



# HỆ ĐIỀU HÀNH

## CHƯƠNG 5: ĐỒNG BỘ TIẾN TRÌNH (PHẦN 2)

Trong phần trước, ta đã tìm hiểu về những kỹ thuật và công cụ để giải quyết vấn đề vùng tranh chấp giải thuật Peterson, mutex lock, Trong nội dung cuối cùng của Chương 5, ta sẽ tiếp tục tìm hiểu các công cụ như semaphore, monitor, đồng thời 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. Diễn tả được cơ chế hoạt động của semaphore và monitor trong việc giải quyết bài toán vùng tranh chấp
2. Phân tích, viết được chương trình sử dụng semaphore để thực hiện đồng bộ thứ tự hoạt động của tiến trình/tiểu trình
3. Trình bày được vấn đề Liveness trong hoạt động đồng bộ tiến trình
4. Giải thích được bài toán đồng bộ bounded-buffer, readers-writers, dining-philosophers
5. Phân tích các vấn đề thường gặp khi thực hiện đồng bộ tiến trình/tiểu trình



# NỘI DUNG

7. Semaphore
8. Monitor
9. Liveness
10. Bài toán đồng bộ bounded-buffer
11. Bài toán đồng bộ readers-writers
12. Bài toán đồng bộ dining-philosophers



# SEMAPHORE

## 5.7.1. Định nghĩa semaphore

Bên cạnh mutex locks, semaphore cũng là một trong những công cụ đồng bộ phổ biến được nhiều hệ điều hành cung cấp. Với semaphore, lập trình viên có thể ứng dụng trong nhiều trường hợp khác nhau bao gồm đồng bộ thứ tự thực thi của các tiến trình/tiểu trình, thứ tự thực thi của các thao tác nằm trên nhiều tiến trình/tiểu trình khác nhau, và đảm bảo loại trừ tương hỗ.

07.



## 5.7.1. Định nghĩa semaphore

- Semaphore là công cụ đồng bộ cung cấp các cách sử dụng linh hoạt (hơn khóa Mutex) để các tiến trình có thể đồng bộ các hoạt động/hành vi của mình.
- Semaphore **S** về bản chất là một **biến số nguyên**
- Chỉ có thể được truy cập thông qua 2 thao tác
  - **wait()** và **signal ()** – hay còn được gọi là P() và V()

Định nghĩa thao tác <b>wait()</b>	Định nghĩa thao tác <b>signal()</b>
<pre>wait(S) {     while (S &lt;= 0)         ; // busy wait     S--; }</pre>	<pre>signal(S) {     S++; }</pre>



## 5.7.1. Định nghĩa semaphore

### Định nghĩa thao tác **wait()**

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

- Nếu semaphore S **không dương** thì tiến trình/tiểu trình phải chờ.
- Khi tiến trình được thực hiện CS thì **trùm semaphore đi 1**.
- Được sử dụng khi muốn **sử dụng tài nguyên**.

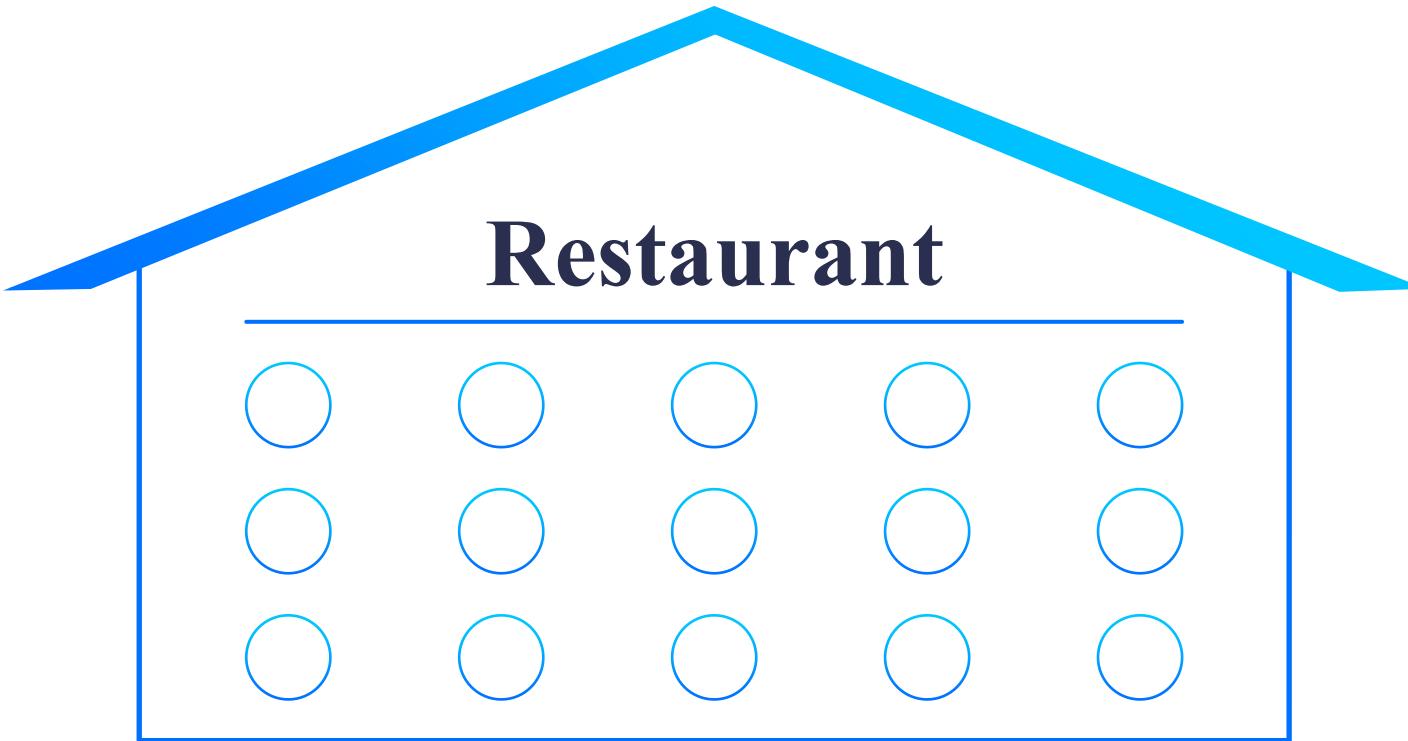
### Định nghĩa thao tác **signal()**

```
signal(S) {  
    S++;  
}
```

- Tăng giá trị của semaphore lên 1.
- Được sử dụng khi **trả lại tài nguyên**.



