



HỆ ĐIỀU HÀNH

CHƯƠNG 5: ĐỒNG BỘ TIẾN TRÌNH (PHẦN 1)

Trong chương này, các vấn đề về đồng bộ tiến trình sẽ được thảo luận và làm rõ bao gồm: vì sao cần phải đồng bộ, các tiêu chuẩn về lời giải cho bài toán đồng bộ và các kỹ thuật đồng bộ.



MỤC TIÊU

1. Trình bày được khái niệm race condition và mô tả được vấn đề vùng tranh chấp
2. Mô tả được các yêu cầu dành cho lời giải của bài toán vùng tranh chấp
3. Liệt kê được các giải pháp đồng bộ sử dụng phần mềm và vấn đề của chúng
4. Trình bày được giải pháp đồng bộ dựa trên phần cứng bao gồm `test_and_set`, `compare_and_swap`, và biến đơn nguyên
5. Diễn tả được cơ chế hoạt động của mutex lock trong việc giải quyết bài toán vùng tranh chấp



NỘI DUNG

1. Race condition
2. Vấn đề vùng tranh chấp
3. Lời giải cho vấn đề vùng tranh chấp
4. Các giải pháp phần mềm
5. Giải pháp phần cứng
6. Mutex locks



RACE CONDITION

5.1.1 Bài toán Producer vs. Consumer

Bài toán Producer vs. Consumer mô tả về 02 tiến trình bao gồm: “Sản xuất” và “Bán hàng”. Nếu gọi biến `count` mô tả số lượng hàng hóa, thì tiến trình “Sản xuất” sẽ làm tăng giá trị của `count`; ngược lại, tiến trình “Bán hàng” sẽ làm giảm giá trị này. Khi “Sản xuất” và “Bán hàng” diễn ra đồng thời, biến `count` sẽ chịu tác động của việc tăng và giảm cùng lúc. Khi đó, liệu rằng giá trị của `count` có còn đúng với logic?

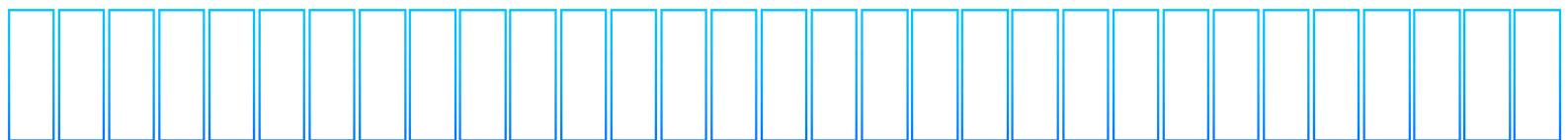
01.



5.1.1. Bài toán Producer vs. Consumer

- Gồm 02 tiến trình diễn ra đồng thời với nhau:
 - Producer: liên tục tạo ra hàng hóa → tăng biến count
 - Consumer: liên tục bán hàng → giảm biến count

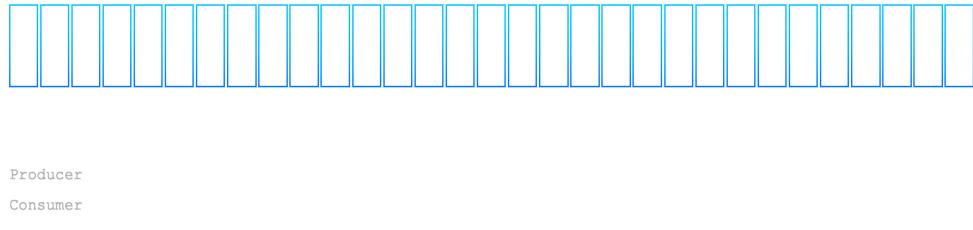
count = 0



Producer
Consumer



5.1.1. Bài toán Producer vs. Consumer

- Gồm 02 tiến trình diễn ra đồng thời với nhau:
 - **Producer:** liên tục tạo ra hàng hóa → tăng biến count
 - **Consumer:** liên tục bán hàng → giảm biến count
- count = 0
- 
- Thông thường các tiến trình đều sẽ được đặt trong vòng while(1) để thực thi liên tục.
 - Khi các tiến trình thực thi đồng thời, các dữ kiện sau sẽ **KHÔNG** thể xác định được:
 - Tiến trình nào thực thi trước?
 - Tiến trình nào thực thi lâu hơn (do giải thuật định thời CPU)?
 - Tiến trình sẽ hết quantum time khi nào?



