



HỆ ĐIỀU HÀNH

CHƯƠNG 5: ĐỒNG BỘ TIẾN TRÌNH (PHẦN 2)

Bên cạnh việc sử dụng các ngắt hoặc cản hõ trợ từ phần cứng, hệ điều hành cũng cung cấp các cơ chế giúp thực thi việc đồng bộ các tiến trình/tiểu tình. Phần 2 của chương đồng bộ tiến trình sẽ giới thiệu các kỹ thuật sử dụng semaphore, mutex, monitor vốn là các kỹ thuật rất phổ biến trong đồng bộ tiến trình.



MỤC TIÊU

1. Diễn tả được cơ chế hoạt động của mutex lock, semaphore và monitor trong việc giải quyết bài toán vùng tranh chấp
2. Phân tích chương trình ứng dụng mutex lock và semaphore
3. Viết được chương trình sử dụng mutex lock và semaphore để thực hiện đồng bộ thứ tự hoạt động của tiến trình/tiểu tiến trình
4. Trình bày được vấn đề Liveness trong hoạt động đồng bộ tiến trình



NỘI DUNG

6. Mutex locks
7. Semaphore
8. Monitor
9. Liveness



Review

GIẢI QUYẾT BÀI TOÁN VÙNG TRANH CHẤP

- Các giải pháp dựa trên ngắt **gây lãng phí** tài nguyên CPU khi liên tục kiểm tra điều kiện chờ đợi tiến vào CS (busy waiting).
- Các giải pháp hỗ trợ từ phần cứng thì **khá phức tạp** và **không thể truy cập** được bởi lập trình viên của các chương trình ứng dụng.



Các nhà thiết kế hệ điều hành xây dựng các **công cụ phần mềm cấp cao** để giải quyết vấn đề vùng tranh chấp:

- **Mutex locks**
- **Semaphore**
- **Monitor**



MUTEX LOCKS

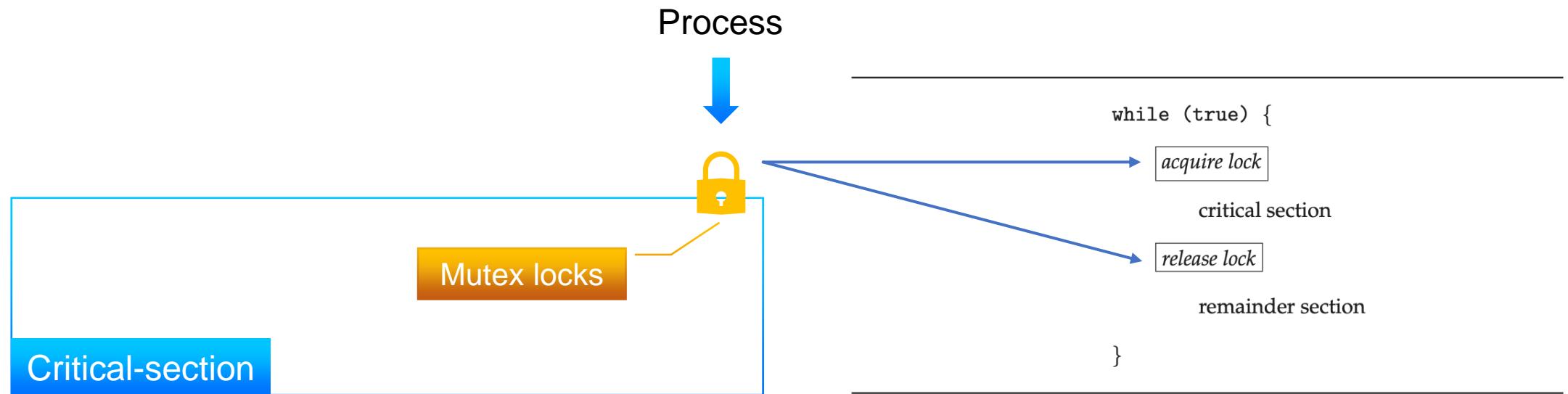
5.6.1. Định nghĩa mutex locks

Mutex (viết tắt của Mutual exclusion) locks hay còn gọi lại “khóa mutex” là kỹ thuật giúp đảm bảo yêu cầu loại trừ tương hỗ khi các tiến trình thực thi đồng thời với nhau. Mutex locks hoạt động như một ô khóa, khi tiến trình tiến vào vùng tranh chấp thì cần phải yêu cầu khóa ô khóa lại, tương tự, sau khi ra khỏi vùng tranh chấp thì tiến trình cần yêu cầu mở khóa mutex để tiến trình khác có thể tiến vào.

06.



5.6.1. Định nghĩa mutex locks





5.6.1. Định nghĩa mutex locks

```
while (true) {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
}
```

```
acquire() {  
    while (!available);  
    /* busy wait */  
    available = false;  
}
```

```
release() {  
    available = true;  
}
```

Thao tác gọi `acquire()` hoặc `release()` phải được thực hiện **đơn nguyên**
→ Có thể được hiện thực thông qua lệnh phần cứng đơn nguyên như `compare_and_swap`.



5.6.1. Định nghĩa mutex locks

```
acquire() {  
    while (!available);  
        /* busy wait */  
    available = false;  
}
```

Yêu cầu busy waiting → Lãng phí CPU

```
release() {  
    available = true;  
}
```

Giải pháp này còn được gọi là **spinlock**



MUTEX LOCKS

5.6.2. Mutex locks không busy waiting

Để tránh busy waiting trong việc sử dụng khóa mutex, hệ điều hành cung cấp cơ chế cho phép khóa tiến trình – hay chủ động đưa tiến trình vào trạng thái ngủ, song song đó là cơ chế đánh thức tiến trình để đưa tiến trình vào trạng thái hoạt động trở lại. Hãy khảo sát cách thức triển khai mutex locks không busy waiting ngay sau đây.

06.