## 5.7.1. Định nghĩa semaphore

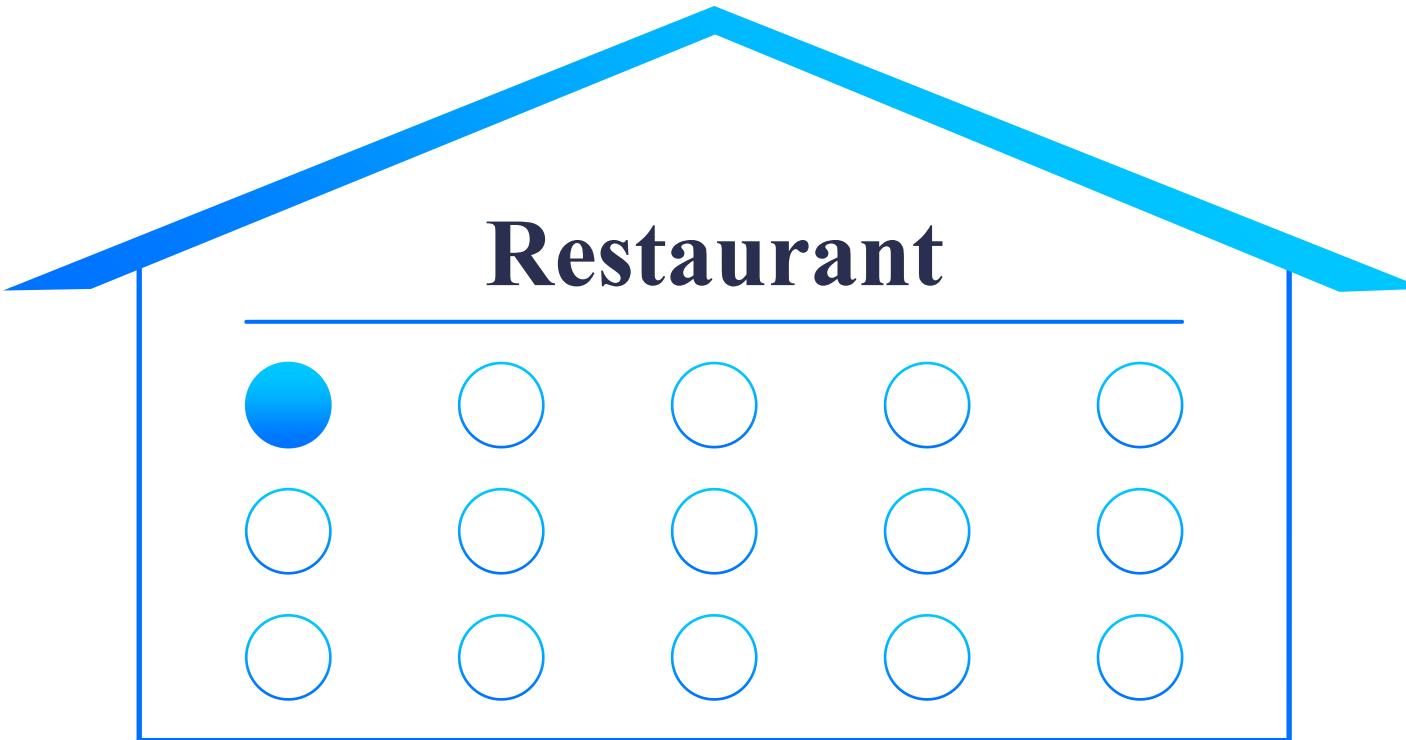


- semaphore `freeTable`
- Khởi tạo `freeTable = 15`

Ví dụ trên nhà hàng



## 5.7.1. Định nghĩa semaphore



Ví dụ trên nhà hàng

- semaphore **freeTable**
- Khởi tạo **freeTable = 15**

**Khách P1 đến**

```
wait (freeTable);  
<dùng bữa>; //freeTable = 14
```



## 5.7.1. Định nghĩa semaphore



Ví dụ trên nhà hàng

- semaphore **freeTable**
- Khởi tạo **freeTable = 15**

**Khách P1 đến**

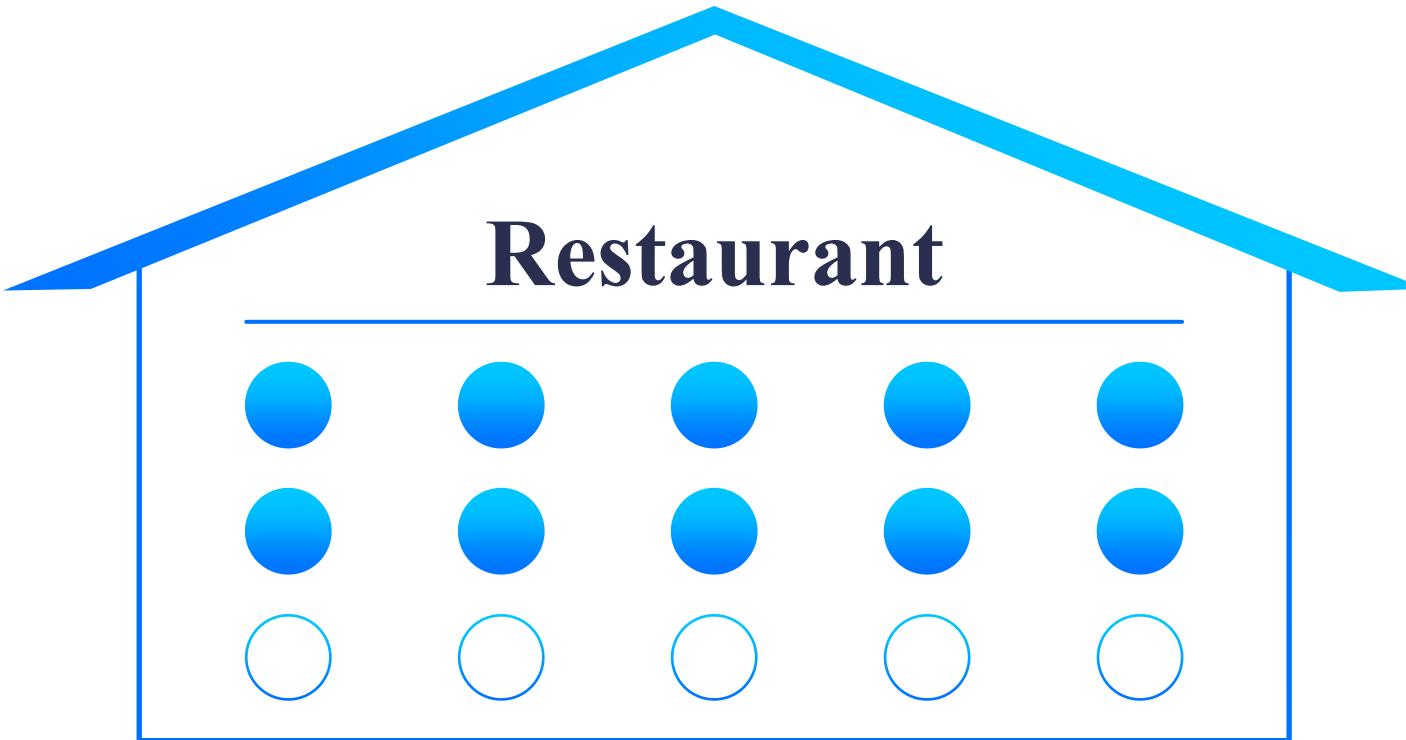
```
wait (freeTable);  
<dùng bữa>; //freeTable = 14
```

**Khách P1 đi**

```
signal (freeTable);  
//freeTable = 15
```



## 5.7.1. Định nghĩa semaphore



Ví dụ trên nhà hàng

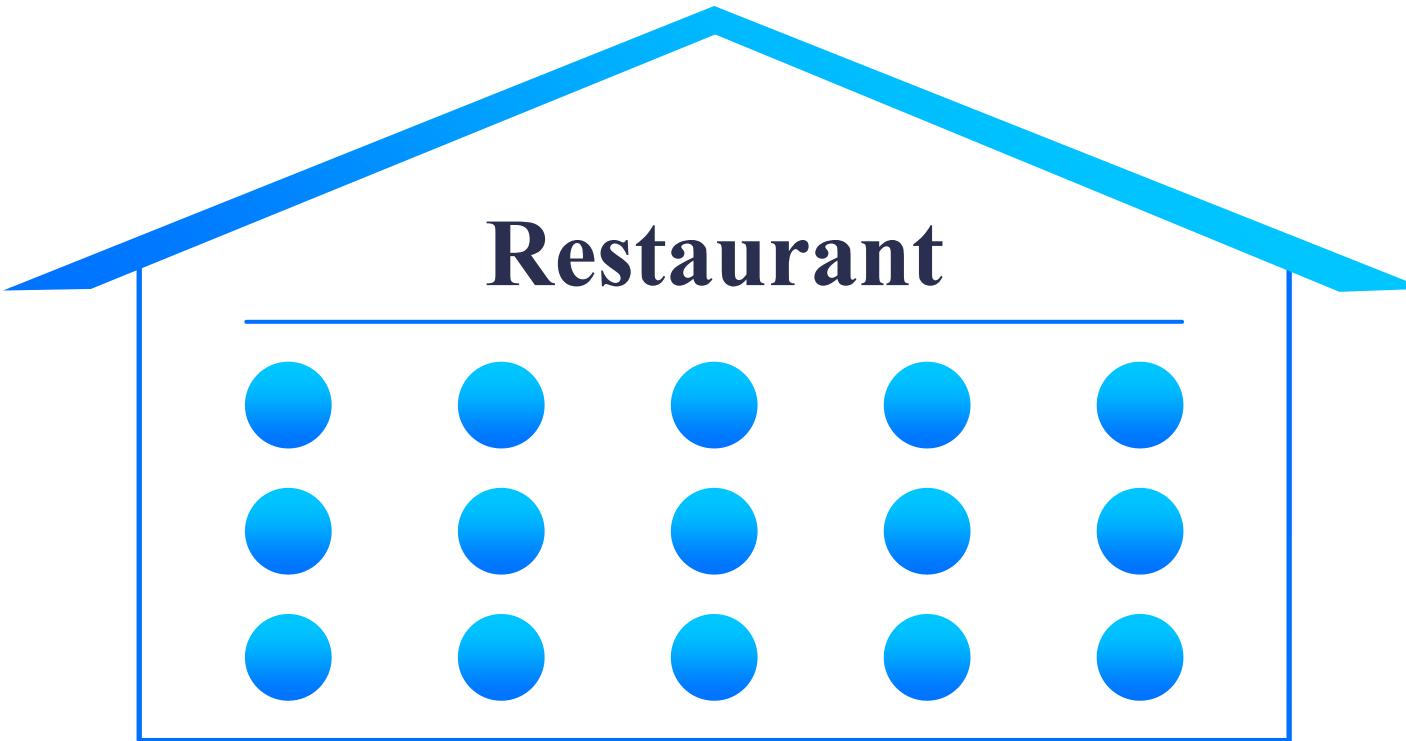
- semaphore `freeTable`
- Khởi tạo `freeTable = 15`

Có 10 khách đến thì:

`freeTable = ...`



## 5.7.1. Định nghĩa semaphore



Ví dụ trên nhà hàng

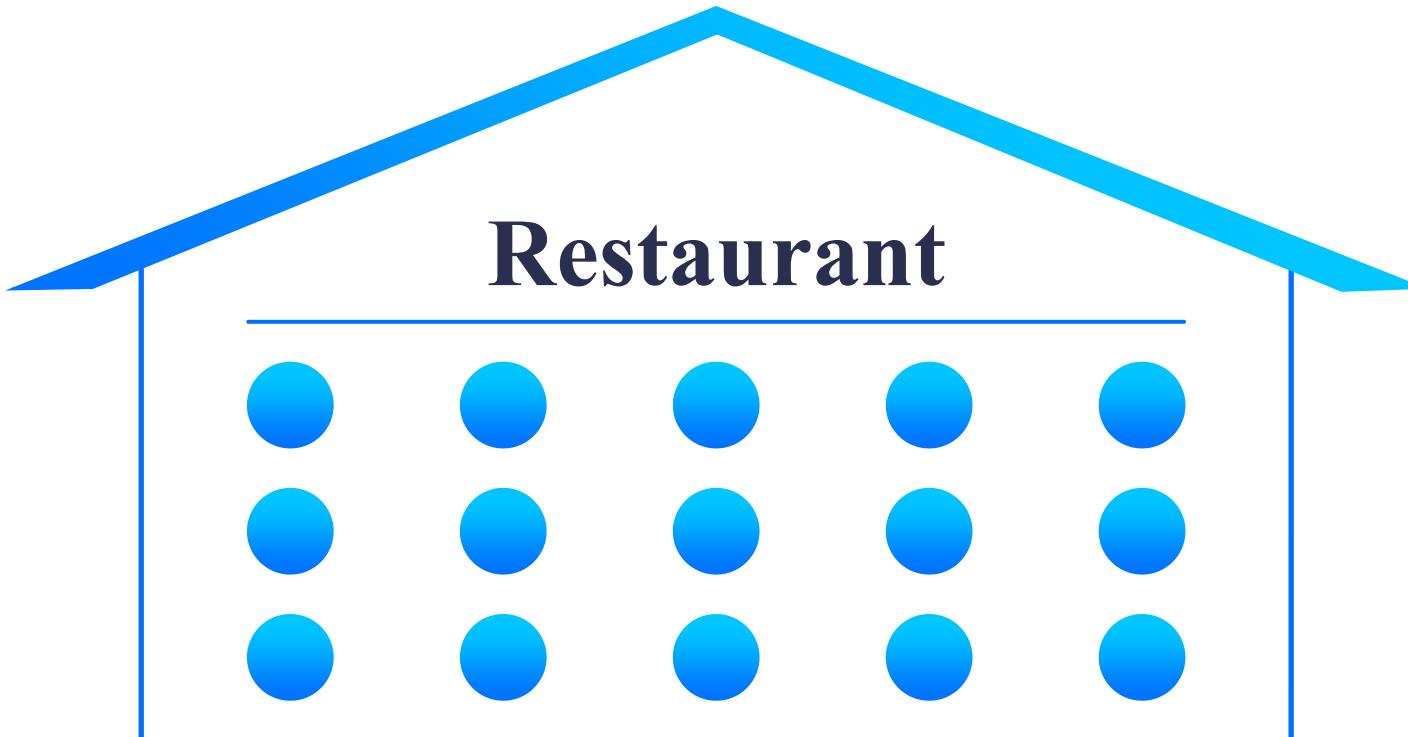
- semaphore `freeTable`
- Khởi tạo `freeTable = 15`

