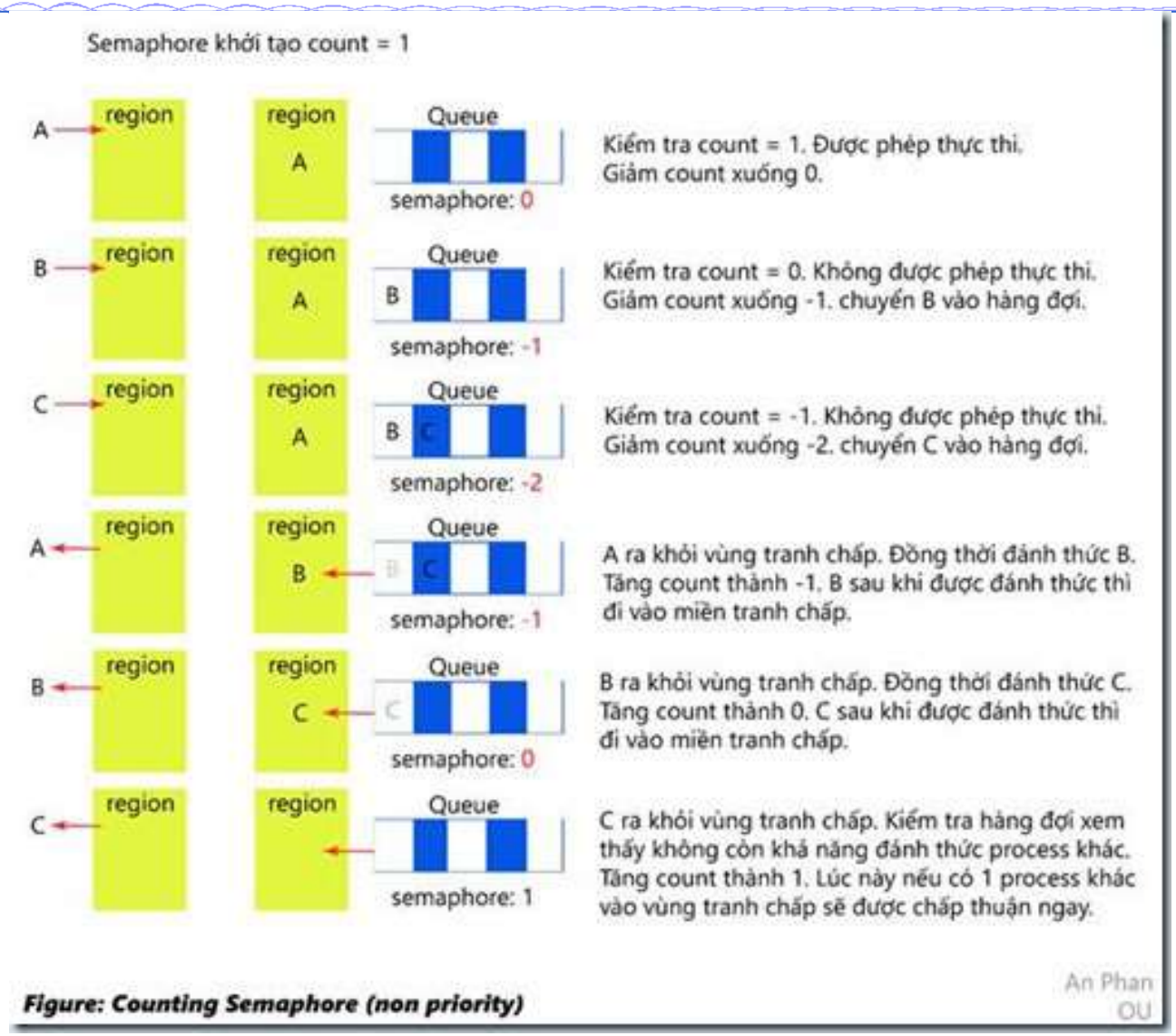


Figure: Counting Semaphore (non priority)

https://anphanhv.wordpress.com/2013/11/13/part-iii-process-giao-tiep-giua-cc-process/

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

- Dùng cho n process
- Chỉ duy nhất một process được vào CS (mutual exclusion)
- Khởi tạo S.value = 1