5.6.2. Mutex locks không busy waiting

- Để tránh busy waiting trong mutex locks, ta tạm thời đặt tiến trình vào trạng thái ngủ khi khóa bị khóa, và sau đó đánh thức tiến trình dậy khi khóa được mở.
- Hệ điều hành cần cung cấp 2 thao tác:
 - **block**: tạm dừng và đặt tiến trình gọi thao tác này vào hàng đợi – *trạng thái ngủ*
 - **wakeup**: xóa một tiến trình ra khỏi hàng đợi và đặt lại vào hàng đợi sẵn sàng – *đánh thức*



5.6.2. Mutex locks không busy waiting

```
acquire() {  
    while (!available);  
    /* busy wait */  
    available = false;  
}
```

```
release() {  
    available = true;  
}
```



```
acquire() {  
    if (!available);  
    block();  
    available = false;  
}
```



```
release() {  
    available = true;  
    wakeup(Q);  
}
```

Tiến trình P muốn vào CS:

- Nếu khóa mutex **đang mở**: tiến trình khóa lại và tiến vào CS
- Nếu khóa mutex **đang khóa**: tiến trình P bị block và vào trạng thái ngủ

Tiến trình P sau khi hoàn thành CS:

- Mở khóa mutex
- Đánh thức tiến trình Q (nếu có) đang ngủ trong hàng đợi



MUTEX LOCKS

5.6.3. Cách sử dụng mutex locks

Sau khi đã tìm hiểu về công dụng của khóa mutex, trong phần tiếp theo, ta sẽ đi tìm hiểu xem cách sử dụng cụ thể của khóa mutex trong code như thế nào?

06.

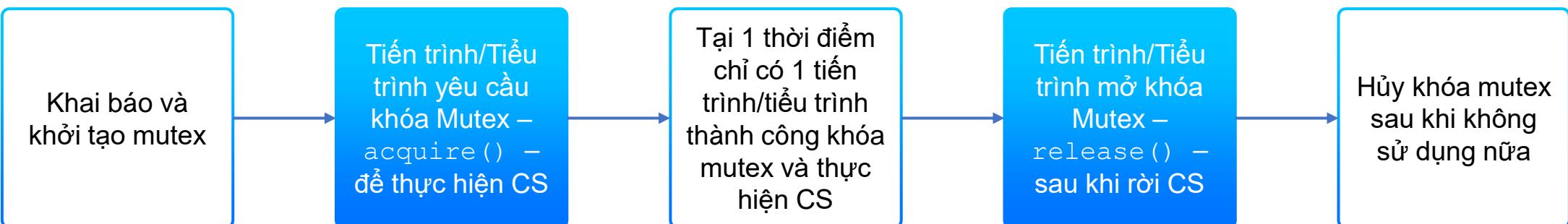


5.6.3. Cách sử dụng mutex locks

```
while (true) {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
}
```

Lưu ý:

- Mutex lock thường sẽ được khai báo toàn cục và được khởi tạo trong hàm main
- Cần phải xác định đúng vùng tranh chấp trước khi thực hiện các thao tác trên khóa mutex (`acquire` và `release`)





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 wait()	Định nghĩa thao tác signal()
<pre>wait(S) { while (S <= 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 **trừ semaphore đi 1**
- Được sử dụng khi muốn **sử dụng tài nguyên**

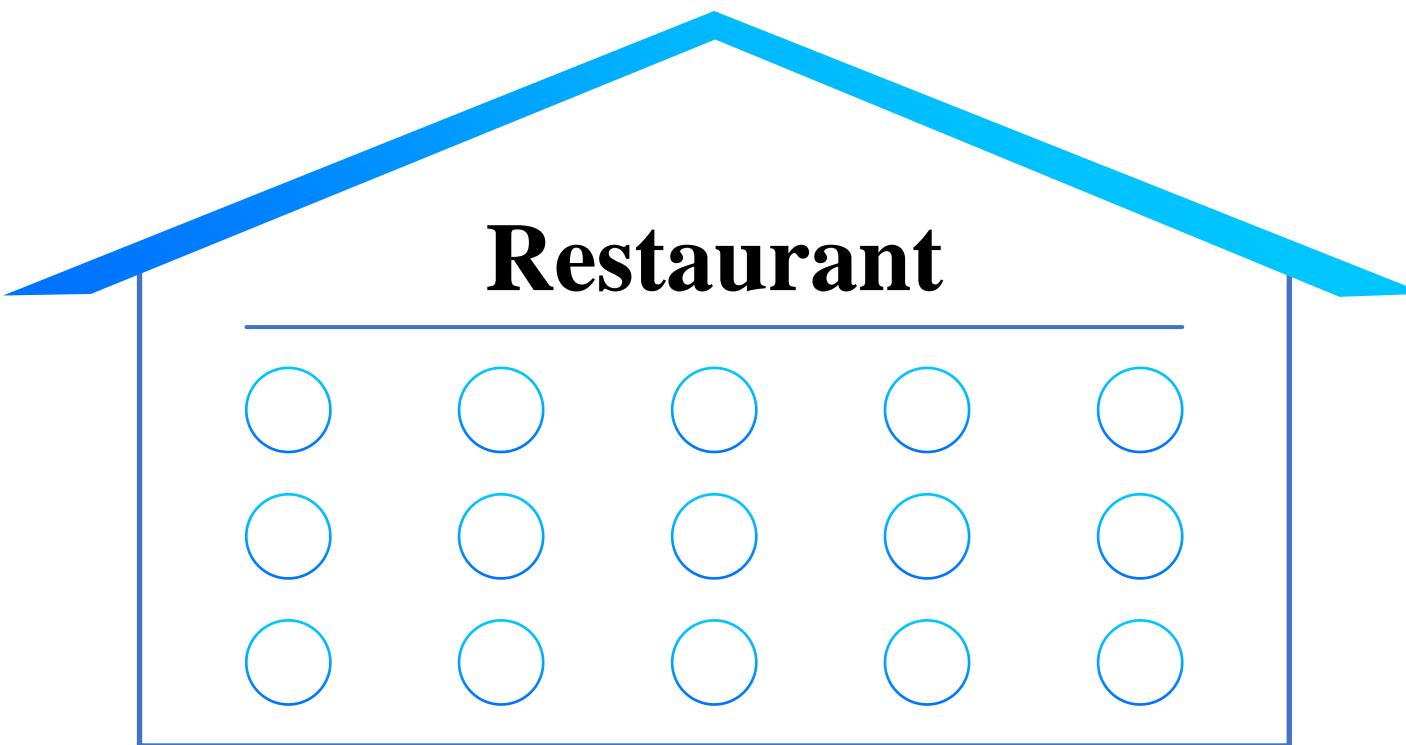
Định nghĩa thao tác signal()

