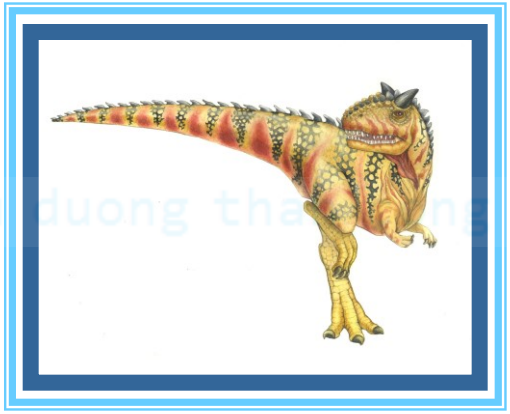
Chương 5: Đồng bộ - 3



UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Mục tiêu

- 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

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Nội dung

- Các giải pháp “Sleep & Wake up”
  - Semaphore
  - Các bài toán đồng bộ kinh điển
  - Monitor

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);
```


UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Semaphore

- Một trong những công cụ đảm bảo sự đồng bộ của các process mà hệ điều hành cung cấp là **Semaphore**.
- Ý tưởng của Semaphore:
  - ✓ Semaphore S là một biến số nguyên.
  - ✓ Ngoài thao tác khởi động biến thì Semaphore chỉ có thể được truy xuất qua hai hàm có tính đơn nguyên (atomic) là **wait** và **signal**

Hàm wait() và signal() còn có tên gọi khác lần lượt là P() và V()

Semaphore

Đầu tiên, hàm wait và signal được hiện thực như sau:

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
signal(S) {
    S++;
}
```

Tuy nhiên, với cách hiện thực này, vòng lặp "while (S <= 0);" trong hàm wait() sẽ dẫn tới busy waiting.

➔ *Để không busy waiting, hàm wait() và signal() được cải tiến. Một hàng đợi semaphore được đưa thêm vào để thay vì phải lặp vòng trong trường hợp semaphore nhỏ hơn hoặc 0, process sẽ được đưa vào hàng đợi này để chờ và sẽ được ra khỏi hàng đợi này khi hàm signal() được gọi.*

Semaphore

- **Hàm wait và signal của Semaphore cải tiến, không busy waiting như sau:**

✓ Định nghĩa semaphore là một record

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

✓ Mỗi Semaphore có một giá trị nguyên của nó và một danh sách các process.

✓ Khi các process chưa sẵn sàng để thực thi thì sẽ được đưa vào danh sách này. Danh sách này còn gọi là hàng đợi semaphore.

Lưu ý: Các process khi ra khỏi hàng đợi semaphore sẽ vào hàng đợi Ready để chờ lấy CPU thực thi.

Semaphore

- **Hàm wait và signal của Semaphore cải tiến, không busy waiting như sau:**

✓ Hàm wait() và signal() 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);
        }
}
```

Lập trình thực tế, tùy từng ngôn ngữ, có thể là:
S.value hoặc
S→value

Semaphore

- **Hàm wait và signal của Semaphore cải tiến, không busy waiting như sau:**

✓ Khi hàm wait() được gọi, ngay lập tức giá trị value của Semaphore S bị giảm đi 1. Và nếu giá trị Semaphore S âm, process này sẽ bị đưa vào danh sách L (đưa vào hàng đợi Semaphore) và bị khóa (block) lại.

✓ Khi hàm signal() được gọi, ngay lập tức giá trị value của Semaphore S tăng lên 1. Và nếu giá trị Semaphore lớn hơn hoặc bằng 0, một process sẽ được chọn lựa trong danh sách L, tức trong hàng đợi Semaphore và bị đánh thức dậy (wakeup) để ra hàng đợi ready và chờ CPU để thực hiện.

Lưu ý: Trong hiện thực, các PCB của các process bị block sẽ được đưa vào danh sách L. Danh sách L có thể dùng hàng đợi FIFO hoặc cấu trúc khác tùy thuộc vào yêu cầu hệ thống.

Semaphore

- **Hàm wait và signal của Semaphore cải tiến, không busy waiting như sau:**

✓ Block() và Wakeup() là hai system calls của hệ điều hành.

- block(): Process này đang thực thi lệnh này sẽ bị khóa lại.

  Process chuyển từ Running sang Waiting

- wakeup(P): hồi phục quá trình thực thi của process P đang bị blocked

  Process P chuyển thừ Waiting sang Ready

Ví dụ sử dụng semaphore (1)

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Ví dụ 1:

Dùng semaphore giải quyết n process truy xuất vào CS.

Có hai trường hợp:

- ✓ Chỉ duy nhất một process được vào CS (mutual exclusion)

  Khởi tạo S.value = 1

- ✓ Cho phép k process vào CS

  Khởi tạo S.value = k

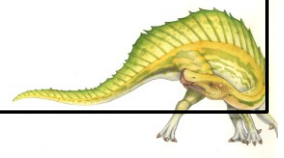
- Shared data:

  semaphore mutex;

  (Khởi tạo mutex.value = 1)

- Process *Pi:*