- 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

- 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;
```

Ví dụ sử dụng semaphore 3

- Xét 2 tiến trình xử lý đoạn chương trình sau:
  - Tiến trình P1 {A1, A2}
  - Tiến trình P2 {B1, B2}
- Đồng bộ hóa hoạt động của 2 tiến trình sao cho cả A1 và B1 đều hoàn tất trước khi A2 và B2 bắt đầu.
- Khởi tạo

semaphore s1.v = s2.v = 0

Để đồng bộ hoạt động theo yêu cầu, P1 phải định nghĩa như sau:

```
A1;
signal(s1);,
wait(s2);
A2;
```

Và P2 định nghĩa như sau:

```
B1
signal(s2);
wait(s1);
B2;
```

Nhận xét

- Khi S.value ≥ 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)

⇒ 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

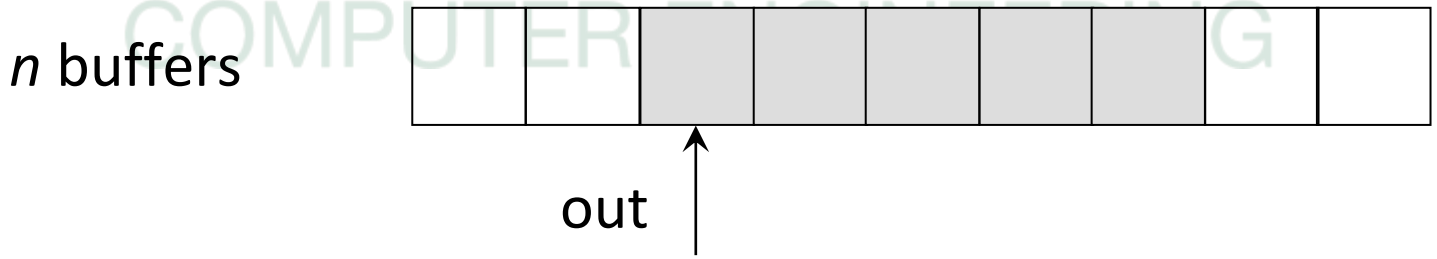
- Counting semaphore: một số nguyên có giá trị không hạn chế.
- Binary semaphore: có trị là 0 hay 1. Binary semaphore rất dễ hiện thực.
- Có thể hiện thực counting semaphore bằng binary semaphore.

Các bài toán đồng bộ kinh điển

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

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;



Bài toán bounded buffer (tt)

producer

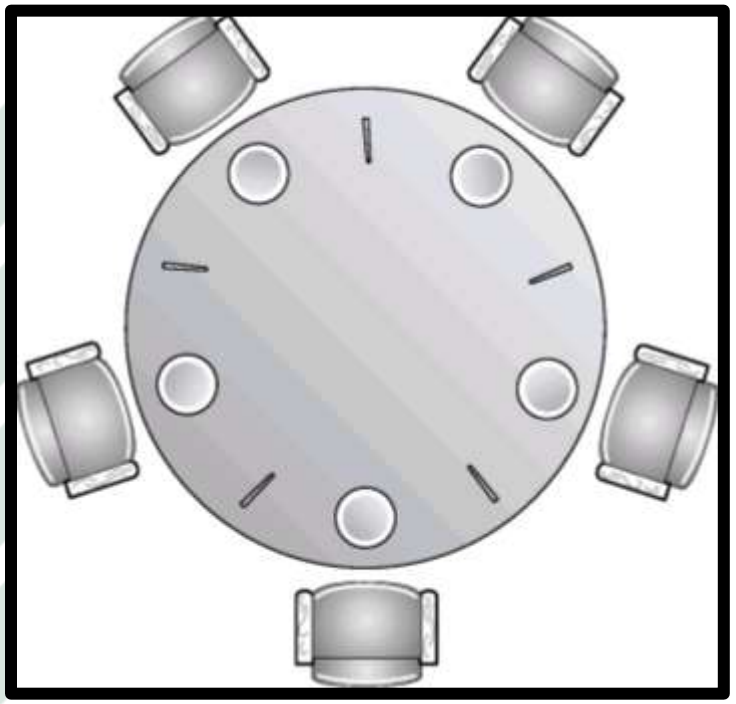
```
do {
        …
        nextp = new_item();
        …
        wait(empty);
        wait(mutex);
        …
        insert_to_buffer(nextp);
        …
        signal(mutex);