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

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



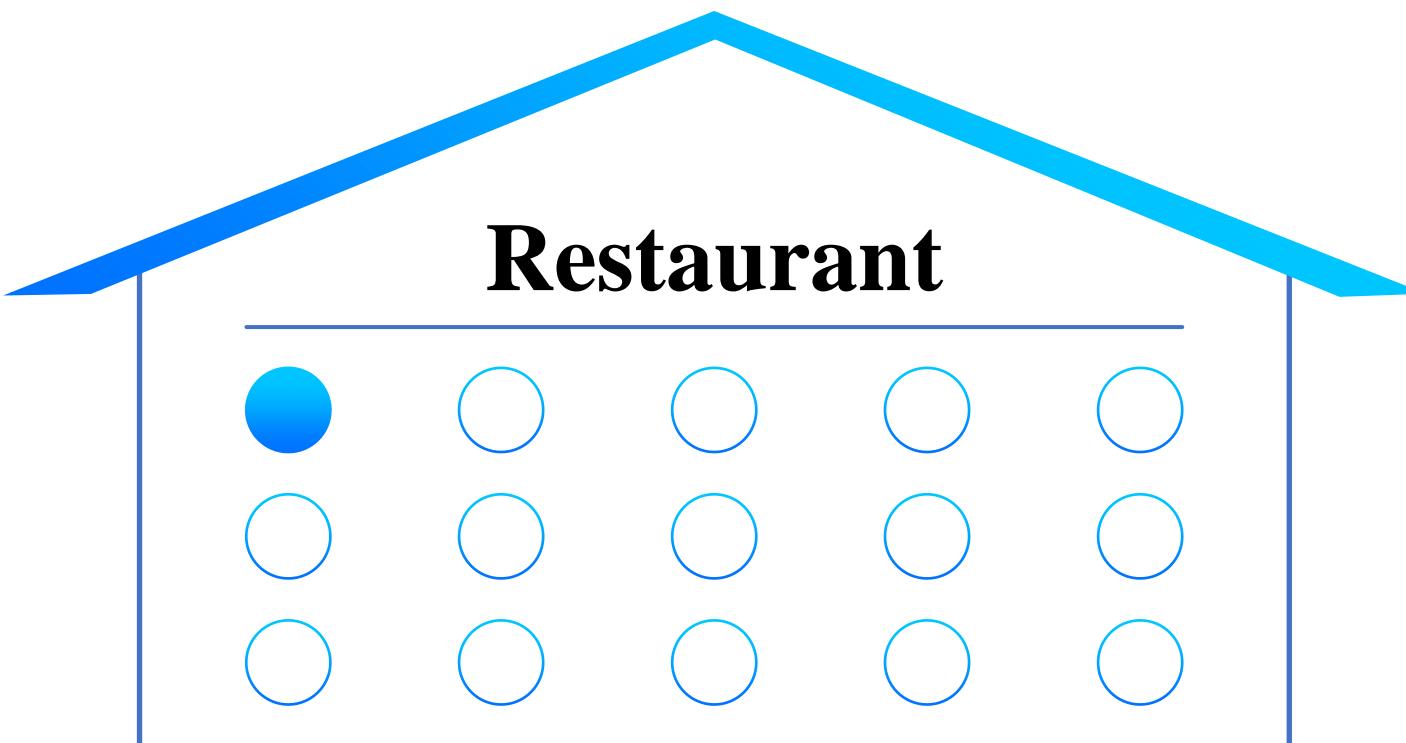
5.7.1. Định nghĩa semaphore



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



5.7.1. Định nghĩa semaphore



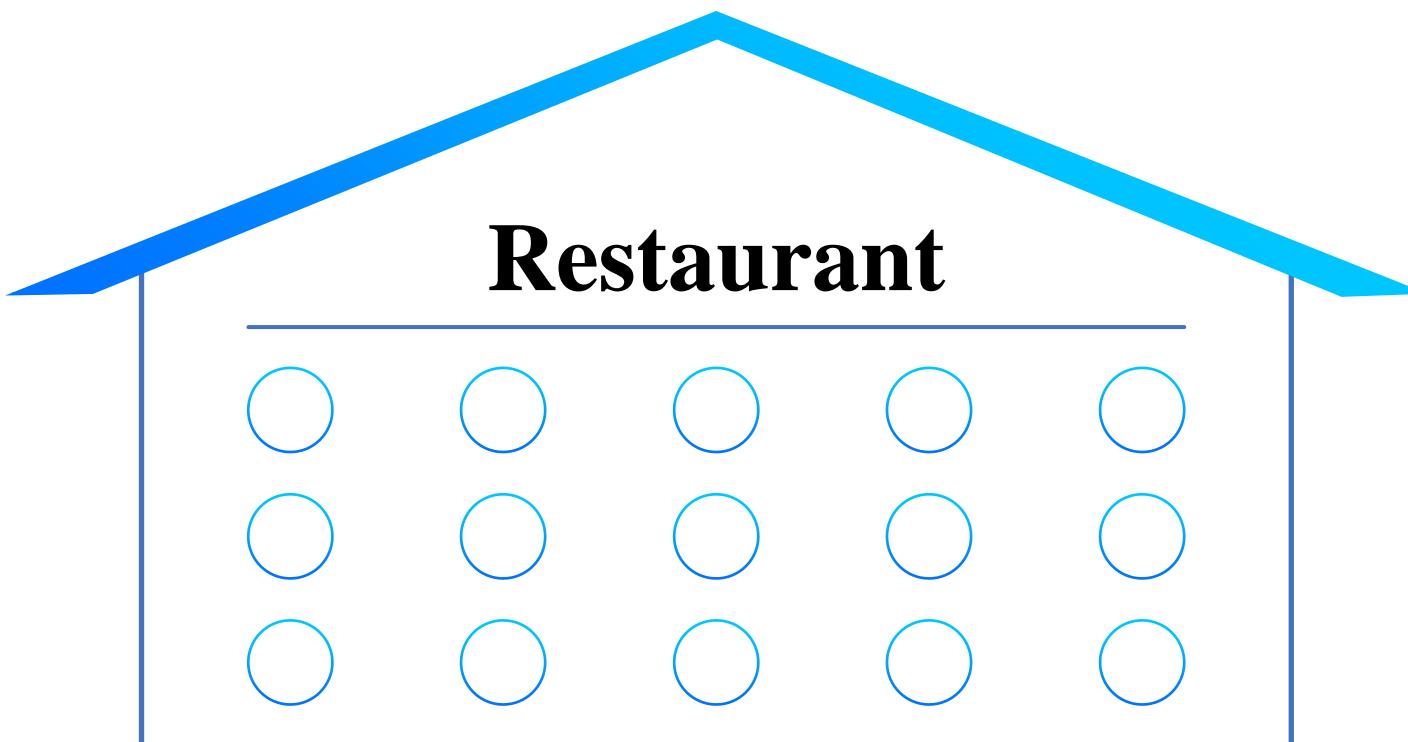
- semaphore **freeTable**
- Khởi tạo **freeTable = 15**