Có 15 khách đến thì:

`freeTable = 0`



## 5.7.1. Định nghĩa semaphore



Ví dụ trên nhà hàng

- semaphore **freeTable**
- Khởi tạo **freeTable = 15**

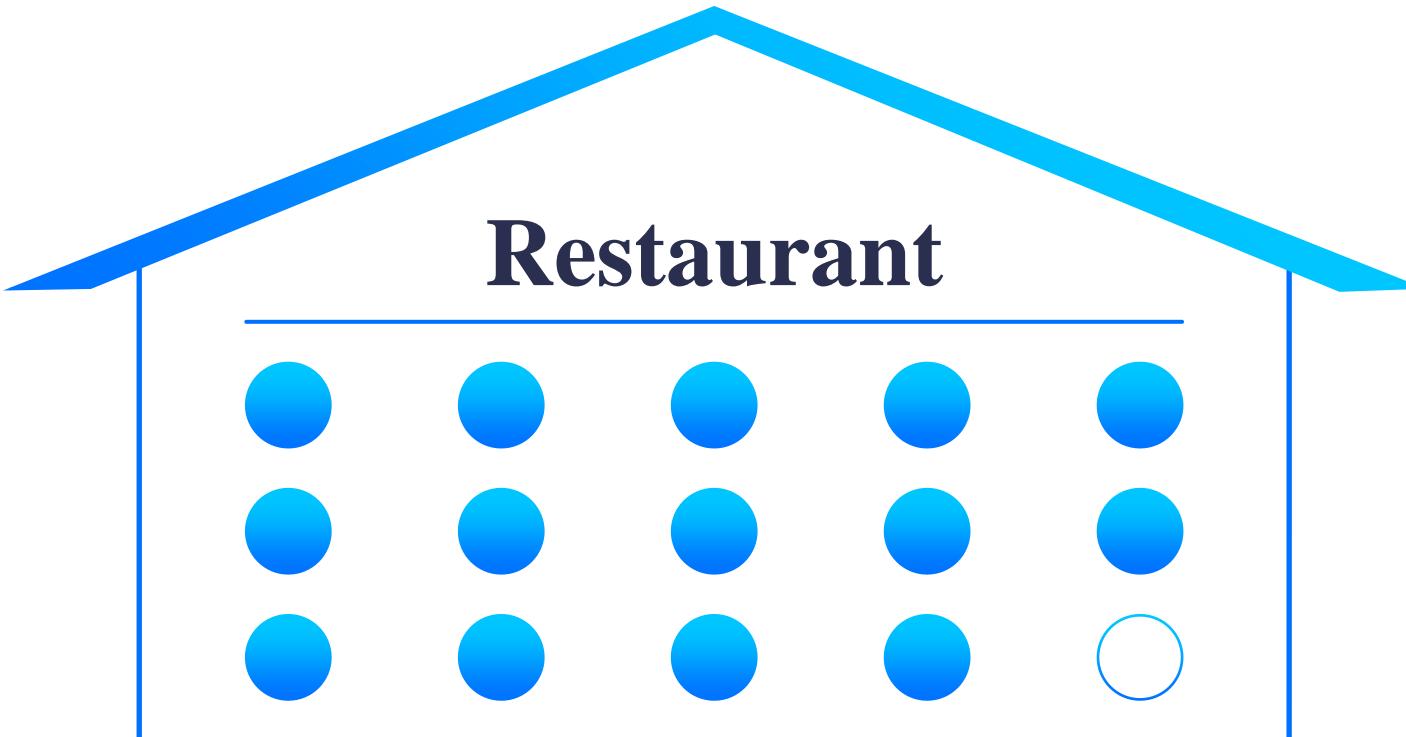
**Khách P16 đến:**

```
wait (freeTable);  
// freeTable = 0 → chờ  
<dùng bữa>;
```

→ Có 1 khách phải chờ ở ngoài



## 5.7.1. Định nghĩa semaphore



Ví dụ trên nhà hàng

- semaphore **freeTable**
- Khởi tạo **freeTable = 15**

**Khách P16 đến:**

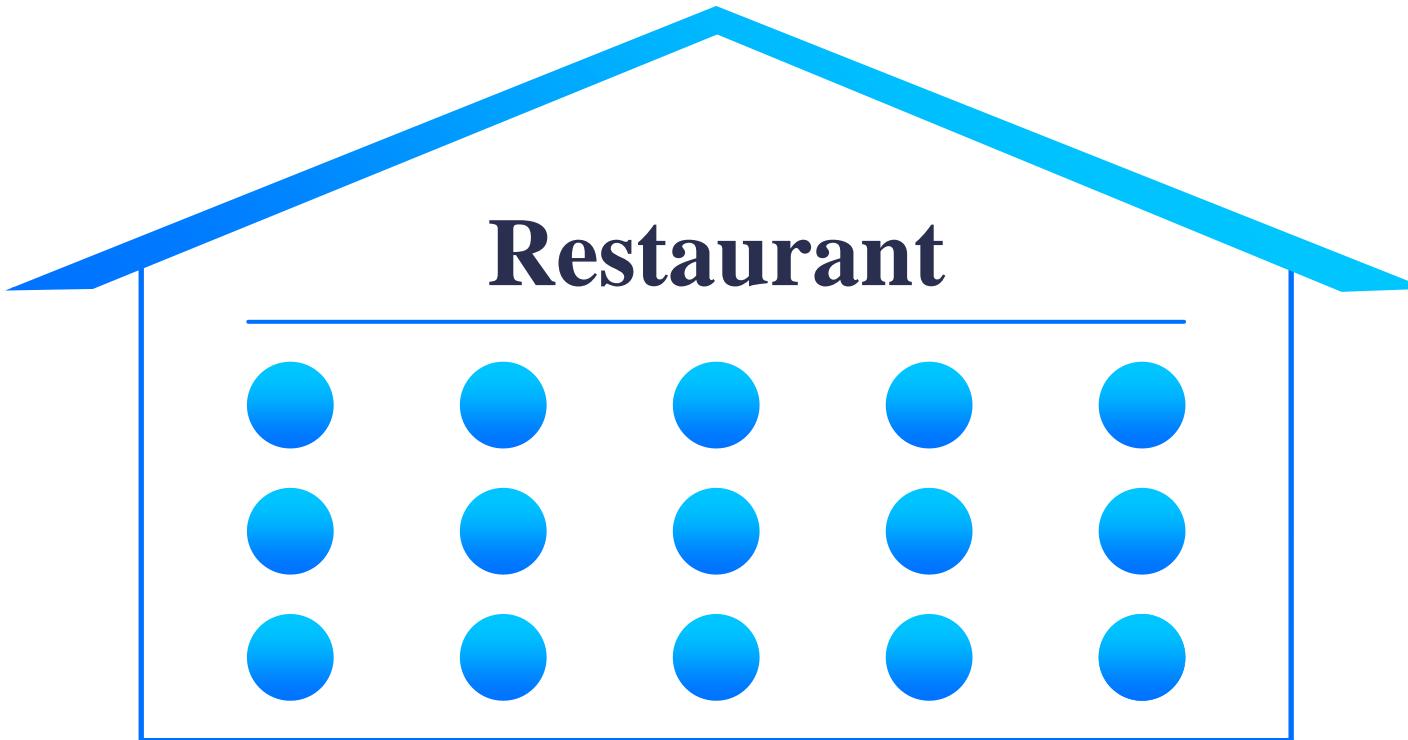
```
wait (freeTable);  
//freeTable = 1 → có thể vào  
<dùng bữa>;
```