```
do {
        wait(mutex);
        critical section
        signal(mutex);
        remainder section
} while (1);
```



Ví dụ sử dụng semaphore (2)

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Ví dụ 2:

Dùng Semaphore giải quyết đồng bộ giữa hai process

Yêu cầu:

Cho hai process: P1 và P2

P1 thực hiện lệnh S1 và P2 thực hiện lệnh S2.

Lệnh S2 chỉ được thực thi sau khi lệnh S1 được thực thi.

Khởi tao semaphore tên synch và

synch.value = 0

- Process P1:

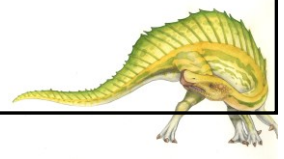
  S1;

  signal(synch);

- Process P2:

  wait(synch);

  S2;



Ví dụ sử dụng semaphore (3)

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Ví dụ 3:

Dùng Semaphore giải quyết đồng bộ giữa hai process

Yêu cầu:

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 2 Semaphore s1 và s2

s1.value = s2.value = 0

- Process P1:

  A1;

  signal(s1);,

  wait(s2);

  A2;

- Process P2:

  B1

  signal(s2);

  wait(s1);

  B2;

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Nhận xét

- Khi S.value ≥ 0: value chính là số process có thể thực thi wait(S) mà không bị blocked
- Khi S.value < 0: trị tuyệt đối của value chính là số process đang đợi trên hàng đợi semaphore

Nhận xét (tt)

- Việc hiện thực Semaphore phải đảm bảo tính chất Atomic và mutual exclusion: tức 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

- 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 hiệu quả trên hệ thống multiprocessor.
  - Multiprocessor: có thể dùng các giải pháp software (như giải Peterson và Bakery) hoặc giải pháp hardware (TestAndSet, Swap).
- Vùng tranh chấp của các tác vụ wait(S) và signal(S) thông thường rất nhỏ: khoảng 10 lệnh.
  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.

Ví dụ thường gặp nhất của deadlock là hai (hoặc nhiều) process đang chờ đợi qua lại các sự kiện của nhau thì mới được thực thi, nhưng cả hai process này đều đã bị block, nên sự kiện này không bao giờ xảy ra và hai process sẽ bị block vĩnh viễn.

Ví dụ: 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);**

Ví dụ khởi tạo S.value và Q.value bằng 1. P0 đầu tiên thực thi wait(S), rồi P1 thực thi wait(Q), rồi P0 thực thi wait(Q) và bị blocked, tiếp theo P1 thực thi wait(S) bị blocked.

➔ Tình huống nà là P0 và P1 bị rơi vào deadlock.

Deadlock và starvation

- **Starvation** (indefinite blocking): Trường hợp một tiến trình có thể không bao giờ được lấy ra khỏi hàng đợi mà nó bị khóa/treo (block) trong hàng đợi đó.

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

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.

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Các bài toán đồng bộ kinh điển

Ba bài toán đồng bộ kinh điển:

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

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Bài toán Bounded-Buffer

Producer sản suất một sản phẩm và đặt vào buffers, buffers giới hạn chỉ chứa được n sản phẩm.

Consumer tiêu thụ mỗi lần một sản phẩm, sản phẩm được lấy ra từ buffers.

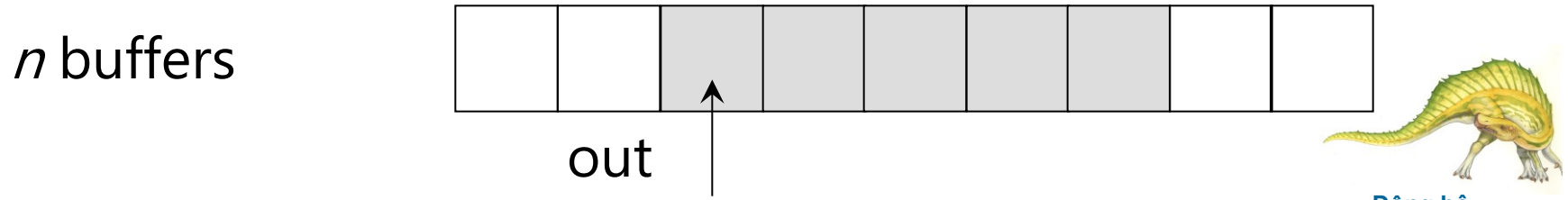
Khi buffers đã chứa n sản phẩm, Producer không thể đưa tiếp sản phẩm vào buffers nữa mà phải chờ đến khi buffers có chỗ trống. Khi buffers rỗng, Consumer không thể lấy sản phẩm để tiêu thụ mà phải chờ đến khi có ít nhất 1 sản phẩm vào buffers.

Bài toán Bounded-Buffer

Để hiện thực bài toán trên, các biên chia sẻ giữa Producer và Consumer như sau:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

- Buffers có n chỗ (n buffer con/vị trí) để chứa sản phẩm
- Biến semaphore mutex cung cấp khả năng mutual exclusion cho việc truy xuất tới buffers. Biến mutex được khởi tạo bằng 1 (tức value của mutex bằng 1).
- Biến semaphore empty và full đếm số buffer rỗng và đầy trong buffers.
- Lúc đầu, toàn bộ buffers chưa có sản phẩm nào được đưa vào: value của empty được khởi tạo bằng n; và value của full được khởi tạo bằng 0



Bài toán bounder buffer

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

producer

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

consumer

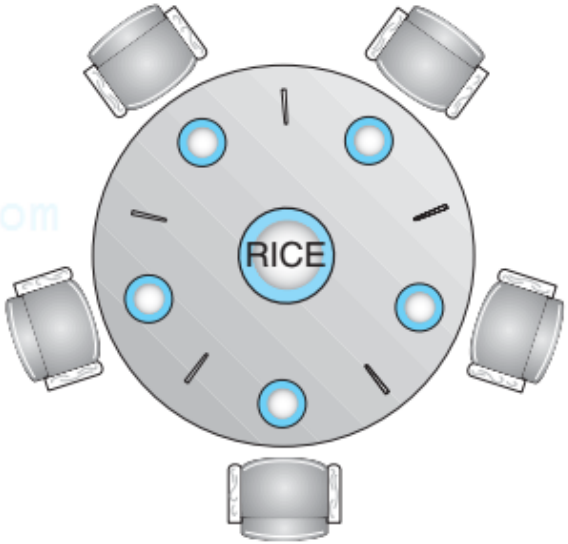
```
do {
        wait(full)
        wait(mutex);
        ...
        nextc = get_buffer_item(out);
        ...
        signal(mutex);
        signal(empty);
        ...
        consume_item(nextc);
        ...
} while (1);
```

Đồng bộ

Bài toán "Dining Philosophers"

- **Bài toán 5 triết gia ăn tối:**

5 triết gia ngồi vào bàn tròn với một đĩa thức ăn ở giữa và chỉ với 5 chiếc đũa đơn được đặt như hình. Khi một triết gia suy nghĩ, sẽ không tương tác với các triết gia khác. Sau một khoảng thời gian, khi triết gia đói, sẽ phải cần lấy 2 chiếc đũa gần nhất để ăn. Tại một thời điểm, triết gia chỉ có thấy lấy 1 chiếc đũa (không thể lấy đũa mà triết gia khác đã cầm). Khi triết gia có 2 chiếc đũa, sẽ lập tức ăn và chỉ bỏ 2 đũa xuống khi nào ăn xong. Sau đó triết gia lại tiếp tục suy nghĩ.



- Đây là một bài toán kinh điển trong việc minh họa sự khó khăn trong việc phân phối tài nguyên giữa các process sao cho không xảy ra deadlock và starvation

Bài toán "Dining Philosophers" (tt)

- Dữ liệu chia sẻ:

  Semaphore chopstick[5]

  Khởi tạo các biến đều là 1

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

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);
```