        signal(full);
} while (1);
```

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
- 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" (tt)

Triết gia thứ i:

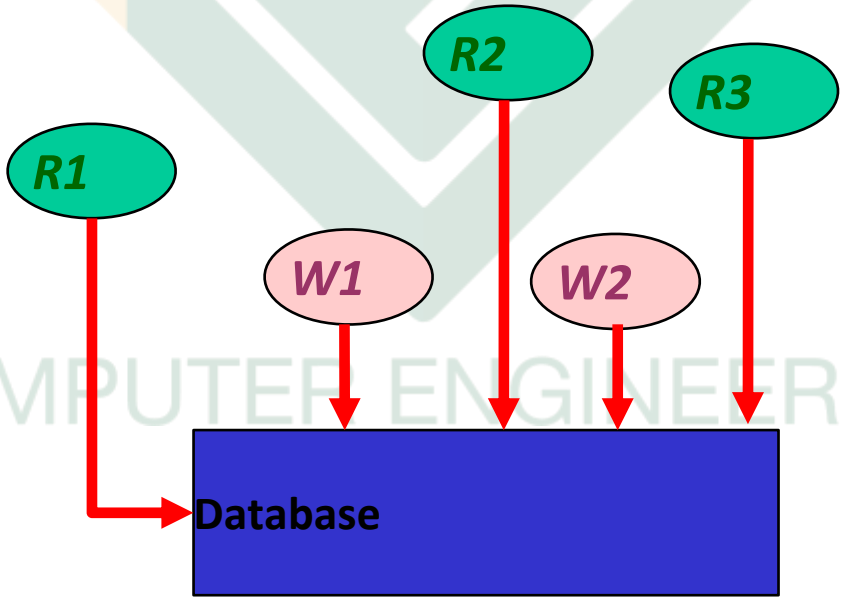
```
do {
    wait(chopstick [ i ])
    wait(chopstick [ (i + 1) % 5 ])
     …
    eat
     …
    signal(chopstick [ i ]);
    signal(chopstick [ (i + 1) % 5 ]);
     …
    think
     …
} while (1);
```

Bài toán "Dining Philosophers" (tt)

- Giải pháp trên có thể gây ra deadlock
  - Khi tất cả triết gia 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 Reader-Writers

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



Bài toán Reader-Writers (tt)

- 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 Reader-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

- 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 sẻ:

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

Producer

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

Consumer

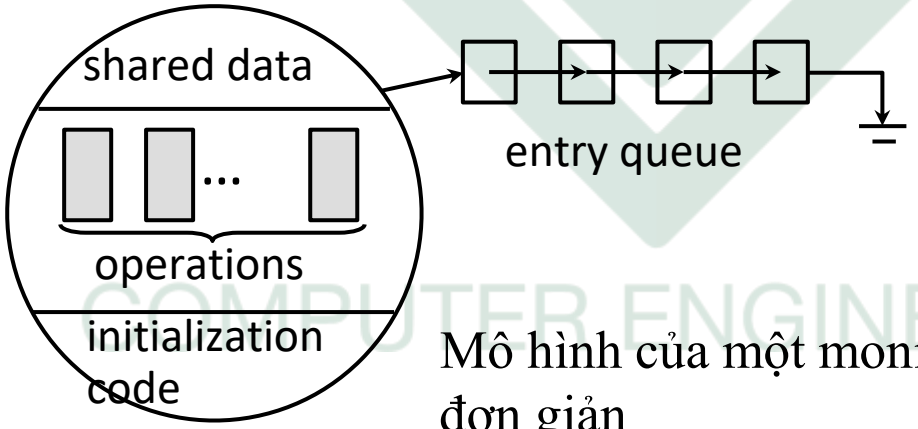
```
region buffer when (count > 0){
        nextc = pool[out];
        out = (out + 1) % n;
        count--;
}
```

Monitor

- 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 (tt)

- 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

Monitor (tt)

- 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
        }
}
```