**Khách P15 đi**

```
signal (freeTable);  
//freeTable = 1
```



## 5.7.1. Định nghĩa semaphore



Ví dụ trên nhà hàng

- semaphore **freeTable**
- Khởi tạo **freeTable = 15**

**Khách P16 đến:**

```
wait (freeTable);  
//freeTable = 1 → có thể vào  
<dùng bữa>;
```

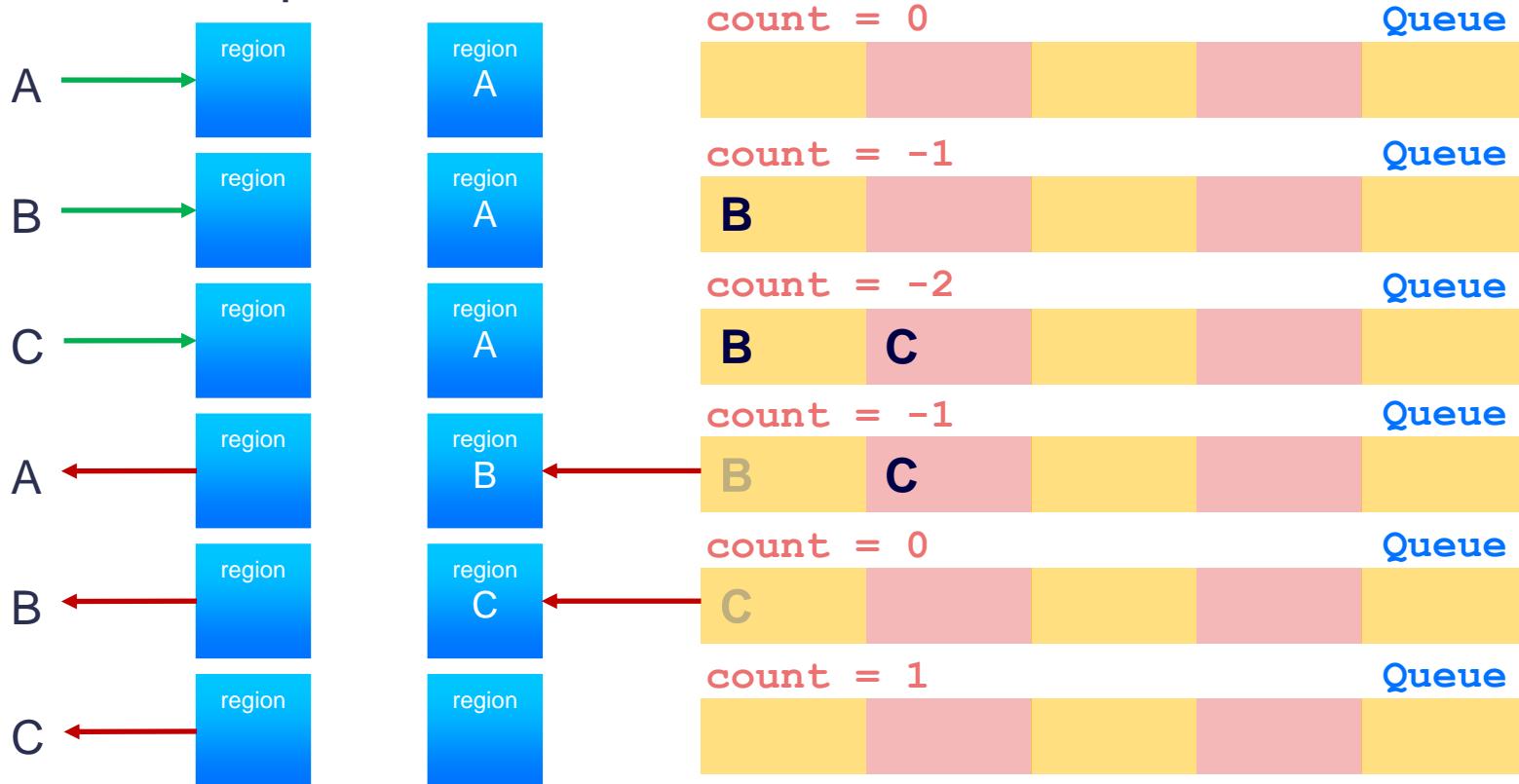
**Khách P15 đi**

```
signal (freeTable);  
//freeTable = 1
```



## 5.7.1. Định nghĩa semaphore

Khởi tạo semaphore count = 1



Ví dụ trên tiến trình



# SEMAPHORE

## 5.7.2. Phân loại semaphore

Semaphore được chia thành 2 loại gồm: counting semaphore và binary semaphore. Trong phần này, ta sẽ đi tìm hiểu về sự khác biệt của 2 loại semaphore này như thế nào.

07.



## 5.7.2. Phân loại semaphore

### Counting semaphore

- Giá trị là số nguyên **không giới hạn**

### Binary semaphore

- Giá trị là **0 hoặc 1**
- Có tác dụng giống với khóa mutex

Có thể sử dụng counting semaphore như một binary semaphore



# SEMAPHORE

## 5.7.3. Hiện thực semaphore

Sau khi đã biết được công dụng của semaphore, câu hỏi đặt ra là: ta sẽ hiện thực semaphore như thế nào?

Trong phần này, ta sẽ thảo luận các cách để hiện thực semaphore, các vấn đề gặp phải và cách hệ điều hành hỗ trợ để hiện thực kỹ thuật này.

07.



## 5.7.3. Hiện thực semaphore

**int S; // semaphore là một số nguyên**

Định nghĩa thao tác **wait()**

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

Định nghĩa thao tác **signal()**

```
signal(S) {  
    S++;  
}
```

- Cần phải đảm bảo rằng không có 2 tiến trình nào cùng lúc thực hiện thao tác **wait()** và **signal()** của một semaphore.
- Việc hiện thực semaphore cũng là một bài toán vùng tranh chấp, hàm **wait()** và **signal()** cũng nằm trong vùng tranh chấp.

- Có thể thực hiện **busy waiting** ở trong vùng tranh chấp
  - Đoạn code thực hiện ngắn.
  - Busy waiting sẽ ngắn nếu CS hiếm khi được thực thi.
- Tuy nhiên, một số chương trình có thể tồn tại nhiều thời gian trong CS → đây không phải giải pháp tốt.



## 5.7.3. Hiện thực semaphore

### Hiện thực semaphore không busy waiting

- Mỗi semaphore được gắn với một hàng đợi.
- Mỗi phần tử trong hàng chờ có 2 thành phần:
  - Giá trị số nguyên (giá trị của semaphore).
  - Con trỏ chỉ đến phần tử tiếp theo (danh sách liên kết đơn).
- Hệ điều hành cần cung cấp 2 thao tác:
  - **block**: tạm dừng và đặt tiến trình gọi thao tác này vào trong hàng đợi – **trạng thái ngủ**.
  - **wakeup**: xóa một tiến trình ra khỏi hàng đợi và đặt lại vào trong hàng đợi sẵn sàng – **đánh thức**.