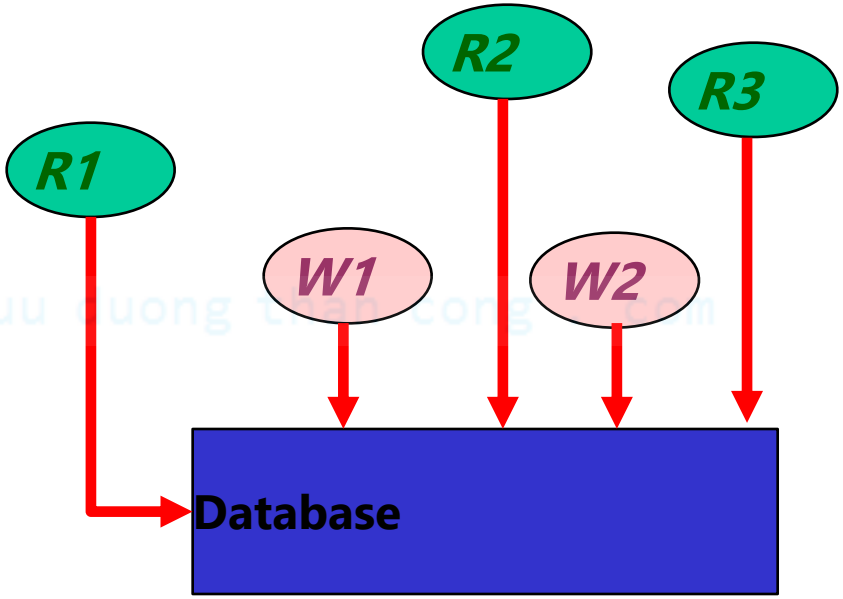

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

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)
  - Đánh số các triết gia từ 0 tới 4. 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

- Có một database hoặc file, nhiều Readers (để đọc) và nhiều Writers (để ghi) dữ liệu vào database.
- Khi một Writer đang truy cập database/file thì không một quá trình nào khác được truy cập.
- Nhiều Readers có thể cùng lúc đọc database/file



Bài toán Reader-Writers (tt)

UIT
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

- 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);
```

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

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)
```