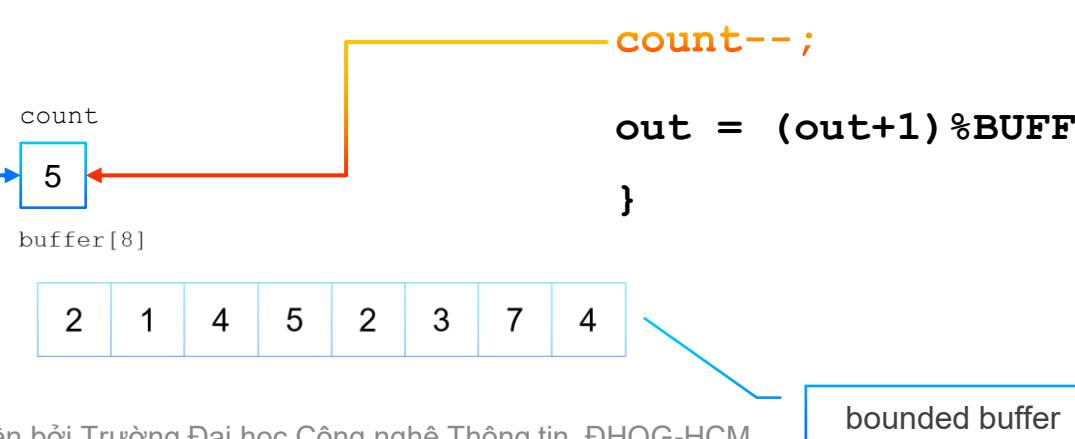
5.1.1. Bài toán Producer vs. Consumer

Producer

```
item nextProduce;  
  
while(1){  
  
    while(count == BUFFER_SIZE);  
        /*khong lam gi*/  
  
    buffer[in] = nextProduce;  
  
    count++;  
  
    in = (in+1)%BUFFER_SIZE;  
}
```

Consumer

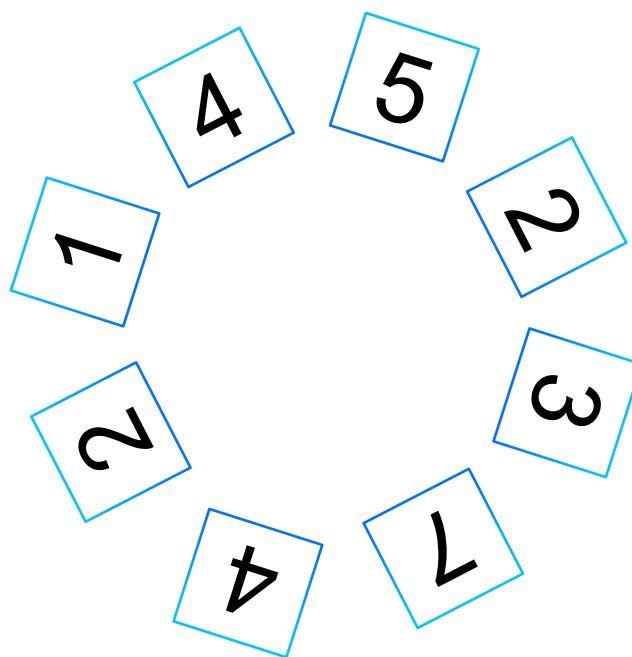
```
item nextConsumer;  
  
while(1){  
  
    while(count == 0);  
        /*khong lam gi*/  
  
    nextConsumer = buffer[out];  
  
    count--;  
  
    out = (out+1)%BUFFER_SIZE;  
}
```