## 5.7.3. Hiện thực semaphore

### Hiện thực semaphore không busy waiting

waiting queue

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```





## 5.7.3. Hiện thực semaphore

### Hiện thực semaphore không busy waiting

#### Định nghĩa thao tác wait()

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

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

#### Định nghĩa thao tác signal()

```
signal(S) {  
    S++;  
}
```

```
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```



# SEMAPHORE

## 5.7.4. Ứng dụng semaphore

Phần này sẽ trình bày về các ứng dụng của semaphore trong các trường hợp thực tế bao gồm: đảm bảo loại trừ tương hỗ, đồng bộ thứ tự hoạt động của các tiến trình, đồng bộ hoạt động theo điều kiện.

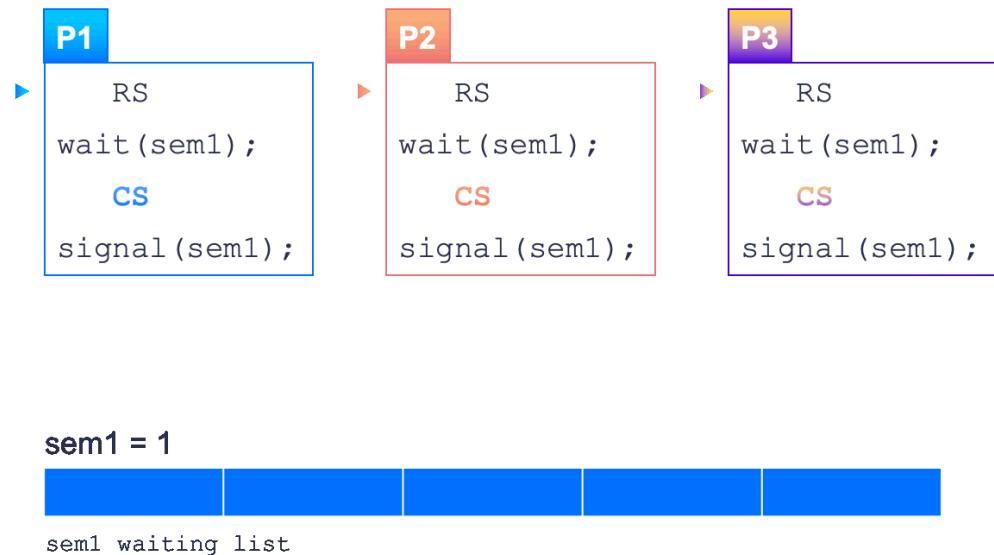
07.



## 5.7.4. Ứng dụng semaphore

### Đảm bảo loại trừ tương hỗ

- Semaphore hoạt động như một khóa mutex.
- Bao CS bằng thao tác wait () và signal () .
- Khởi tạo giá trị của semaphore là 1  
→ Chỉ tiến trình nào gọi wait () trước thì mới được tiến vào CS.





## 5.7.4. Ứng dụng semaphore

### Đảm bảo thứ tự thực thi

Đồng bộ P1 và P2 sao cho S1 luôn luôn thực thi trước S2.

- Khởi tạo semaphore synch = 0
- Phân tích thứ tự thực thi:
  - Nếu S1 thực thi trước: không sao
  - Nếu S2 thực thi trước: block



synch = 0



synch waiting list



## 5.7.4. Ứng dụng semaphore

### Đảm bảo điều kiện

Đồng bộ tiến trình

**Produce** và **Consume** sao cho  
**sells <= products**

- **Bước 1:** Dựa vào điều kiện, xác định tài nguyên
  - Số đơn vị mà **sells** được tăng
- **Bước 2:** Xác định số lượng semaphore
  - Quản lý 1 tài nguyên → **1 semaphore**
- **Bước 3:** Đặt **wait()** và **signal()**
- **Bước 4:** Dựa vào trạng thái của hệ thống, xác định giá trị của semaphore

Produce

products++

semaphore sem;

Produce

products++

signal (sem)

products = 0, sells = 0

sem = 0

Consume

sells++

Consume

wait (sem)

sells++



# SEMAPHORE

## 5.7.5. Một số nhận xét về semaphore

Semaphore là một công cụ đắc lực giúp cho việc đồng bộ các tiến trình/tiểu trình trở nên dễ dàng hơn rất nhiều. Khi đã nắm rõ được cách hoạt động và ứng dụng của semaphore, chúng ta có thể rút ra được một vài nhận xét sau khi quan sát cơ chế mà semaphore hoạt động.

07.



## 5.7.5. Một số nhận xét về semaphore

Xét semaphore S:

- Khi  $S->value \geq 0$ : số lần mà các tiến trình/tiểu trình có thể thực thi wait(S) mà không bị blocked là  $S->value$

$S = 5 // Các tiến trình có thể thực thi wait(S) 5 lần mà không bị blocked$



S waiting list

- Khi  $S->value < 0$ : số tiến trình/tiểu trình đang đợi trên S là  $|S->value|$

$S = -5 // Có 5 tiến trình đang bị blocked trong hàng đợi của semaphore S$



S waiting list



## 5.7.5. Một số nhận xét về semaphore

- **Atomic và mutual exclusion:** không được xảy ra trường hợp 2 tiến trình cùng đang ở trong thân lệnh `wait (S)` và `signal (S)` (cùng semaphore S) tại một thời điểm (ngay cả với hệ thống multiprocessor).  
→ **Đoạn mã định nghĩa các lệnh `wait (S)` và `signal (S)` cũng chính là vùng tranh chấp.**
- 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.



# SEMAPHORE

## 5.7.6. Các vấn đề khi sử dụng semaphore

Việc sử dụng semaphore yêu cầu tính cẩn thận và chính xác rất cao. Thứ tự của các lệnh wait() và signal() hay giá trị khởi tạo của semaphore có thể ảnh hưởng đến tính đúng đắn hay hiệu suất của chương trình. Hãy cùng khảo sát vấn đề có thể phát sinh nếu sử dụng semaphore không đúng cách trong nội dung sau đây.

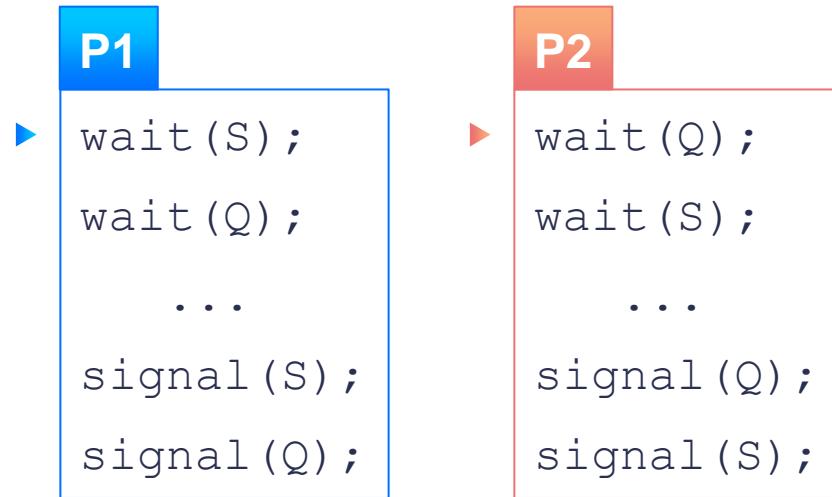
07.



## 5.7.6. Các vấn đề khi sử dụng semaphore

### Khởi tạo

semaphore S = 1, Q = 1



Tồn tại khả năng xảy ra:

P1 | wait(S) // S = 0

P2 | wait(Q) // Q = 0

P1 | wait(Q) // **P1 bị blocked**

P2 | wait(S) // **P2 bị blocked**

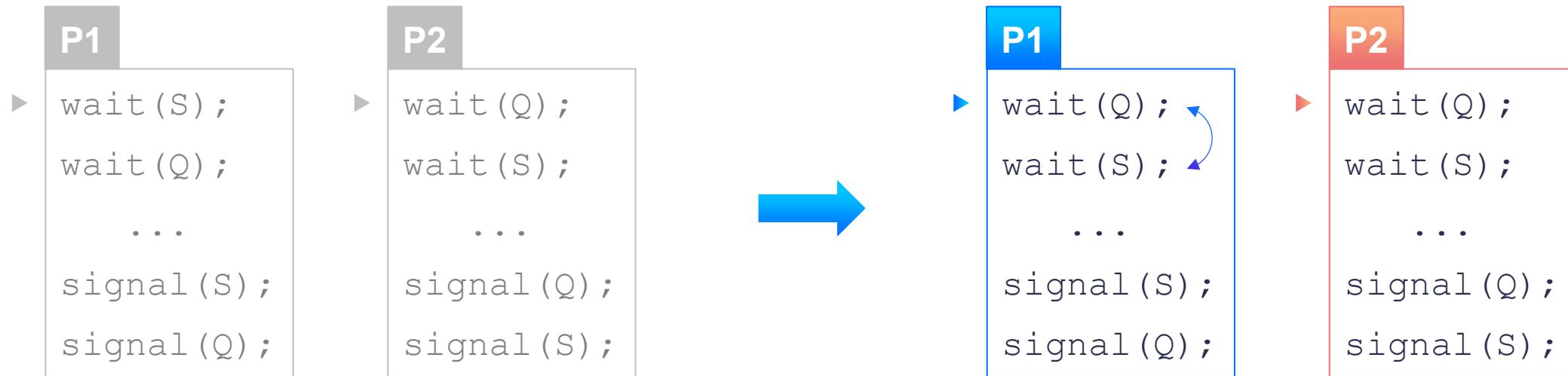
→ DEADLOCK



## 5.7.6. Các vấn đề khi sử dụng semaphore

Khởi tạo

semaphore S = 1, Q = 1



→ Cần phải lưu ý giá trị khởi tạo và thứ tự sắp xếp các thao tác khi sử dụng semaphore



# MONITOR

## 5.8.1. Định nghĩa monitor

Nhiều loại vấn đề có thể dễ dàng phát sinh khi xử lý các vấn đề liên quan đến vùng tranh chấp nếu các lập trình viên sử dụng semaphore và khóa mutex không đúng cách. Một giải pháp được đề xuất để giải quyết các lỗi trên đó là sử dụng các công cụ đồng bộ đơn giản như một cấu trúc ngôn ngữ bậc cao, và đó chính là *monitor*.

08.



## 5.8.1. Định nghĩa monitor

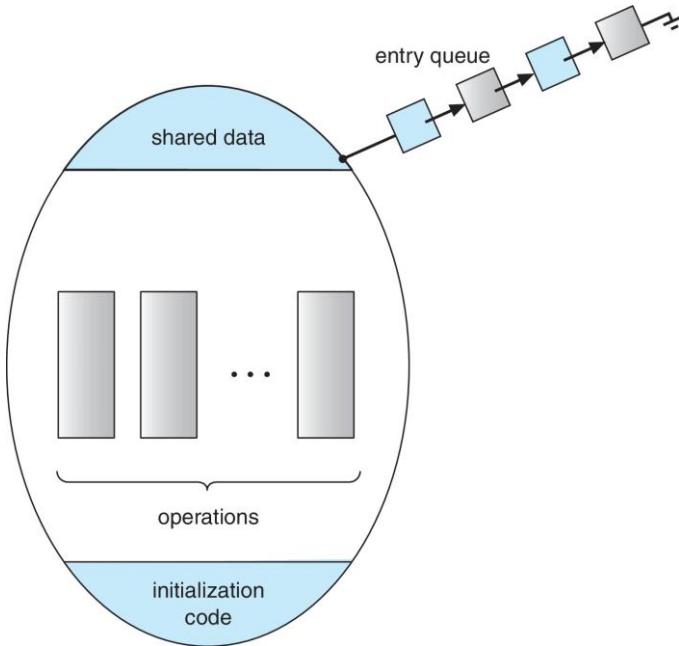
- Là một kiểu dữ liệu trừu tượng (abstract data type) đóng gói những thành phần sau:
  - Các biến nội bộ: được khai báo bên trong monitor và chỉ có thể được truy cập bởi các hàm nội bộ trong monitor.
  - Các thủ tục: Một tập các thao tác được định nghĩa bởi lập trình viên và được thực thi theo loại trừ tương hỗ, các thủ tục này cũng chỉ có thể truy cập các biến nội bộ được khai báo ở trên
  - Đoạn code khởi tạo

### Mã giả của một monitor

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



## 5.8.1. Định nghĩa monitor



Mô hình một monitor đơn giản

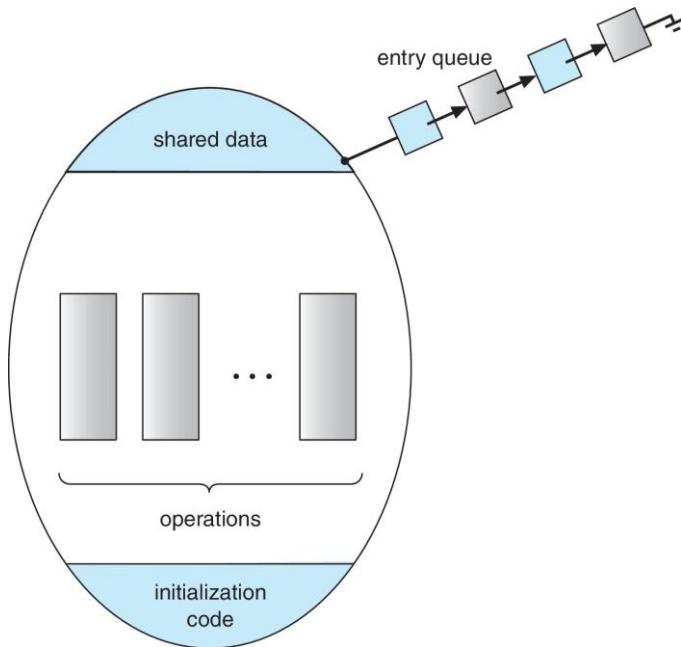
### Hiện thực monitor với semaphore

- Variables:  
semaphore mutex  
mutex = 1
- Mỗi thủ tục P sẽ được thay thế bởi đoạn mã bên dưới:

```
wait (mutex);  
...  
body of P;  
...  
signal (mutex);
```



## 5.8.1. Định nghĩa monitor



Mô hình một monitor đơn giản

### Đặc điểm của monitor

- Tiến trình “vào monitor” bằng cách gọi một trong các thủ tục được định nghĩa trong monitor.
- Chỉ có một tiến trình có thể vào monitor tại một thời điểm → mutual exclusion được bảo đảm.
- Tuy nhiên, cấu trúc của monitor không thực sự mạnh mẽ cho các mô hình đồng bộ khác → cần định nghĩa thêm cơ chế đồng bộ → **cấu trúc condition**.



# MONITOR

## 5.8.2. Condition variable

Condition variable (biến điều kiện) là các biến có kiểu dữ liệu là condition. Chỉ có 2 thao tác có thể được thực hiện trên các biến kiểu condition đó là wait() và signal(). Condition variable cho phép lập trình viên hiện thực các mô hình đồng bộ riêng biệt theo đúng nhu cầu của từng chương trình.

08.



## 5.8.2. Condition variable

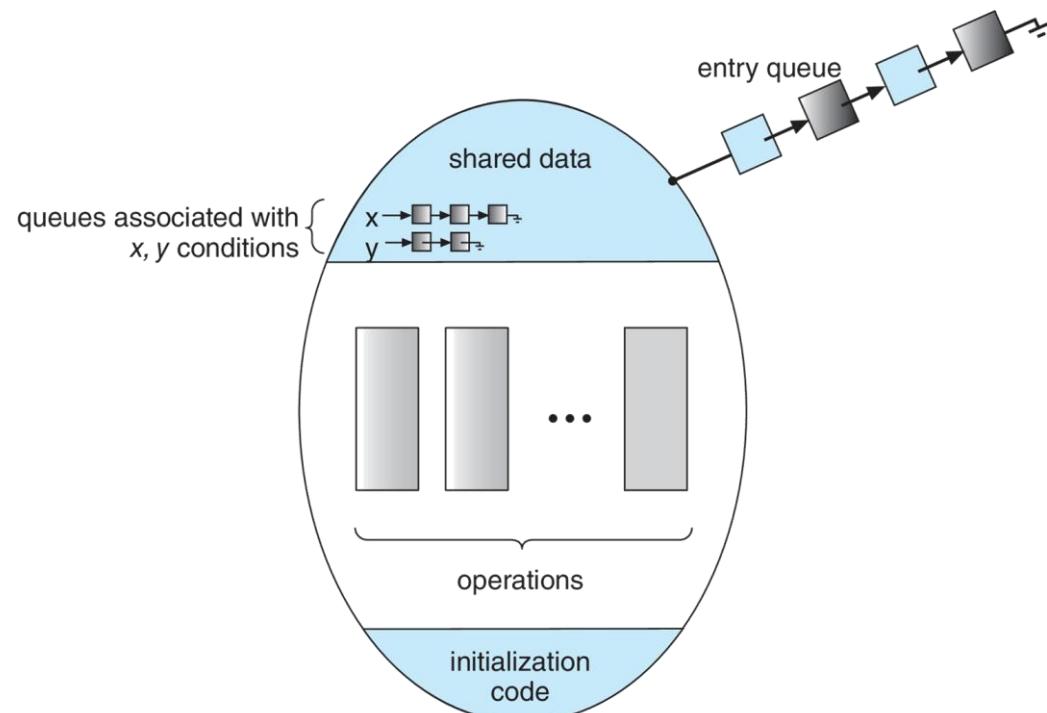
- Nhằm cho phép tiến trình đợi “trong monitor”, chỉ có thể được truy cập bên trong monitor.
- Khai báo:  
`condition x, y;`
- Chỉ có thể thao tác lên condition variable bằng 02 thao tác:
  - `x.wait()` – tiến trình thực thi thao tác này sẽ bị *block* trên condition variable x cho đến khi thao tác `x.signal()` được thực thi.
  - `x.signal()` – phục hồi quá trình thực thi của một tiến trình (nếu có) bị block trên condition variable x.
    - Nếu có nhiều tiến trình bị block: chỉ một tiến trình được phục hồi.
    - Nếu không có tiến trình nào bị block: không có tác dụng.



## 5.8.2. Condition variable

### Đặc điểm của condition variable

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

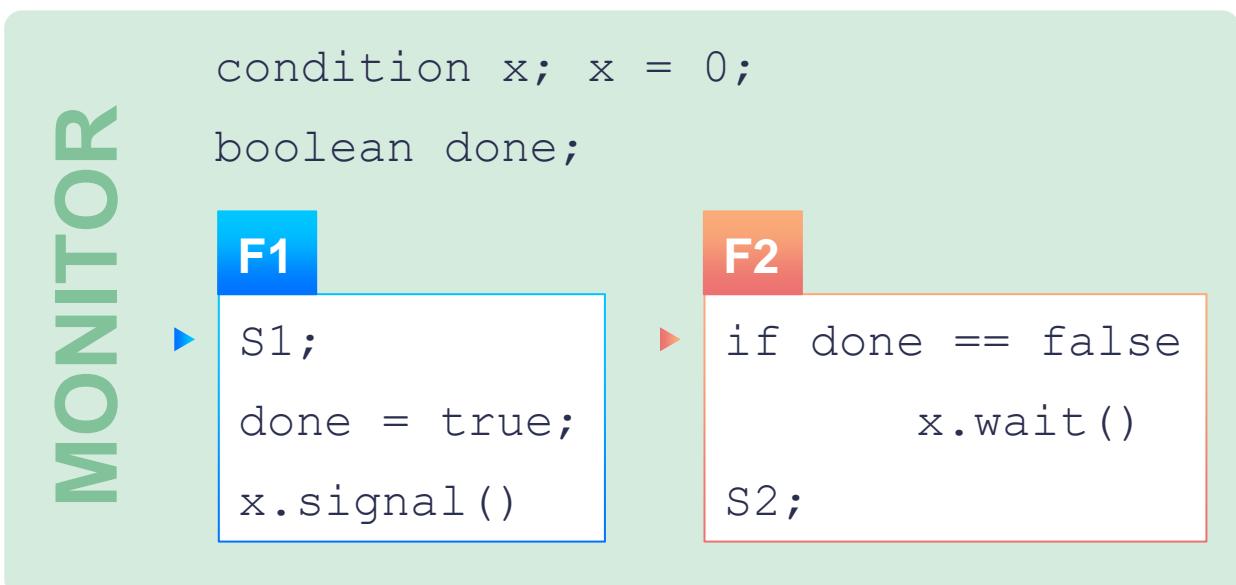




## 5.8.2. Condition variable

### Sử dụng condition variable

- Đồng bộ P1 và P2 sao cho S1 luôn luôn thực thi trước S2





# LIVENESS

Quá trình đồng bộ tiến trình có thể gây ra các lỗi nghiêm trọng trong việc làm tiến trình bị “kẹt” và không thể tiếp tục chạy. Liveness là thuật ngữ để chỉ một tập các đặc điểm mà hệ thống phải thỏa mãn để đảm bảo rằng các tiến trình thực sự đang chạy.

09.



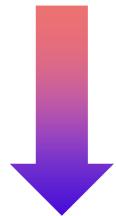
## 5.9 Liveness

- Tiến trình có thể phải chờ vô thời hạn để cố gắng yêu cầu các công cụ đồng bộ như mutex hay semaphore → vi phạm tiêu chí **progress** và **bounded waiting**.
- **Liveness** là thuật ngữ để chỉ một tập các đặc điểm mà hệ thống phải thỏa mãn để đảm bảo tiến trình thực sự chạy.
- Chờ đợi không giới hạn là một ví dụ tiêu biểu cho việc liveness thất bại (tiến trình không còn chạy).



## 5.9 Liveness

### DEADLOCK



là tình trạng **hai hay nhiều tiến trình** đang **chờ đợi không giới hạn** cho một sự kiện mà sự kiện này chỉ có thể được thực hiện bởi một trong các tiến trình đang chờ ở trên.

P1

```
▶ wait (S) ;  
wait (Q) ;  
...  
signal (S) ;  
signal (Q) ;
```

P2

```
▶ wait (Q) ;  
wait (S) ;  
...  
signal (Q) ;  
signal (S) ;
```

Một số dạng khác của deadlock:

- **Starvation – đói:**
  - Một tiến trình có thể không bao giờ được thoát ra khỏi hàng đợi của semaphore mà nó đang chờ.
- **Priority inversion – nghịch đảo ưu tiên:**
  - Vấn đề định thời khi tiến trình có độ ưu tiên thấp giữ khóa mà đang được cần bởi tiến trình có độ ưu tiên cao.
  - Có thể được giải quyết bằng priority inheritance protocol.



# BÀI TOÁN ĐỒNG BỘ BOUNDED-BUFFER

## 5.10.1. Phát biểu bài toán bounded-buffer

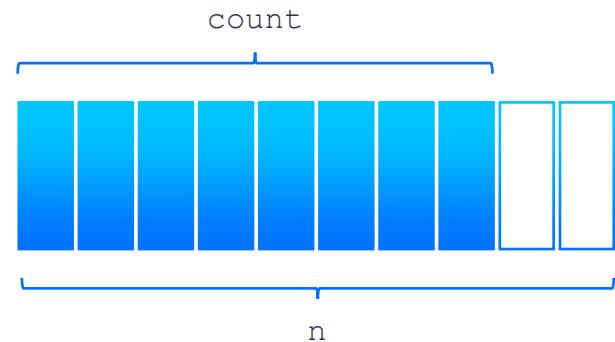
Bài toán bounded-buffer (đã được đề cập trong phần 5.1) mô tả một mảng có giới hạn số phần tử và 02 tiến trình lùn lượt là Producer và Consumer có nhiệm vụ đồng thời ghi và xóa phần tử ra khỏi mảng.

10.



## 5.10.1. Phát biểu bài toán bounded-buffer

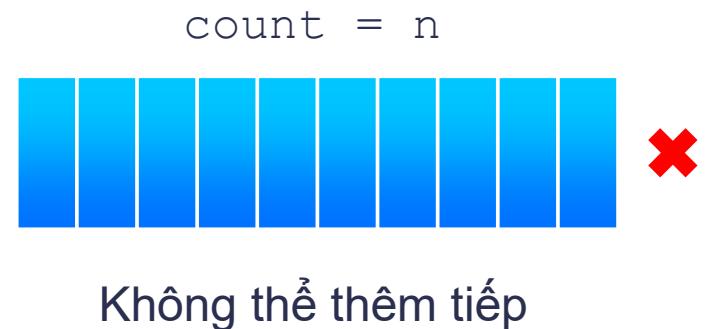
- Một mảng buffer[] có tối đa n phần tử  
Số lượng phần tử trong mảng là count
- Tiến trình **Producer** thêm phần tử vào mảng
- Tiến trình **Consumer** xóa phần tử khỏi mảng



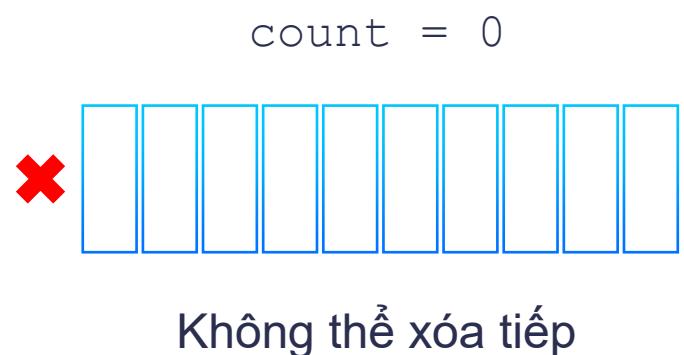


## 5.10.1. Phát biểu bài toán bounded-buffer

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



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





# BÀI TOÁN ĐỒNG BỘ BOUNDED-BUFFER

## 5.10.2. Giải pháp cho bài toán bounded-buffer

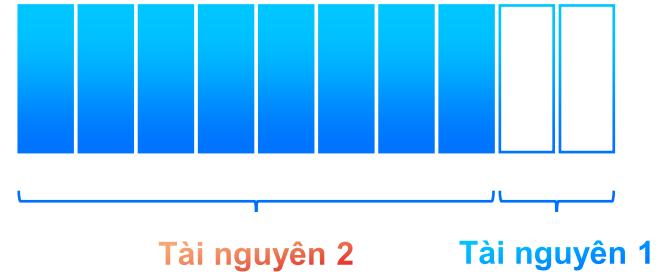
Để giải quyết bài toán bounded-buffer, ta cần xác định đúng điều kiện và áp dụng semaphore để đồng bộ cho các điều kiện này.

10.



## 5.10.2. 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++;
```