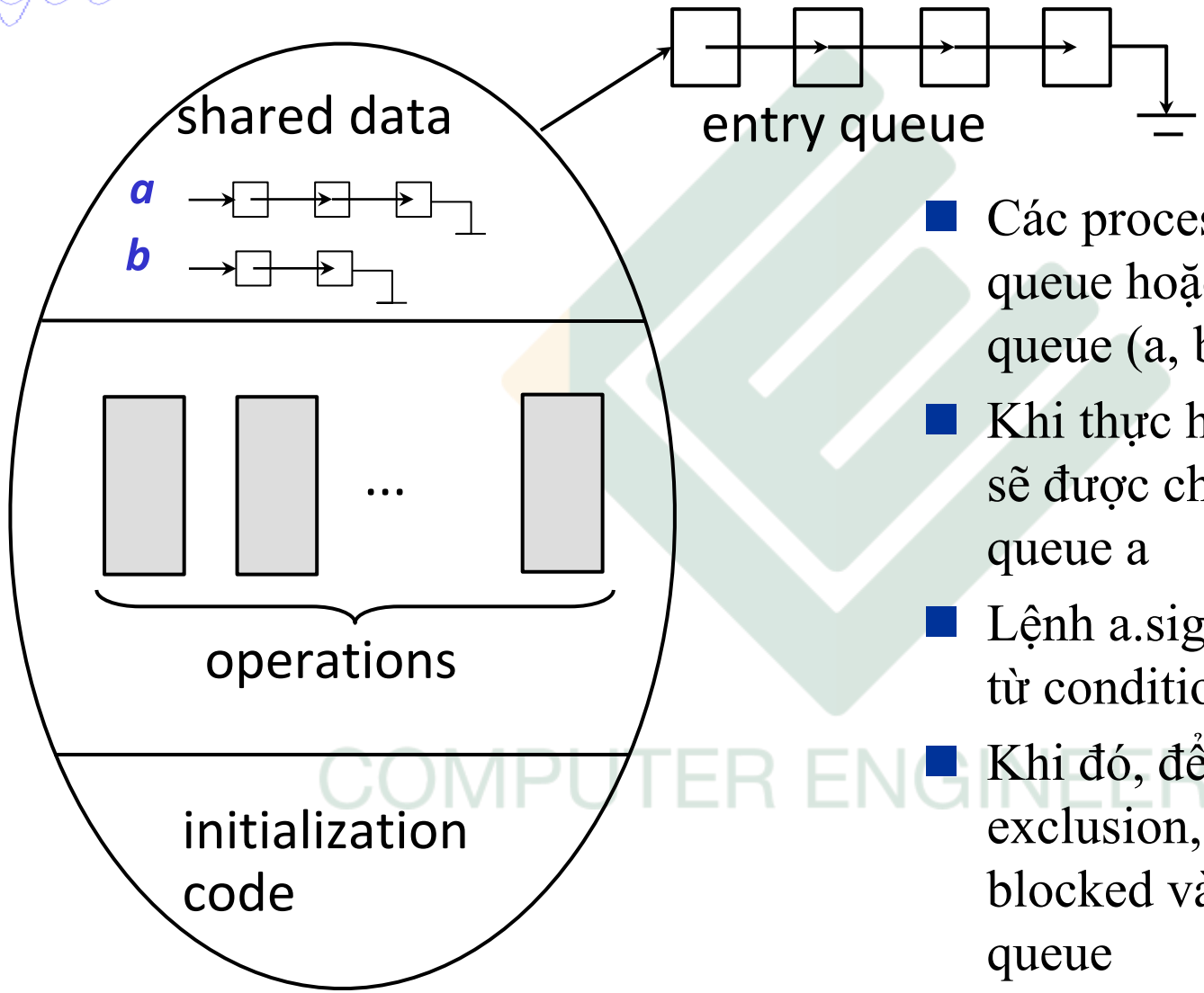
Condition variable

- Nhằm cho phép một process đợi "trong monitor", phải khai báo biến điều kiện (condition variable)

  condition a, b;