Các trường hợp sử dụng Semaphore có thể gây lỗi

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

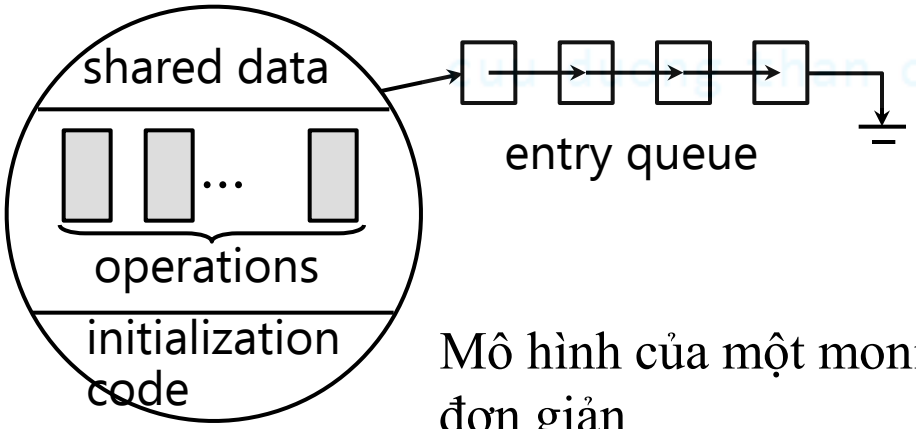
Monitor

- 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)

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- 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

