



HỆ ĐIỀU HÀNH

CHƯƠNG 5: ĐỒNG BỘ TIẾN TRÌNH (PHẦN 3)

Trong 02 phần trước, ta đã tìm hiểu về những kỹ thuật và công cụ để giải quyết vấn đề vùng tranh chấp giải thuật Peterson, semaphore, mutex, monitor. Trong nội dung cuối cùng của Chương 5, ta sẽ thảo luận việc áp dụng các kỹ thuật trên vào các bài toán đồng bộ kinh điển như thế nào và những vấn đề có thể phát sinh khi thực hiện.



MỤC TIÊU

1. Giải thích được bài toán đồng bộ bounded-buffer
2. Giải thích được bài toán đồng bộ readers-writers
3. Giải thích được bài toán đồng bộ dining-philosophers
4. Phân tích các vấn đề thường gặp khi thực hiện đồng bộ tiến trình/tiểu trình



NỘI DUNG

9. Bài toán đồng bộ bounded-buffer
10. Bài toán đồng bộ readers-writers
11. Bài toán đồng bộ dining-philosophers



BÀI TOÁN ĐỒNG BỘ BOUNDED-BUFFER

5.9.1. Phát biểu bài toán bounded-buffer

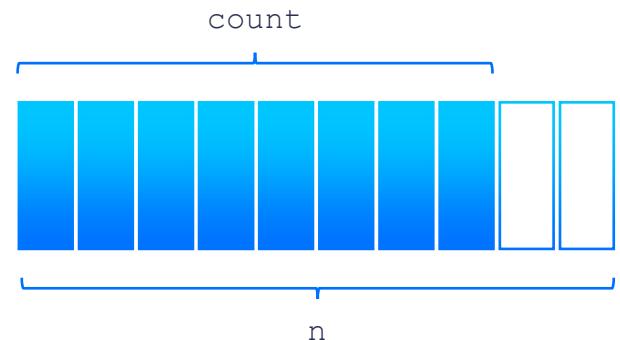
Bài toán bounded-buffer (đã được đề cập trong phần 5.1) mô tả một mảng có giới hạn số phần tử và 02 tiến trình lùn lượt là Producer và Consumer có nhiệm vụ đồng thời ghi và xóa phần tử ra khỏi mảng.

09.



5.9.1. Phát biểu bài toán bounded-buffer

- Một mảng buffer[] có tối đa n phần tử
Số lượng phần tử trong mảng là count



- Tiến trình **Producer** thêm phần tử vào mảng
- Tiến trình **Consumer** xóa phần tử khỏi mảng





5.9.1. Phát biểu bài toán bounded-buffer

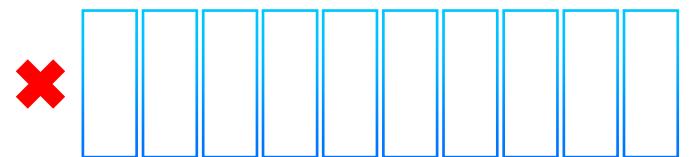
- Tiến trình **Producer** thêm phần tử vào mảng
→ Không thể thêm khi mảng đã đầy

count = n



Không thể thêm tiếp

count = 0



Không thể xóa tiếp

- Tiến trình **Consumer** xóa phần tử khỏi mảng
→ Không thể xóa khi mảng đang rỗng



BÀI TOÁN ĐỒNG BỘ BOUNDED-BUFFER

5.9.2. Giải pháp cho bài toán bounded-buffer

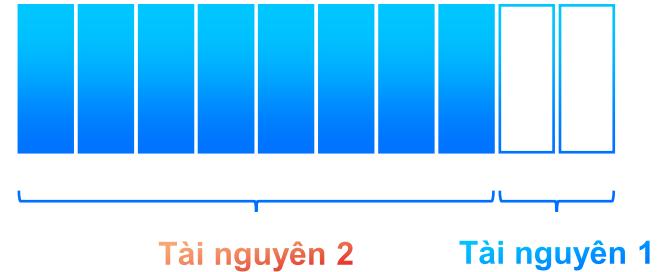
Để giải quyết bài toán bounded-buffer, ta cần xác định đúng điều kiện và áp dụng semaphore để đồng bộ cho các điều kiện này.