Khách P1 đến

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



5.7.1. Định nghĩa semaphore



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

Khách P1 đến

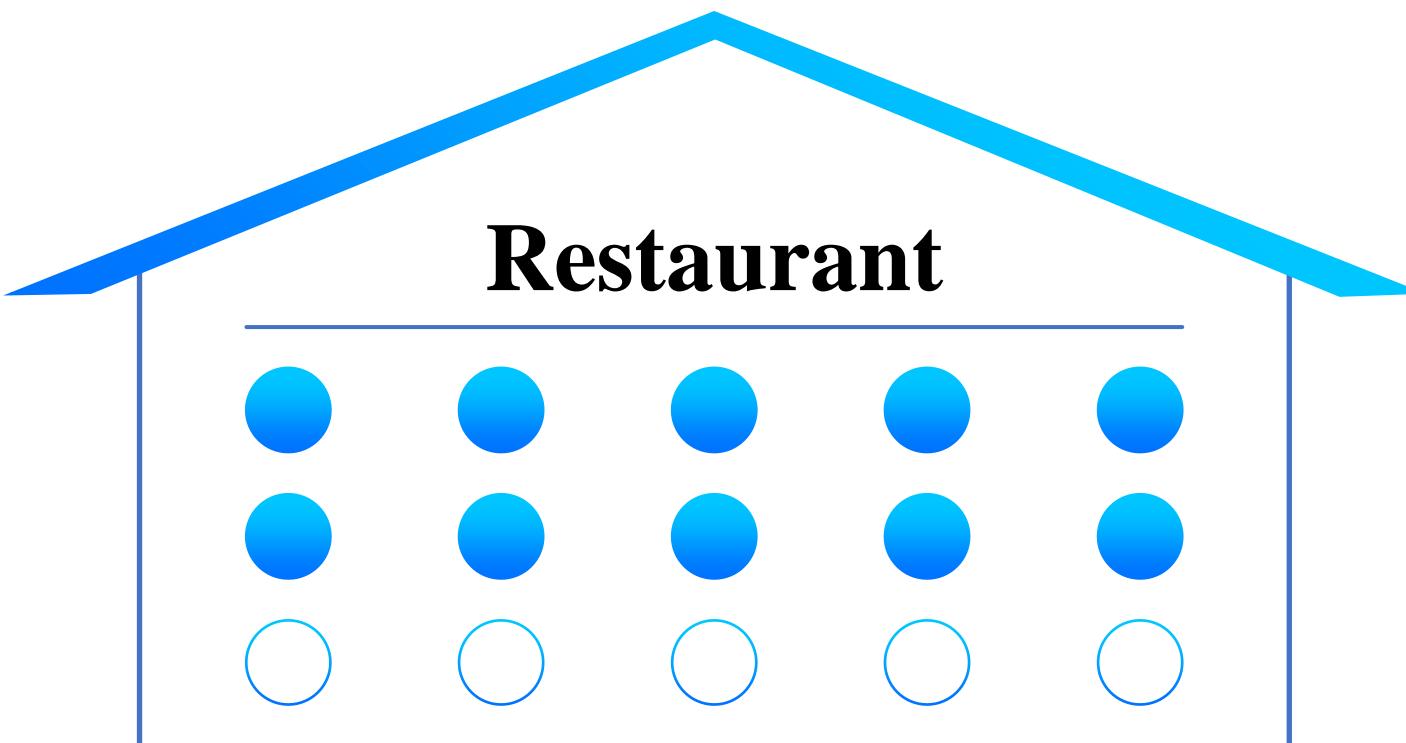
```
wait (freeTable);  
<dùng bữa>; //freeTable = 14
```