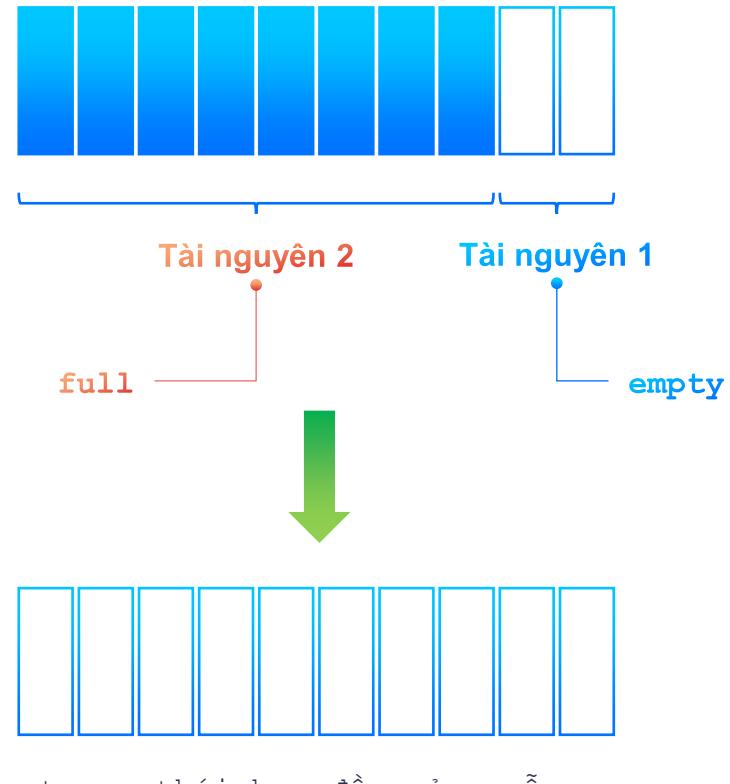
### Consumer

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



## 5.10.2. 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**



trạng thái ban đầu mảng rỗng:

**empty = n;**

**full = 0;**



## 5.10.2. Giải pháp cho bài toán bounded-buffer

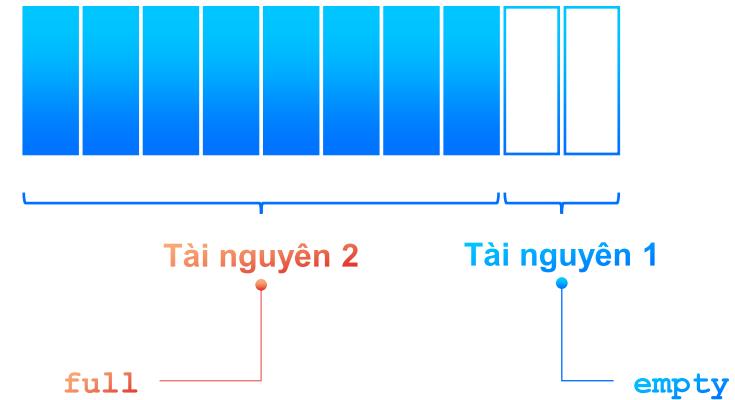
- Bước 3: Đặt wait () và signal ()

### 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

## 5.10.3. 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.

10.



## 5.10.3. 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);  
}
```

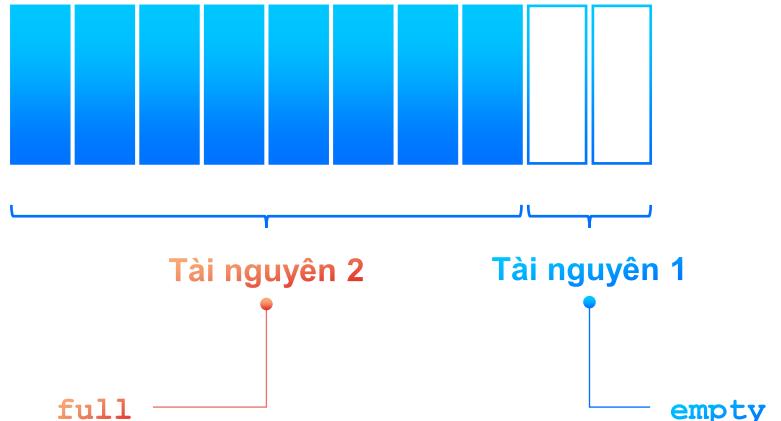
BUSY WAITING

#### 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);
```





## 5.10.3. 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);
```

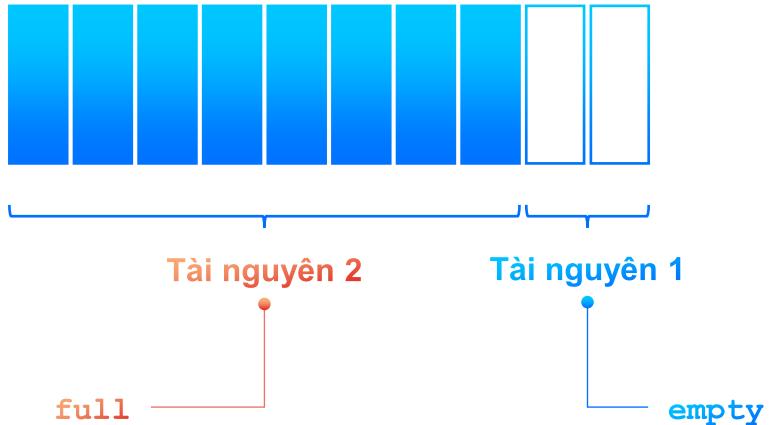
**Consumer**

```
▶ 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

## 5.11.1. Phát biểu bài toán Readers-Writers

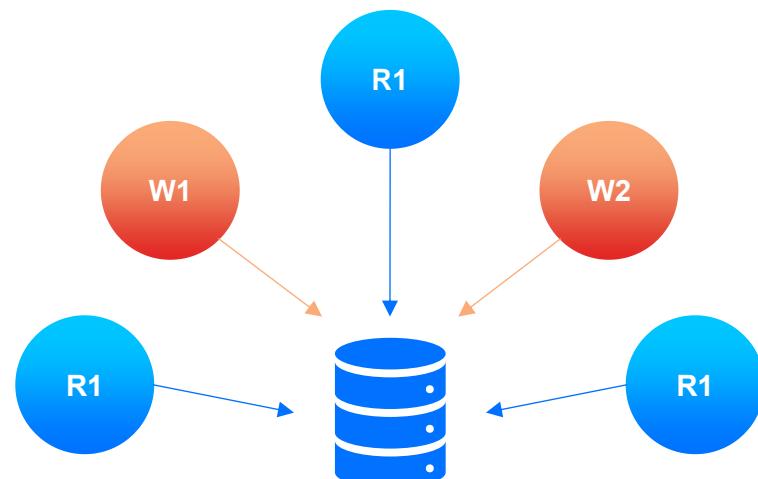
Giả sử tồn tại một cơ sở dữ liệu được chia sẻ giữa những tiến trình đang thực thi đồng thời. Một số tiến trình chỉ muốn đọc dữ liệu trong khi một số khác muốn cập nhật (vừa đọc và ghi) vào cơ sở dữ liệu này. Các tiến trình chỉ muốn đọc dữ liệu được gọi là **Readers** và các tiến trình muốn cập nhật được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

11.



## 5.11.1. Phát biểu bài toán Readers-Writers

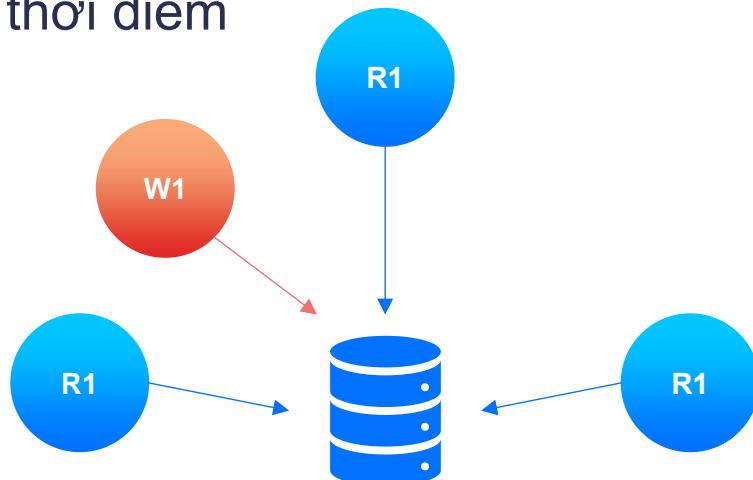
- Dữ liệu được chia sẻ giữa các tiến trình đang thực thi đồng thời
  - **Readers**: Chỉ đọc dữ liệu; không thực hiện cập nhật
  - **Writers**: Có thể vừa đọc vừa ghi dữ liệu





## 5.11.1. 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





## 5.11.1. 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

## 5.11.2. 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.

11.