09.



5.9.2. Giải pháp cho bài toán bounded-buffer

- Bước 1: Dựa vào điều kiện, xác định tài nguyên:
 - Không thể thêm khi mảng đã đầy
→ Tài nguyên 1: **số vị trí có thể thêm**
 - Không thể xóa khi mảng đang rỗng
→ Tài nguyên 2: **số vị trí có thể xóa**
 - Vùng tranh chấp: mảng buffer và count
→ Bảo vệ vùng tranh chấp → mutual exclusion



Producer

```
► // add to buffer[]  
count++;
```

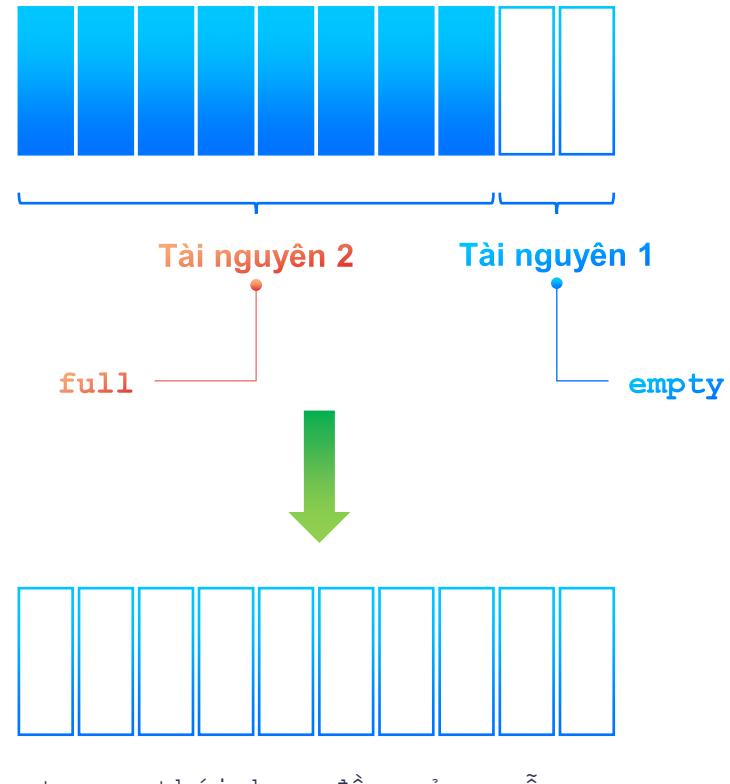
Consumer

```
► // remove from buffer[]  
count--;
```



5.9.2. Giải pháp cho bài toán bounded-buffer

- Bước 2: Xác định số lượng semaphore
 - Tài nguyên 1: số vị trí có thể thêm
→ **semaphore empty**: khởi tạo là **n**
 - Tài nguyên 2: số vị trí có thể xóa
→ **semaphore full**: khởi tạo là **0**
 - Bảo vệ vùng tranh chấp → mutual exclusion
→ **semaphore mutex**: khởi tạo là **1**





5.9.2. Giải pháp cho bài toán bounded-buffer

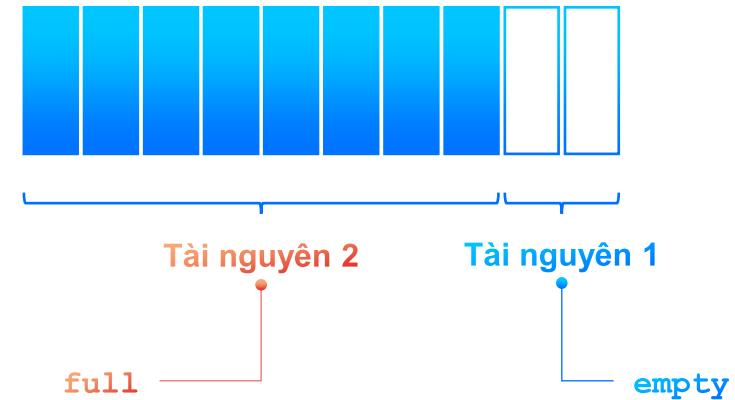
- Bước 3: Đặt wait () và signal ()

Producer

```
▶ wait(empty);  
wait(mutex);  
// add to buffer[]  
count++;  
signal(mutex);  
signal(full);
```