Khách P1 đi

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



5.7.1. Định nghĩa semaphore



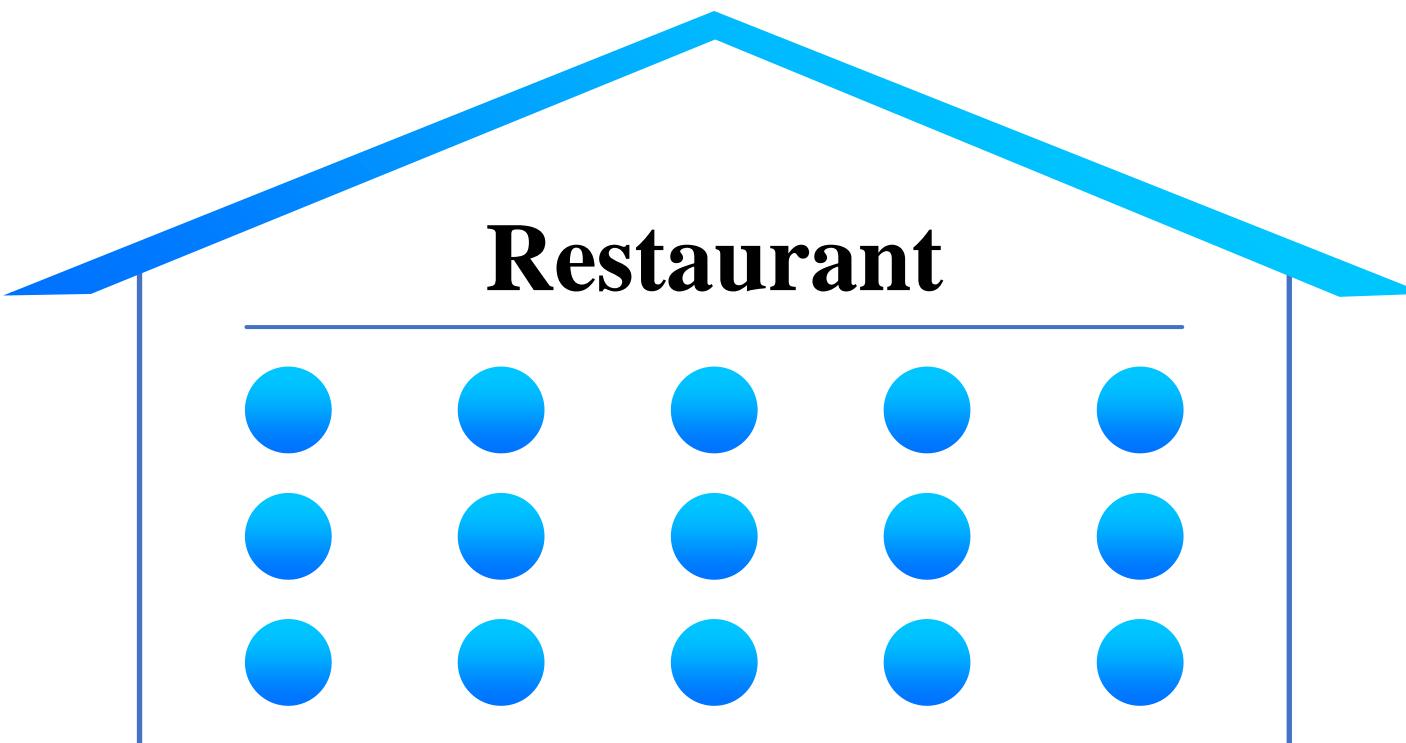
- semaphore `freeTable`
- Khởi tạo `freeTable = 15`

Có 10 khách đến thì:

`freeTable = ...`



5.7.1. Định nghĩa semaphore



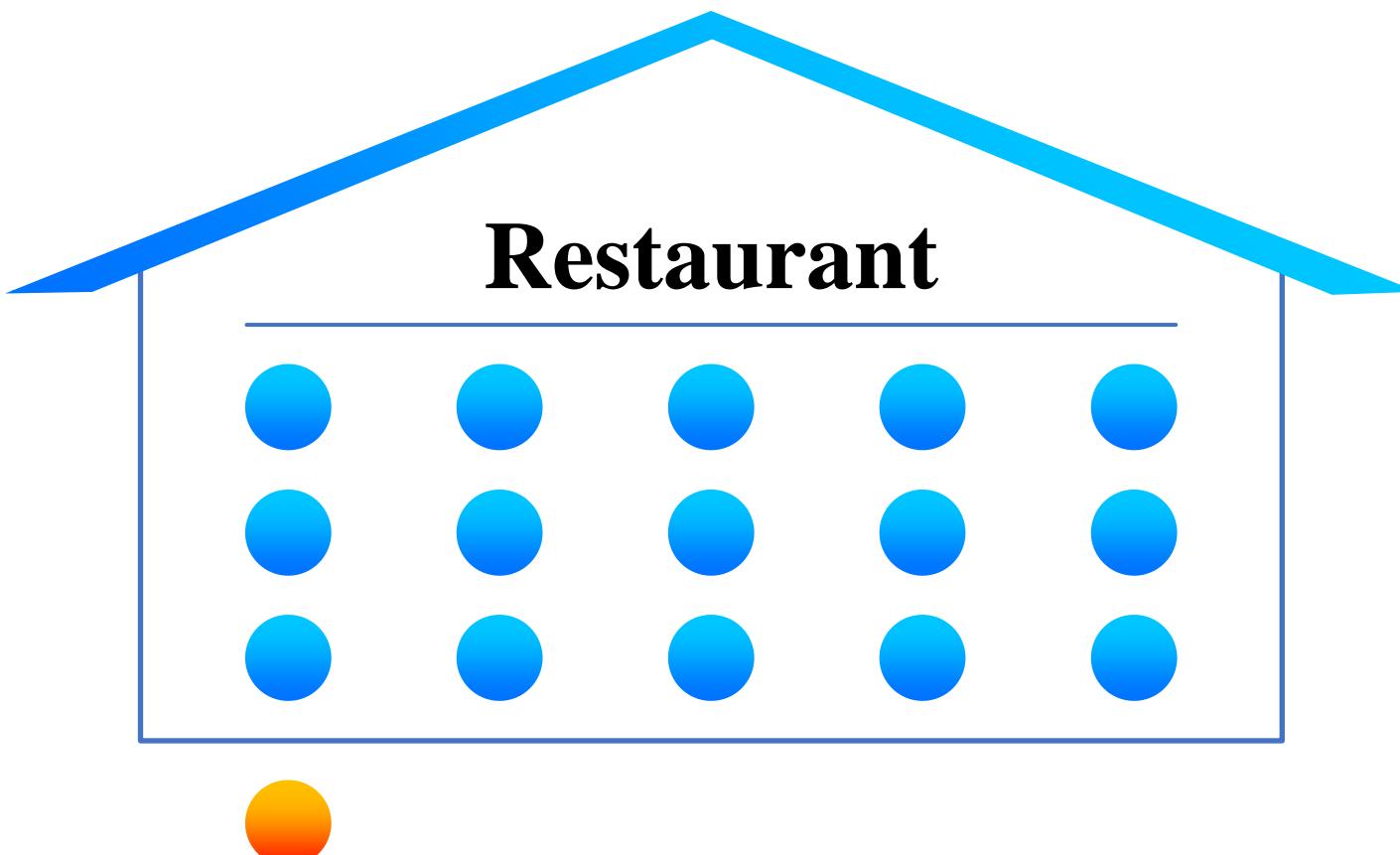
- semaphore `freeTable`
- Khởi tạo `freeTable = 15`