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Cấu trúc của monitor

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


UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

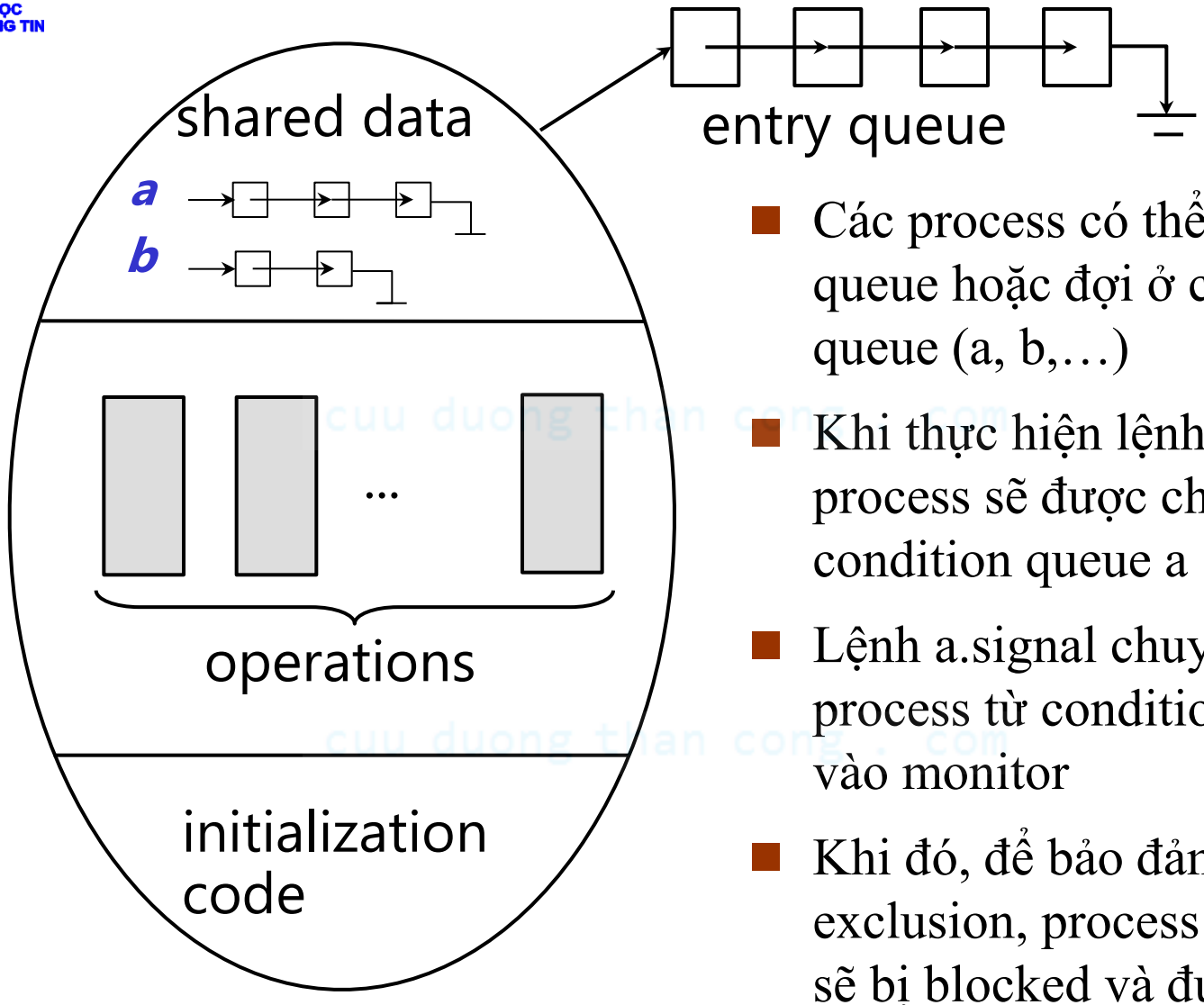
Condition variable

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

  condition a, b;

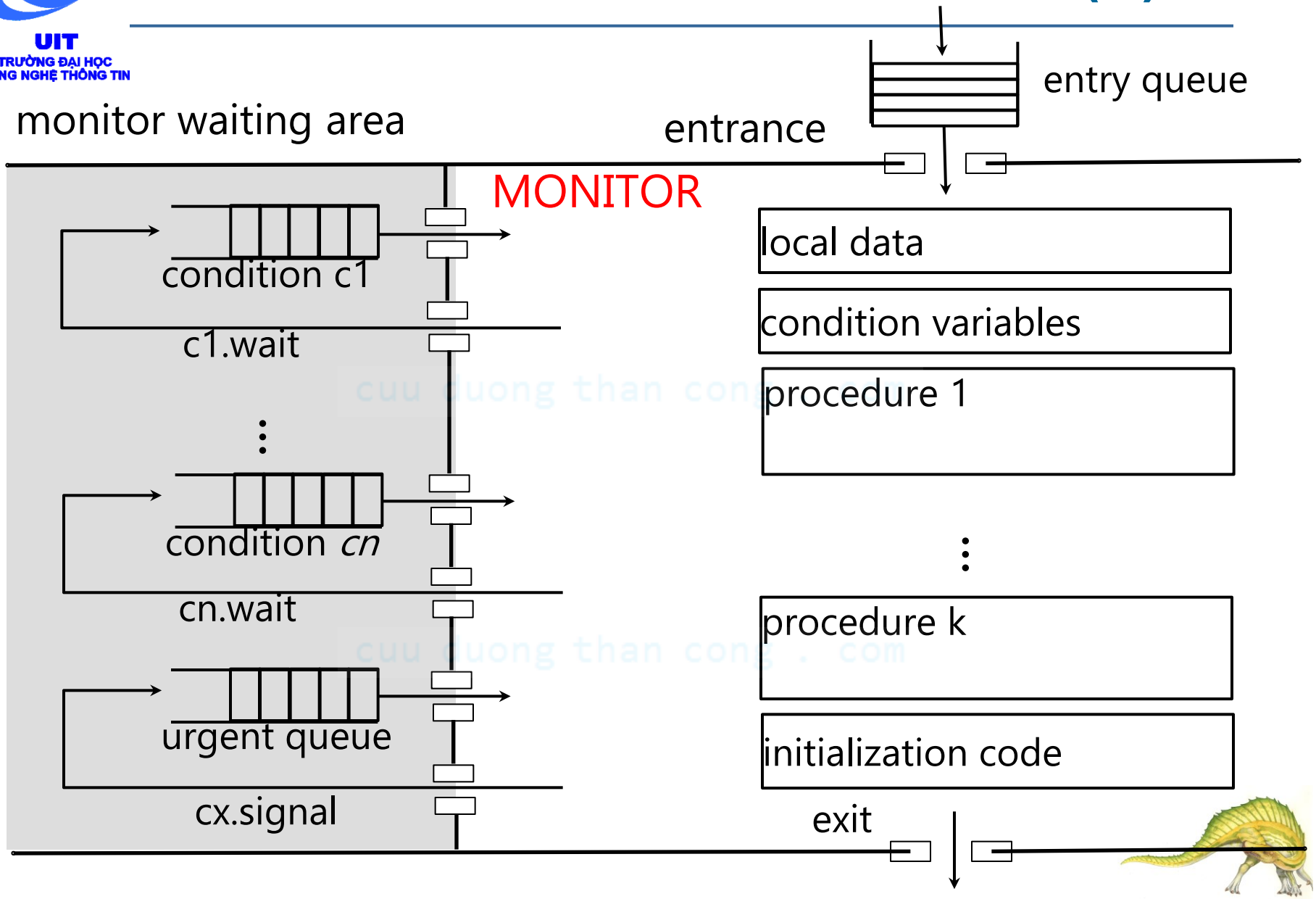
- Các biến điều kiện đều cục bộ và chỉ được truy cập bên trong monitor.
- Chỉ có thể thao tác lên biến điều kiện bằng hai thủ tục:
  - a.wait: process gọi tác vụ này sẽ bị "block trên biến điều kiện" a
    - process này chỉ có thể tiếp tục thực thi khi có process khác thực hiện tác vụ a.signal
  - a.signal: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