Consumer

```
▶ wait(full);  
wait(mutex);  
// remove from buffer[]  
count--;  
signal(mutex);  
signal(empty);
```





BÀI TOÁN ĐỒNG BỘ BOUNDED-BUFFER

5.9.3. Các lỗi thường gặp

Một vấn đề thường thấy khi chưa hiểu rõ các công cụ đồng bộ đó là người lập trình thường cố gắng sử dụng while hoặc if để đồng bộ. Một số khác khi phân tích bài toán lại bỏ quên vùng tranh chấp dẫn đến việc đồng bộ không đảm bảo loại trừ tương hỗ.

09.



5.9.3. Các lỗi thường gặp

Sử dụng hàm while hoặc if

Producer

```
▶ wait(empty);  
wait(mutex);  
// add to buffer[]  
count++;  
signal(mutex);  
signal(full);
```

Producer

```
while (count < n){  
    wait(mutex);  
    // add to buffer[]  
    count++;  
    signal(mutex);  
}
```

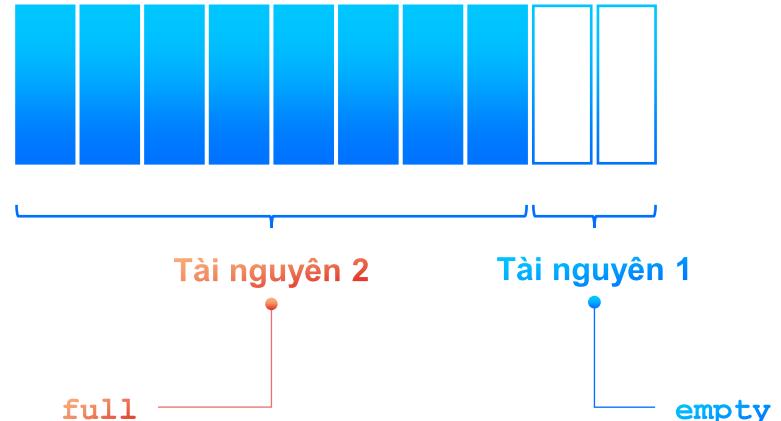
BUSY WAITING

Consumer

```
while (count > 0){  
    wait(mutex);  
    // remove from buffer[]  
    count--;  
    signal(mutex);  
}
```

Consumer

```
▶ wait(full);  
wait(mutex);  
// remove from buffer[]  
count--;  
signal(mutex);  
signal(empty);
```





5.9.3. Các lỗi thường gặp

Bỏ qua vùng tranh chấp

Producer

```
▶ wait(empty);  
wait(mutex);  
// add to buffer[]  
count++;  
signal(mutex);  
signal(full);
```

Producer

```
▶ wait(empty);  
// add to buffer[]  
count++;  
signal(full);
```

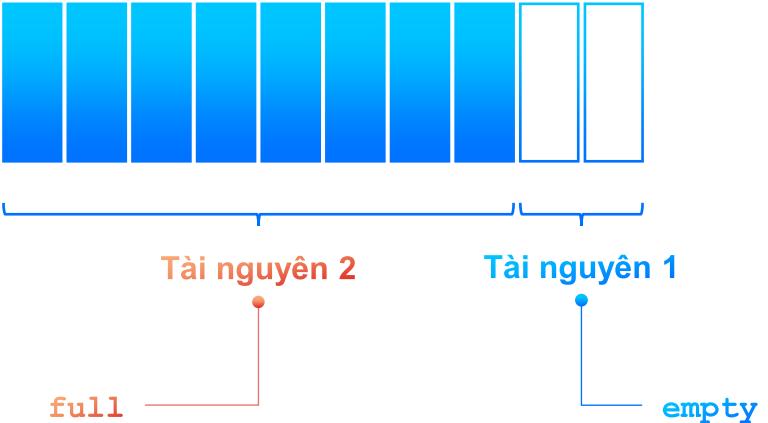
Consumer

```
▶ wait(full);  
// remove from buffer[]  
count--;  
signal(empty);
```

Consumer

```
▶ wait(full);  
wait(mutex);  
// remove from buffer[]  
count--;  
signal(mutex);  
signal(empty);
```