Có 15 khách đến thì:

`freeTable = 0`



5.7.1. Định nghĩa semaphore



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

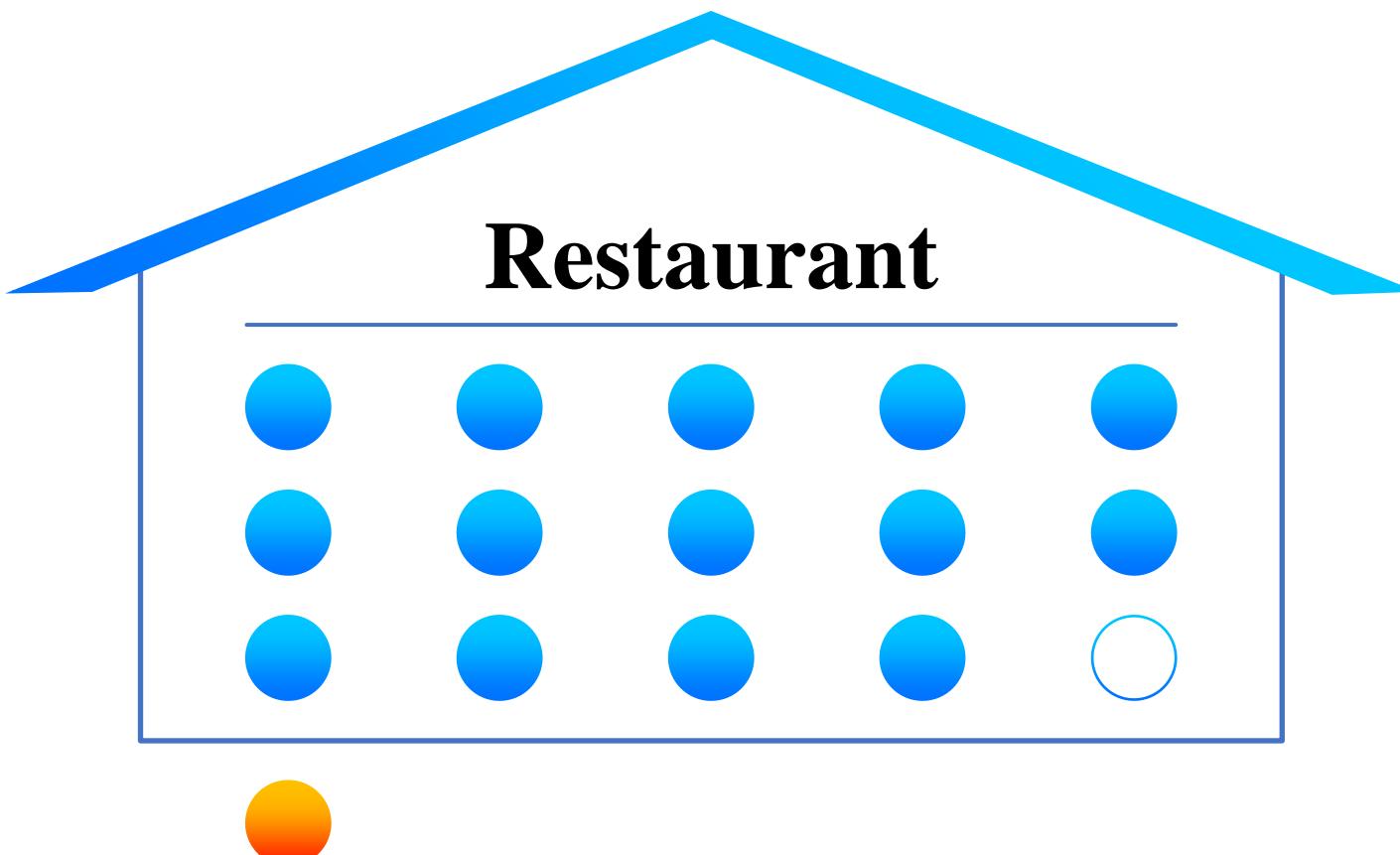
Khách P16 đến:

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

→ Có 1 khách phải chờ ở ngoài



5.7.1. Định nghĩa semaphore



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

Khách P16 đến:

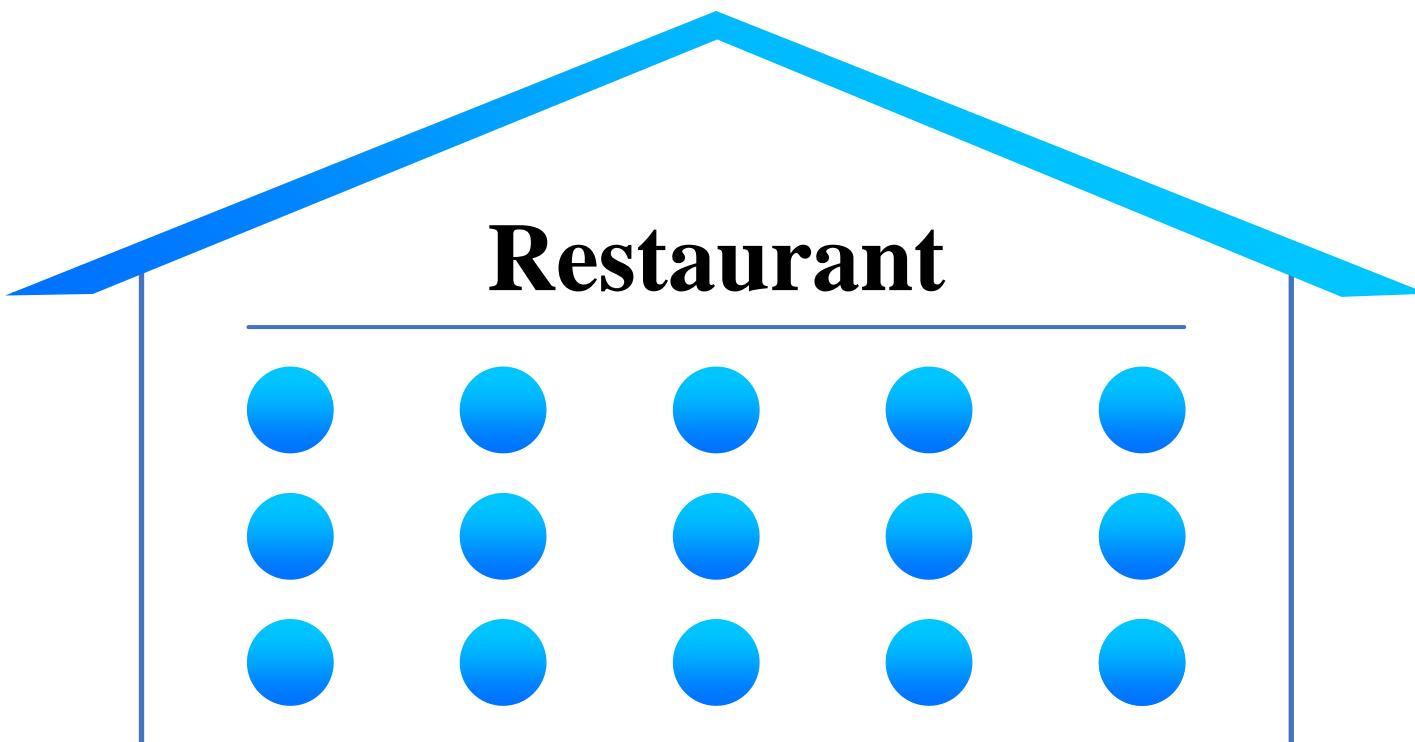
```
wait (freeTable);  
//freeTable = 1 → có thể vào  
<dùng bữa>;
```

Khách P15 đi

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



5.7.1. Định nghĩa semaphore



- 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
```



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 ngăn 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 thế 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.

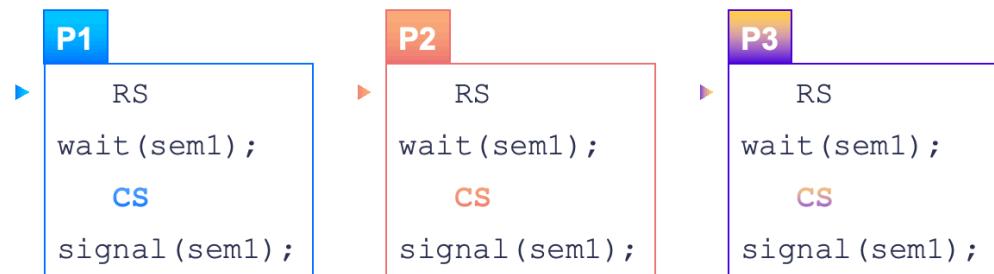
07.



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.

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

- ▶ P1
 - ▶ wait(S);
 - wait(Q);
 - ...
 - signal(S);
 - signal(Q);
- ▶ P2
 - ▶ wait(Q);
 - wait(S);
 - ...
 - signal(Q);
 - signal(S);

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

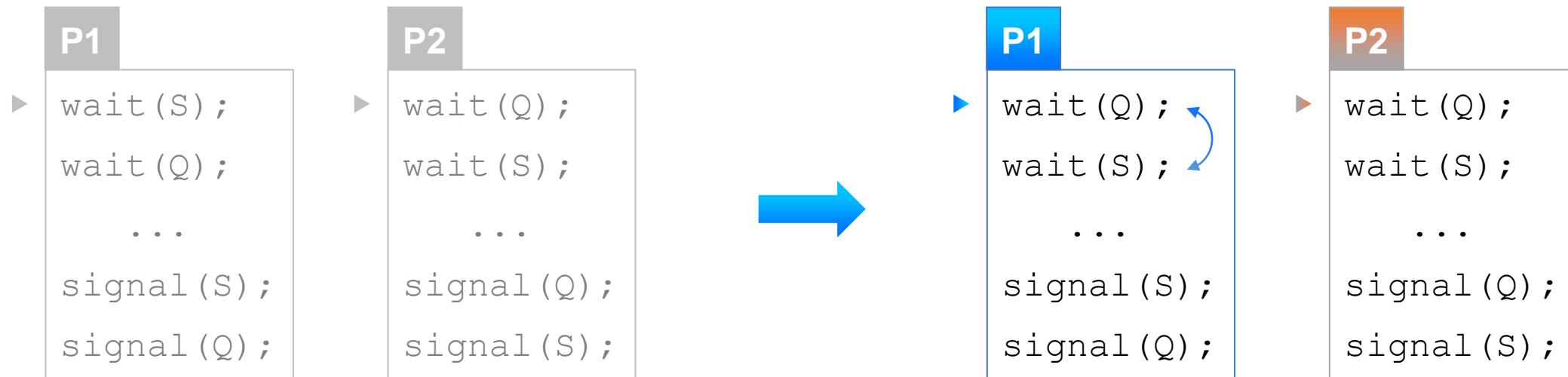
→ DEADLOCK



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

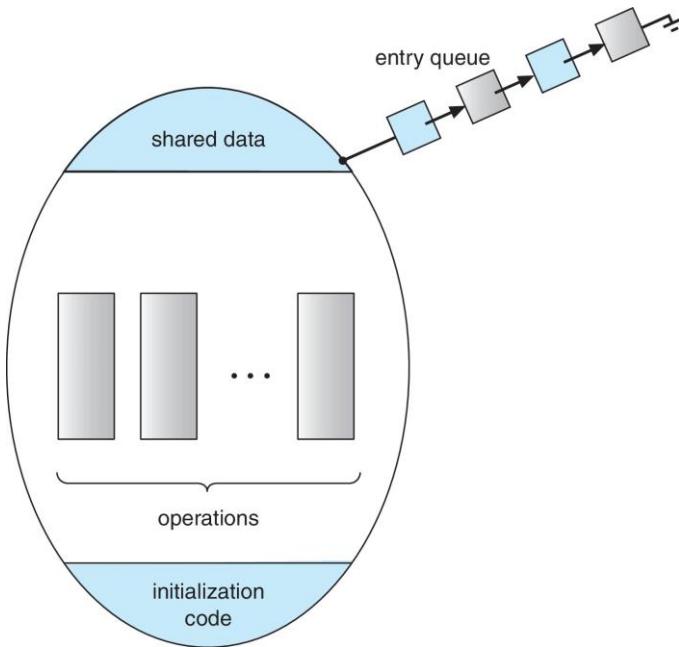
Mã giả của một monitor

```
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);

...