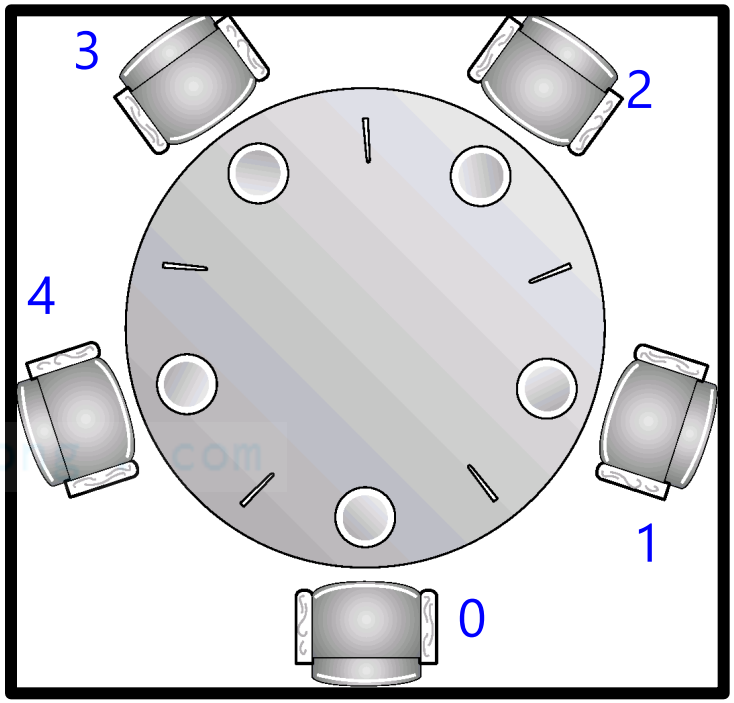


Monitor và dining philosophers

Bài toán Dining Philosophers giải theo dùng Monitor:

Để tránh deadlock, bài toán đưa thêm ràng buộc:

Một triết gia chỉ có thể lấy đôi đũa để ăn trong trường hợp 2 chiếc đũa hai bên đều đang sẵn sàng.



```
monitor DiningPhilosophers
{
   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((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;
    }
}
```

Cấu trúc một Monitor cho bài toán Dining Philosophers

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Dining philosophers (tt)

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


Dining philosophers (tt)

```
void pickup(int i) {
        state[ i ] = hungry;
        test[ i ];      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.

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Câu hỏi ôn tập và bài tập

- Semaphore là gì? Nêu cách hoạt động của semaphore và ứng dụng vào một bài toán đồng bộ?
- Monitor là gì? Nêu cách hoạt động của monitor và ứng dụng vào một bài toán đồng bộ?
- Bài tập về nhà: các bài tập chương 5 trên moodle