## 5.11.2. Giải pháp cho bài toán Readers-Writers

### Biến thê 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



## 5.11.2. 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

## 5.12.1. Phát biểu bài toán Dining-Philosopher

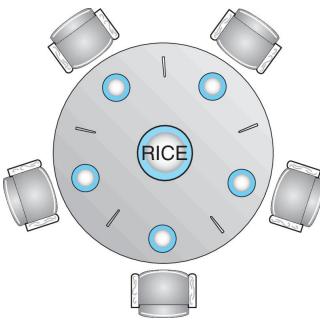
Bài toán Dining-Philosopher hay còn gọi là Bài toán Các Triết gia ăn tối được xem là bài toán đồng bộ kinh điển không phải bởi vì tính quan trọng trong thực hành hay bởi các nhà khoa học máy tính không thích các Triết gia mà bởi vì đây là một ví dụ cho các vấn đề đồng bộ trên quy mô lớn. Bài toán này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

12.



## 5.12.1. Phát biểu bài toán Dining-Philosopher

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

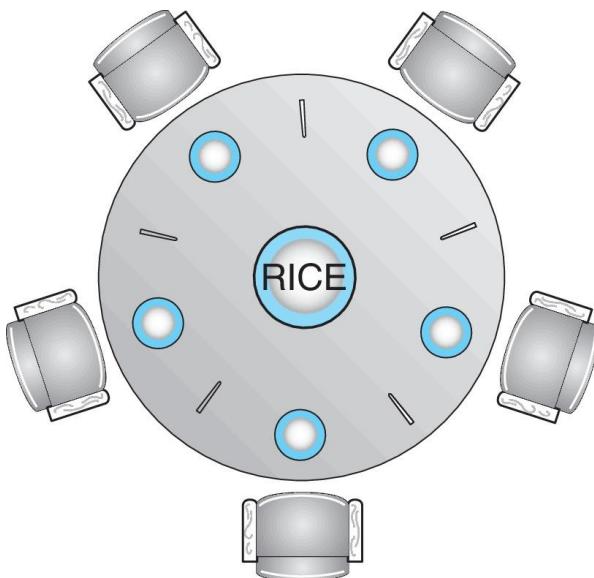


- Họ dành thời gian chỉ để làm 2 công việc: suy nghĩ và ăn.
- Họ không tương tác với những người bên cạnh.
- Thi thoảng các Triết gia sẽ cầm lên 2 chiếc đũa (mỗi chiếc 1 lần) để ăn:
  - Cần phải có đủ 2 chiếc đũa thì mới ăn được, sau khi ăn xong thì trả lại 2 chiếc đũa.
  - Số chiếc đũa cũng là  $n$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



## 5.12.1. 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

## 5.12.2. 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.

12.

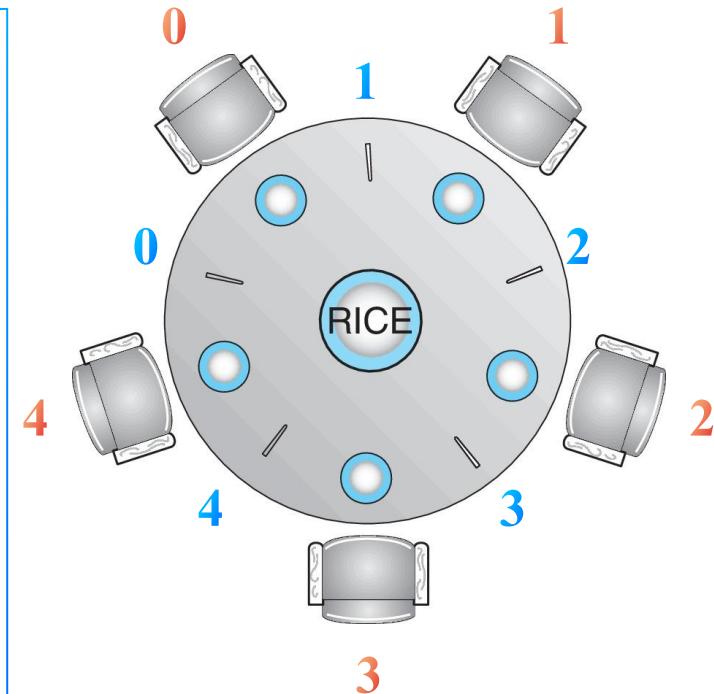


## 5.12.2. Giải pháp cho bài toán Dining-Philosopher

### 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 */  
...
```





## 5.12.2. Giải pháp cho bài toán Dining-Philosopher

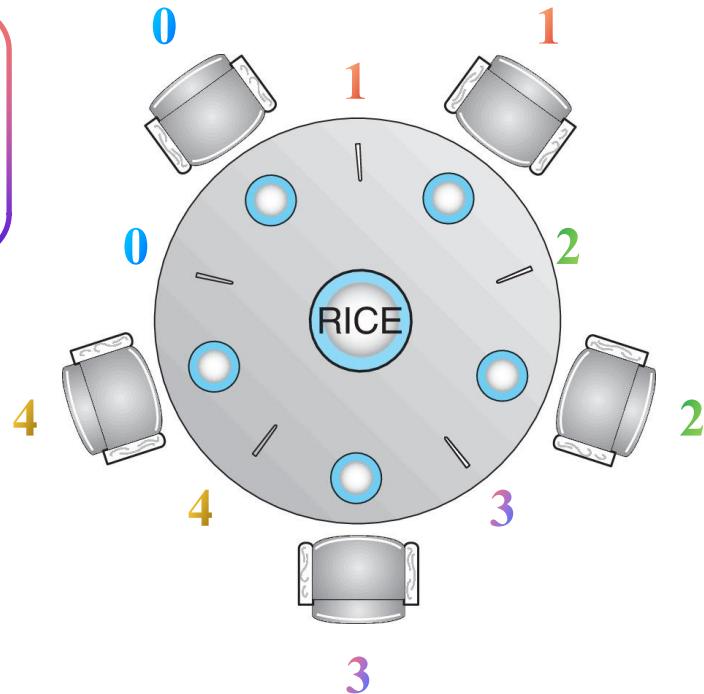
### Giải pháp sử dụng semaphore

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**





## 5.12.2. 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.

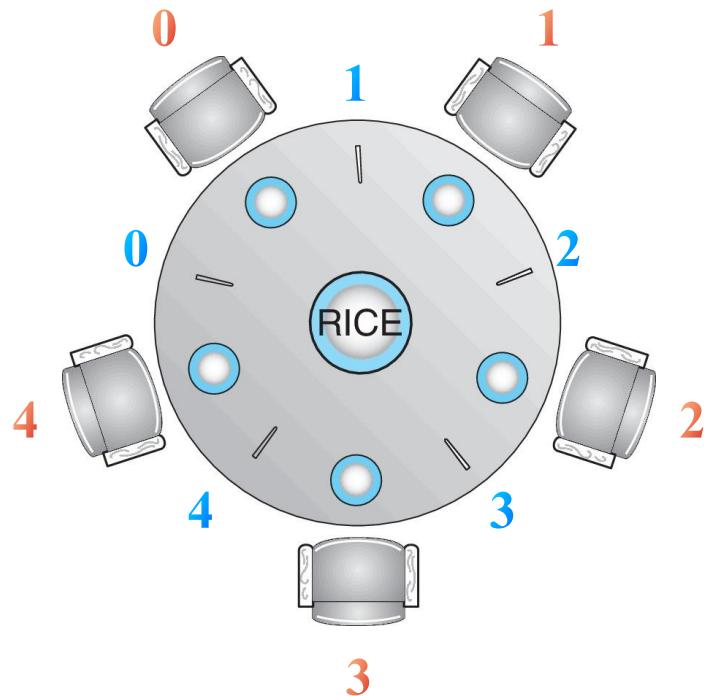


## 5.12.2. Giải pháp cho bài toán Dining-Philosopher

### 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





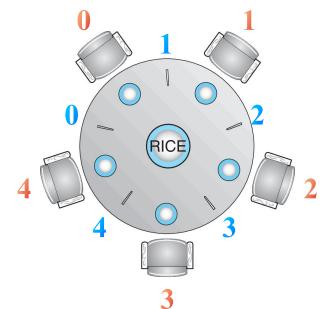
## 5.12.2. Giải pháp cho bài toán Dining-Philosopher

### 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;
}
}
```





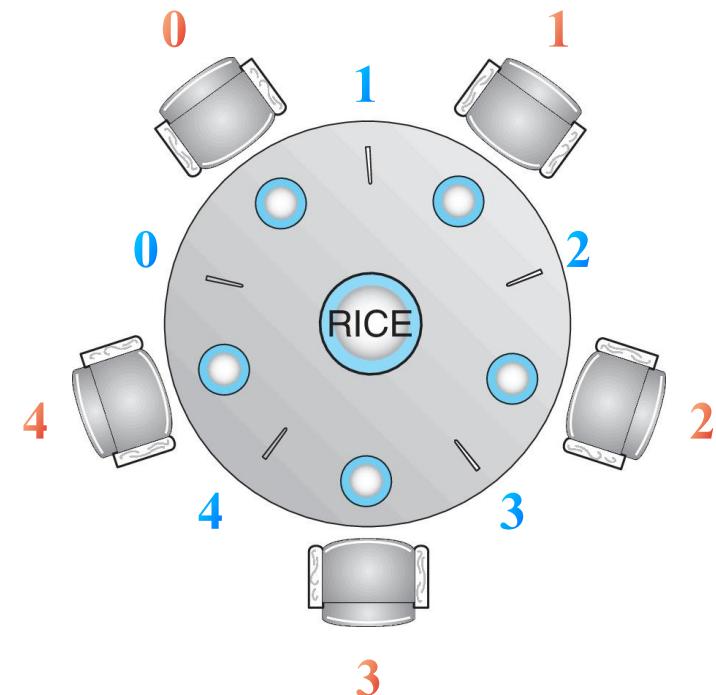
## 5.12.2. Giải pháp cho bài toán Dining-Philosopher

### 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

- Các giải pháp đồng bộ
  - Semaphore
  - Monitor
  - Liveness
- Các bài toán đồng bộ
  - Bài toán đồng bộ bounded-buffer
  - Bài toán đồng bộ readers-writers
  - Bài toán đồng bộ dining-philosophers



# THẢO LUẬN