DỮ LIỆU
KHÔNG NHẤT QUÁN





BÀI TOÁN ĐỒNG BỘ READERS-WRITERS

5.10.1. Phát biểu bài toán Readers-Writers

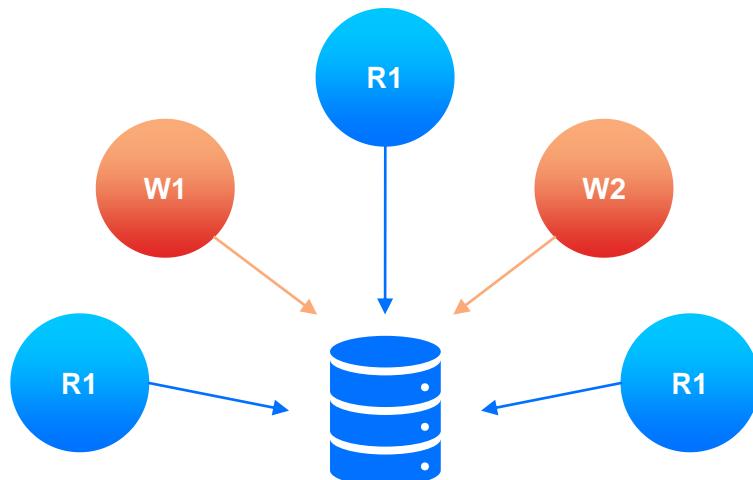
Giả sử tồn tại một cơ sở dữ liệu được chia sẻ giữa những tiến trình đang thực thi đồng thời. Một số tiến trình chỉ muốn đọc dữ liệu trong khi một số khác muốn cập nhật (vừa đọc và ghi) vào cơ sở dữ liệu này. Các tiến trình chỉ muốn đọc dữ liệu được gọi là **Readers** và các tiến trình muốn cập nhật được gọi là **Writers**.

10.



5.10.1. Phát biểu bài toán Readers-Writers

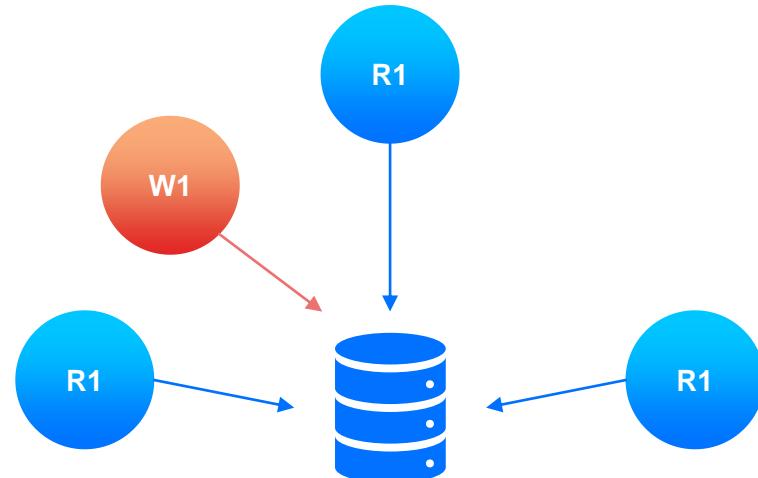
- Dữ liệu được chia sẻ giữa các tiến trình đang thực thi đồng thời
 - **Readers:** Chỉ đọc dữ liệu; không thực hiện cập nhật
 - **Writers:** Có thể vừa đọc vừa ghi dữ liệu





5.10.1. Phát biểu bài toán Readers-Writers

- Dữ liệu được chia sẻ giữa các tiến trình đang thực thi đồng thời
 - **Readers:** Chỉ đọc dữ liệu; không thực hiện cập nhật
 - **Writers:** Có thể vừa đọc vừa ghi dữ liệu
- Vấn đề:
 - Cho phép nhiều **Readers** cùng đọc dữ liệu đồng thời
 - Chỉ một **Writers** được phép *truy cập* dữ liệu tại một thời điểm





5.10.1. Phát biểu bài toán Readers-Writers

Các biến thể của bài toán Readers - Writers

Biến thể 1

(The First Readers – Writers Problem)