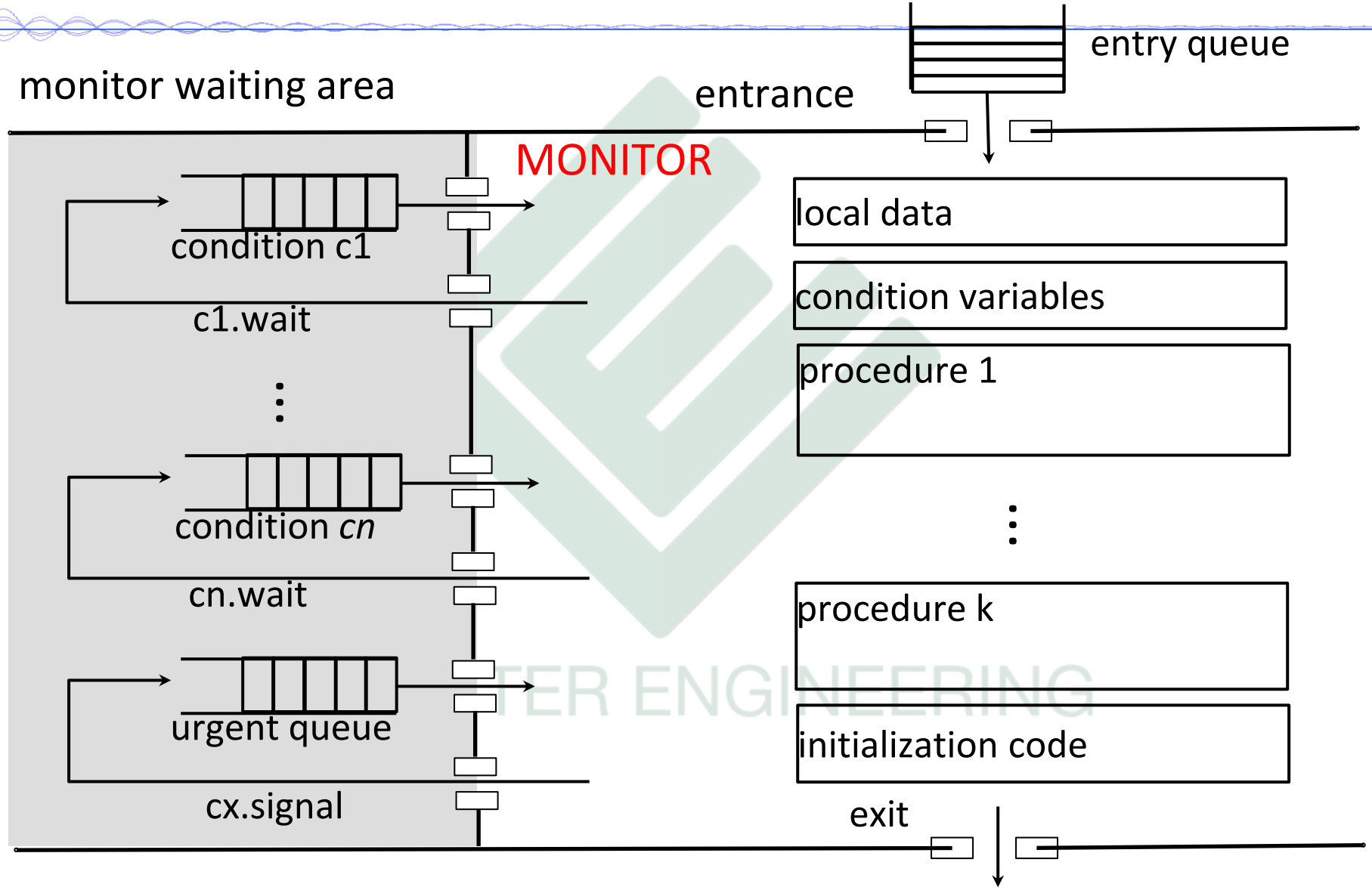
- Các biến điều kiện đều cục bộ và chỉ được truy cập bên trong monitor.
- Chỉ có thể thao tác lên biến điều kiện bằng hai thủ tục:
  - a.wait: process gọi tác vụ này sẽ bị "block trên biến điều kiện" a
    - Process này chỉ có thể tiếp tục thực thi khi có process khác thực hiện tác vụ a.signal
  - a.signal: 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