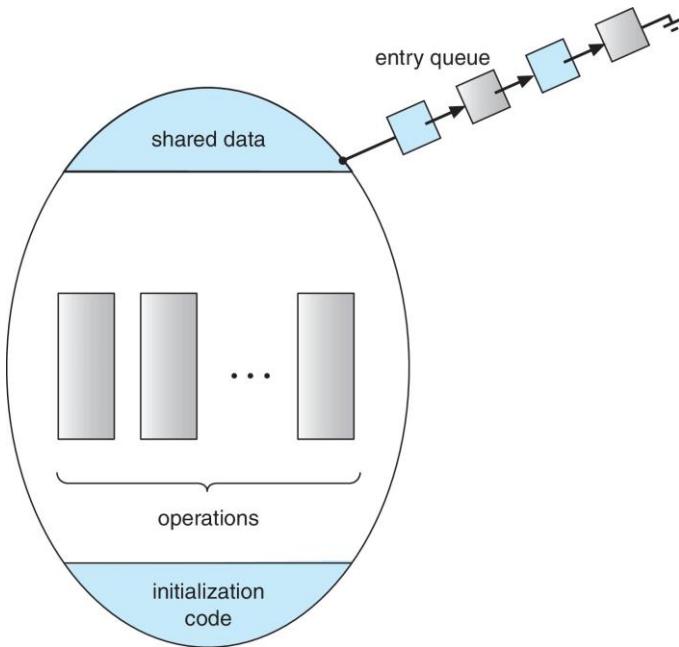
body of P;

...

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

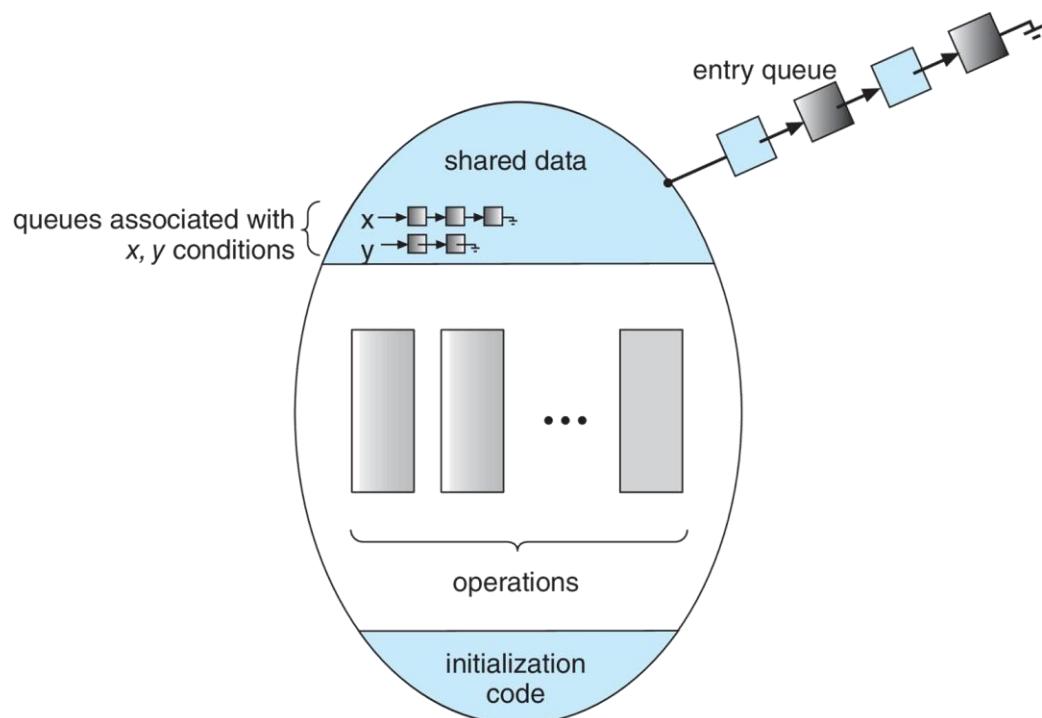
- Nhằm cho phép tiến trình đợi “trong monitor”, chỉ có thể được truy cập bên trong monitor.
- Khai báo:
`condition x, y;`
- Chỉ có thể thao tác lên condition variable bằng 02 thao tác:
 - `x.wait()` – tiến trình thực thi thao tác này sẽ bị *block* trên condition variable x cho đến khi thao tác `x.signal()`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



MONITOR

```
condition x; x = 0;
boolean done;
F1
    S1;
    done = true;
    x.signal()
F2
    if done == false
        x.wait()
    S2;
```



LIVENESS

Quá trình đồng bộ tiến trình có thể gây ra các lỗi nghiêm trọng trong việc lâm 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.



5.9 Liveness

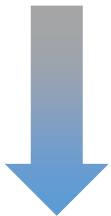
- Tiến trình có thể phải chờ vô thời hạn để cố gắng yêu cầu các công cụ đồng bộ như mutex hay semaphore → vi phạm tiêu chí **progress** và **bounded waiting**.
- **Liveness** là thuật ngữ để chỉ một tập các đặc điểm mà hệ thống phải thỏa mãn để đảm bảo tiến trình thực sự chạy.
- Chờ đợi không giới hạn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



Tóm tắt lại nội dung buổi học

- Mutex locks
- Semaphore
- Monitor
- Liveness



THẢO LUẬN