- **Ưu tiên Readers.**
 - Các Readers có thể đọc đồng thời cùng nhau.
 - Khi một Readers đang đọc, không có Readers nào phải chờ chỉ vì có một Writers đang chờ trước nó.
- Writers có thể bị starvation

Biến thể 2

(The Second Readers – Writers Problem)

- **Ưu tiên Writers.**
 - Khi một Writers đã sẵn sàng thì Writers này sẽ được thực thi càng sớm càng tốt.
 - Nếu một Writers đang chờ để truy cập dữ liệu, không có một Readers mới nào được phép thực thi.
- Readers có thể bị starvation



BÀI TOÁN ĐỒNG BỘ READERS-WRITERS

5.10.2. Giải pháp cho bài toán Readers-Writers

Việc 2 Readers cùng truy cập và đọc dữ liệu được chia sẻ không gây ra vấn đề gì đến tính nhất quán của dữ liệu và tính đúng đắn của chương trình. Tuy nhiên, khi có một Writers cùng với một vài tiến trình khác (bất kể Readers hay Writers khác) truy cập đồng thời vào dữ liệu được chia sẻ thì dữ liệu sẽ bị ảnh hưởng. Giải pháp cho bài toán Readers – Writers cần phải dựa vào độ ưu tiên giữa Readers và Writers, đồng thời đảm bảo được tính loại trừ tương hỗ khi một Writer truy cập dữ liệu.

10.



5.10.2. Giải pháp cho bài toán Readers-Writers

Biến thể 1 – Ưu tiên Readers

Reader
► // đọc dữ liệu

Writer
► // cập nhật dữ liệu

- Các dữ liệu được chia sẻ:
 - Cơ sở dữ liệu/Biến/Mảng/...
 - **semaphore rw_mutex**: khởi tạo là **1** // bảo vệ **cơ sở dữ liệu**
 - **semaphore mutex**: khởi tạo là **1** // bảo vệ **read_count**
 - Biến **read_count**: khởi tạo là **0** // đếm số Readers // được chia sẻ giữa các Readers



5.10.2. Giải pháp cho bài toán Readers-Writers

Biến thể 1 – Ưu tiên Readers

Reader

```
▶ wait(mutex);  
read_count++;  
if (read_count == 1) /* first reader */  
    wait(rw_mutex);  
signal(mutex);  
...  
/* đọc dữ liệu */  
...  
wait(mutex);  
read_count--;  
if (read_count == 0) /* last reader */  
    signal(rw_mutex);  
signal(mutex);
```

Writer

```
▶ wait(rw_mutex);  
...  
/* cập nhật dữ liệu */  
...  
signal(rw_mutex);
```

- Nếu một **Writer** đang ở trong CS và có n **Readers** đang đợi thì một **Reader** được xếp trong hàng đợi của **rw_mutex** và n - 1 **Reader** kia trong hàng đợi của **mutex**.
- Khi **Writer** thực thi **signal(rw_mutex)**, hệ thống có thể phục hồi thực thi của một trong các **Reader** đang đợi hoặc **Writer** đang đợi.



BÀI TOÁN ĐỒNG BỘ DINING-PHILOSOPHER

5.11.1. Phát biểu bài toán Dining-Philosopher

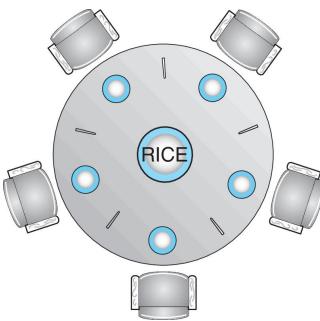
Bài toán Dining-Philosopher hay còn gọi là Bài toán Các Triết gia ăn tối được xem là bài toán đồng bộ kinh điển không phải bởi vì tính quan trọng trong thực hành hay bởi các nhà khoa học máy tính không thích các Triết gia mà bởi vì đây là một ví dụ cho các vấn đề đồng bộ trên quy mô lớn. Bài toán này cho thấy sự khó khăn khi phải cấp phát các tài nguyên giữa các tiến trình sao cho không bị deadlock và starvation.

11.