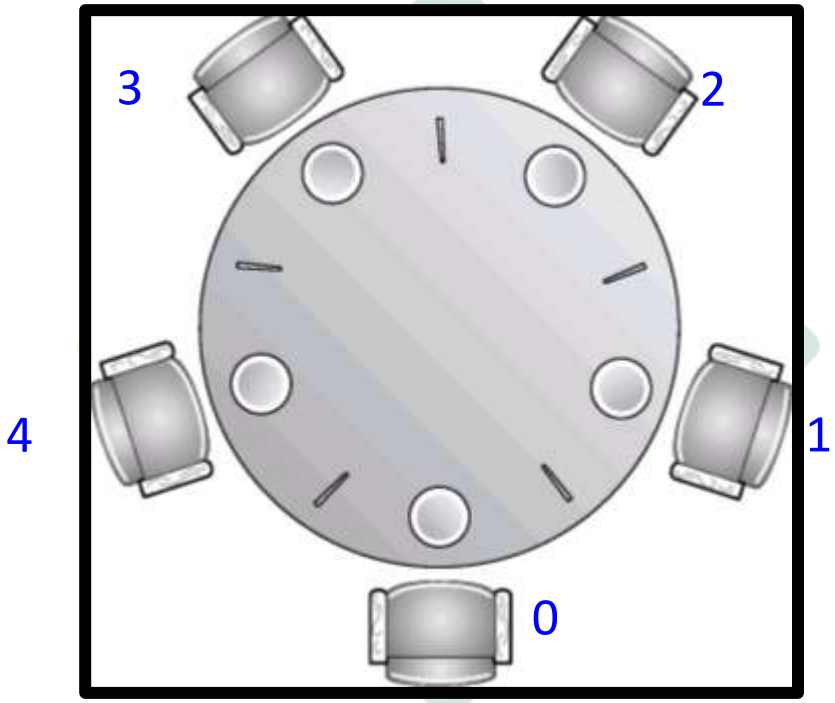


- Các process có thể đợi ở entry queue hoặc đợi ở các condition queue (a, b,…)
- Khi thực hiện lệnh a.wait, process sẽ được chuyển vào condition queue a
- Lệnh a.signal chuyển một process từ condition queue a vào monitor
- Khi đó, để bảo đảm mutual exclusion, process gọi a.signal sẽ bị blocked và được đưa vào urgent queue

Monitor có condition variable (tt)



Monitor và dining philosophers




```
monitor dp {
        enum {thinking, hungry, eating} state[5];
        condition self[5];
```

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()

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

- Giải thuật không deadlock nhưng có thể gây starvation.

Các bài toán đồng bộ kinh điển

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

Tóm tắt lại nội dung buổi học

- Biết được các giải pháp đồng bộ tiến trình theo kiểu "Sleep & Wake up" bao gồm:
  - Semaphore
  - Critical Region
  - Monitor
- Áp dụng các giải pháp này vào các bài toán đồng bộ kinh điển

Bài tập 1

- Xét giải pháp phần mềm do Dekker đề nghị để tổ chức truy xuất độc quyền cho 2 tiến trình. Hai tiến trình P0 và 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 thỏa 3 yêu cầu trong việc giải quyết tranh chấp không?

Bài tập 2

- Xét giải pháp đồng bộ hóa sau:

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

Giải pháp này có thỏa yêu cầu độc quyền truy xuất không?

Bài tập 3

- 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 4

- Xét hai tiến trình sau:

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

a. Đồng bộ hóa xử lý của 2 tiến trình trên, sử dụng 2 semaphore tổng quát, sao cho tại bất kỳ thời điểm nào cũng có nb <= na <= nb +10

b. Nếu giảm điều kiện chỉ có là na <= nb +10, giải pháp của bạn sẽ được sửa chữa như thế nào?

c. Giải pháp của bạn có còn đúng nếu có nhiều tiến trình loại A và B cùng thực hiện?

Bài tập 5

- Một biến X được chia sẻ bởi 2 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 đảm bảo X không vượt quá 20?

Bài tập 6

- Xét 2 tiến trình xử lý đoạn chương trình sau:

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

Đồng bộ hóa hoạt động của 2 tiến trình này sao cho cả A1 và B1 đều hoàn tất trước khi A2 và B2 bắt đầu

Bài tập 7

- Tổng quát hóa câu hỏi 6 cho các tiến trình có đoạn chương trình sau:

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

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

Đồng bộ hóa hoạt động của 2 tiến trình này sao cho với *k* bất kỳ (2<=k<=100), A_k chỉ có thể bắt đầu khi $B_{(k-1)}$ đã kết thúc và B_k chỉ có thể bắt đầu khi $A_{(k-1)}$ đã kết thúc.

Bài tập 8

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

```
w := x1 * x2
v := x3 * x4
y := v * x5
z := v * x6
x := w * y
z := w * z
ans := y + z
```

Bài kiểm tra 30 phút

Bài 1: Xét giải pháp đồng bộ hóa sau có bảo đảm 3 điều kiện không?

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

Bài 2: 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 = B*x5 + C;