5.11.1. Phát biểu bài toán Dining-Philosopher

- Có n Triết gia ngồi trên một chiếc bàn tròn với tô cơm ở giữa

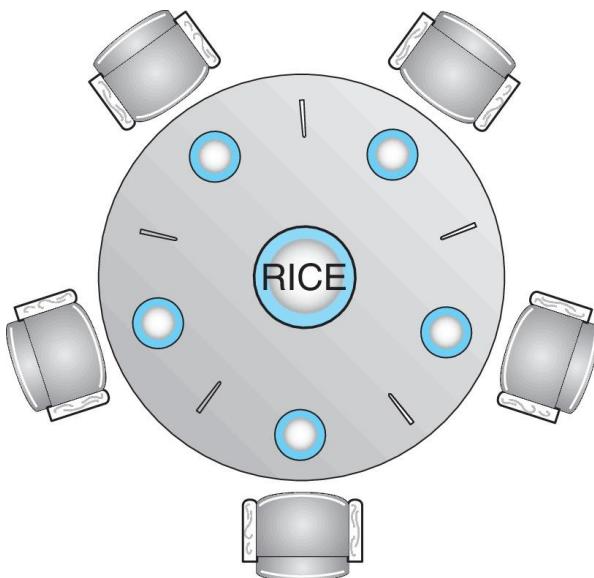


- Họ dành thời gian chỉ để làm 2 công việc: suy nghĩ và ăn
- Họ không tương tác với những người bên cạnh
- Thi thoảng các Triết gia sẽ cầm lên 2 chiếc đũa (mỗi chiếc 1 lần) để ăn
 - Cần phải có đủ 2 chiếc đũa thì mới ăn được, sau khi ăn xong thì trả lại 2 chiếc đũa
 - Số chiếc đũa cũng là n



5.11.1. Phát biểu bài toán Dining-Philosopher

- Trường hợp có 5 Triết gia, dữ liệu được chia sẻ như sau:
 - Tô cơm (dữ liệu)
 - Mảng **semaphore chopstick[5]**: tất cả khởi tạo là 1 // 5 chiếc đũa





BÀI TOÁN ĐỒNG BỘ DINING-PHILOSOPHER

5.11.2. Giải pháp cho bài toán Dining-Philosopher

Bài toán Dining-Philosopher hay còn gọi là Bài toán Các Triết gia ăn tối được xem là bài toán đồng bộ kinh điển không phải bởi vì tính quan trọng trong thực hành hay bởi các nhà khoa học máy tính không thích các Triết gia mà bởi vì đây là một ví dụ cho các vấn đề đồng bộ trên quy mô lớn. Bài toán này cho thấy sự khó khăn khi phải cấp phát các tài nguyên giữa các tiến trình sao cho không bị deadlock và starvation.

11.

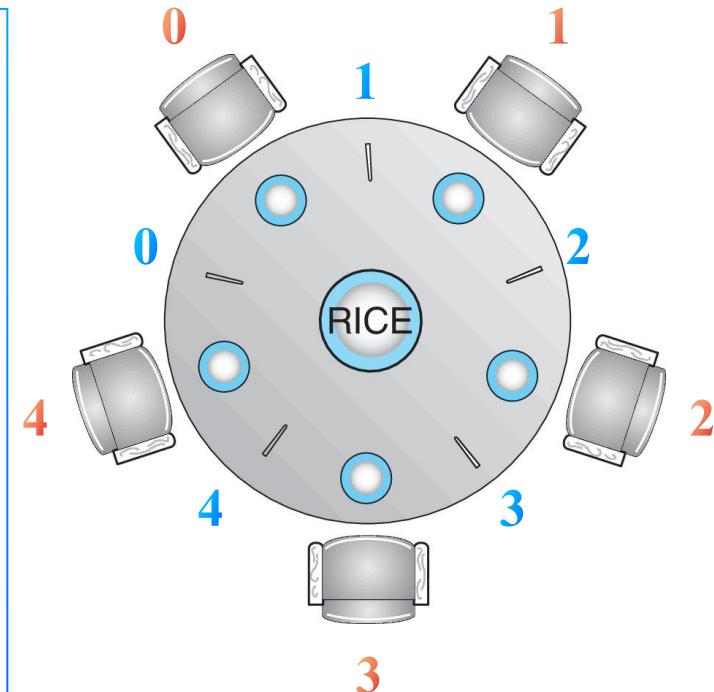


5.11.2. Giải pháp cho bài toán Dining-Philosopher

Giải pháp sử dụng semaphore

Triết gia thứ i:

```
▶ wait (chopstick[i] ); // chờ đũa bên trái  
wait (chopstick[ (i + 1) % 5] ); // chờ đũa bên phải  
...  
/* eat for awhile */  
...  
signal (chopstick[i] ); // trả đũa bên trái  
signal (chopstick[ (i + 1) % 5] ); // trả đũa bên phải  
...  
/* think for awhile */  
...
```





5.11.2. Giải pháp cho bài toán Dining-Philosopher

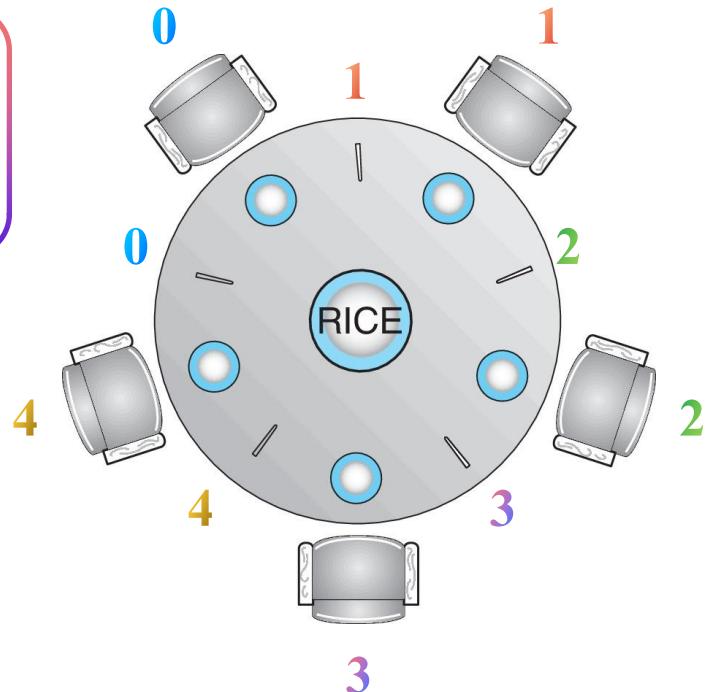
Giải pháp sử dụng semaphore

Triết gia thứ i:

```
▶ wait (chopstick[i] );
  wait (chopstick[ (i + 1) % 5] );
  ...
  /* eat for awhile */
  ...
  signal (chopstick[i] );
  signal (chopstick[ (i + 1) % 5] );
  ...
  /* think for awhile */
  ...
```

Nếu tất cả Triết gia
đồng thời cầm đũa
bên trái

DEADLOCK





5.11.2. Giải pháp cho bài toán Dining-Philosopher

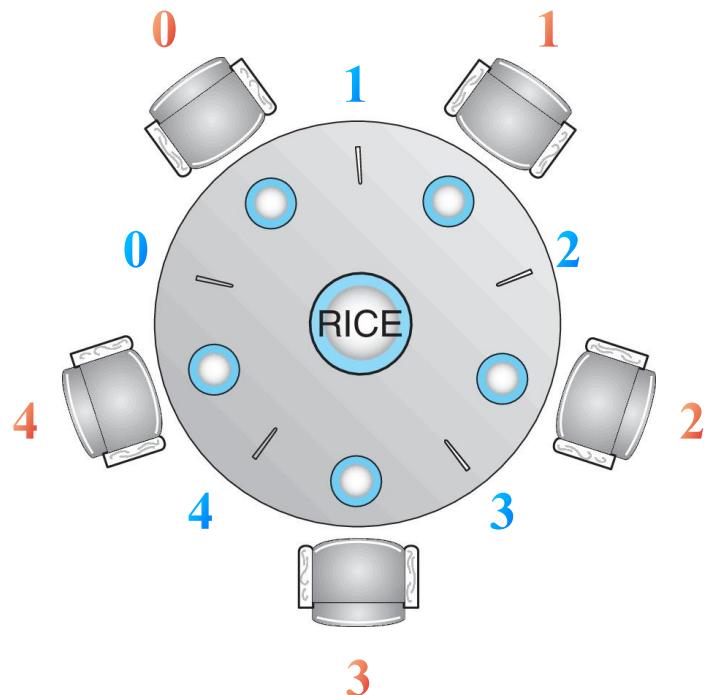
- Một số giải pháp có thể tránh deadlock:
 - Cho phép tối đa 4 Triết gia ngồi vào bàn
 - Chỉ cho phép Triết gia cầm đũa khi cả 2 chiếc đũa đã sẵn sàng → Cần phải thực hiện hành động cầm đũa trong CS
 - Giải pháp bất đối xứng: Triết gia ngồi vị trí lẻ cầm đũa bên trái trước, rồi sau đó cầm đũa bên phải; trong khi Triết gia ngồi vị trí chẵn cầm đũa bên phải trước, rồi sau đó cầm đũa bên trái.
 - Cần phải lưu ý tình trạng Starvation vẫn có thể xảy ra



5.11.2. Giải pháp cho bài toán Dining-Philosopher

Giải pháp sử dụng monitor

- Triết gia chỉ có thể cầm đũa khi cả 2 chiếc đã sẵn sàng
- Khai báo kiểu dữ liệu
 - `enum { THINKING; HUNGRY, EATING } state [5];`
- Triết gia thứ i chỉ có thể đặt trạng thái `state[i] = EATING` khi cả 2 người kế bên đang không ăn
 - `(state[(i+4) %5] != EATING)`
 - `&&`
 - `(state[(i+1) %5] != EATING)`
- Khai báo `condition self[5]`: cho phép Triết gia thứ i tự trì hoãn khi mình bị đói nhưng không thể cầm đũa





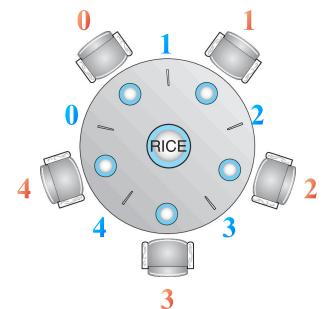
5.11.2. Giải pháp cho bài toán Dining-Philosopher

Giải pháp sử dụng monitor

Khai báo monitor:

```
► 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;
}
```





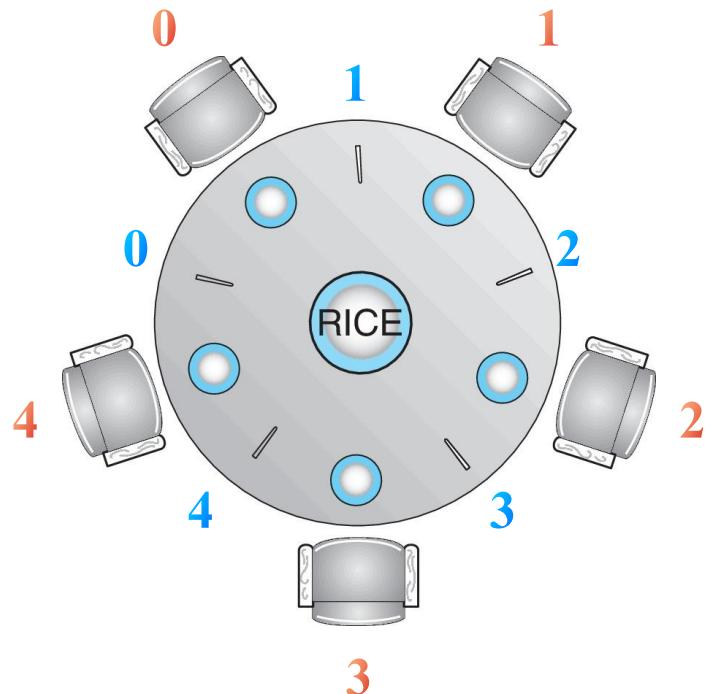
5.11.2. Giải pháp cho bài toán Dining-Philosopher

Giải pháp sử dụng monitor

Triết gia thứ i:

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

- Giải thuật không bao giờ bị deadlock
- Tuy nhiên, vẫn có thể xảy ra starvation





Tóm tắt lại nội dung buổi học

- Bài toán đồng bộ bounded-buffer
- Bài toán đồng bộ readers-writers
- Bài toán đồng bộ dining-philosophers



THẢO LUẬN



Thực hiện bởi Trường Đại học Công nghệ Thông tin, ĐHQG-HCM