`buffer[8]`

2	1	4	5	2	3	7	4
---	---	---	---	---	---	---	---

buffer[8]





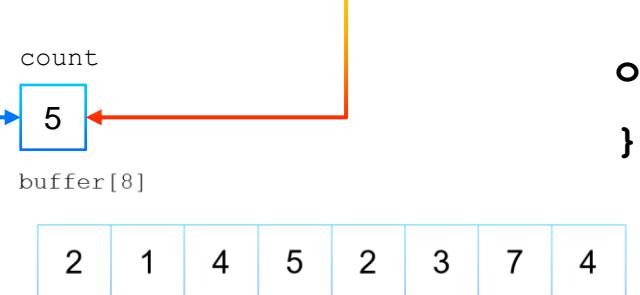
5.1.1. Bài toán Producer vs. Consumer

Producer

```
item nextProduce;  
  
while(1){  
  
    while(count == BUFFER_SIZE);  
        /*khong lam gi*/  
  
    buffer[in] = nextProduce;  
  
    count++;  
  
    in = (in+1)%BUFFER_SIZE;  
}
```

Consumer

```
item nextConsumer;  
  
while(1){  
  
    while(count == 0);  
        /*khong lam gi*/  
  
    nextConsumer = buffer[out];  
  
    count--;  
  
    out = (out+1)%BUFFER_SIZE;  
}
```





5.1.1. Bài toán Producer vs. Consumer

count++

Load: **reg1 = count**

Inc: **reg1 = reg1 + 1**

Store: **count = reg1**

count--

Load: **reg2 = count**

Dec: **reg2 = reg2 - 1**

Store: **count = reg2**



**reg1, reg2 là các thanh ghi*

Giả sử **count = 5**, hãy cho biết giá trị của count với 02 trường hợp dưới đây?

Quantum = 3 cycles

T1 Producer: reg1 = count	(reg1 = 5)
T2 Producer: reg1 = reg1 + 1	(reg1 = 6)
T3 Producer: count = reg1	(count = 6)
T4 Consumer: reg2 = count	(reg2 = 6)
T5 Consumer: reg2 = reg2 - 1	(reg2 = 5)
T6 Consumer: count = reg2	(count = 5)

Quantum = 2 cycles

T1 Producer: reg1 = count	(reg1 = 5)
T2 Producer: reg1 = reg1 + 1	(reg1 = 6)
T3 Consumer: reg2 = count	(reg2 = 5)
T4 Consumer: reg2 = reg2 - 1	(reg2 = 4)
T5 Producer: count = reg1	(count = 6)
T6 Consumer: count = reg2	(count = 4)

Quá trình
thực thi của 2
lệnh
count++
và
count--
bị đan xen
vào nhau

Giá trị không chính xác



RACE CONDITION

5.1.2 Bài toán Cấp phát PID

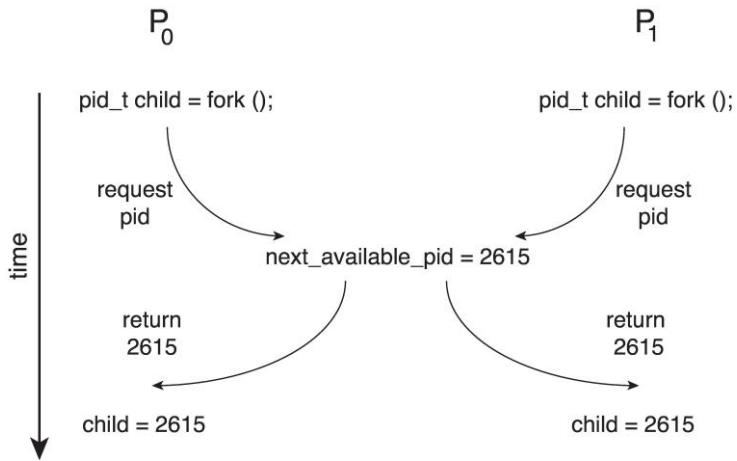
Khi một tiến trình P gọi hàm fork(), một tiến trình con sẽ được tạo ra, hệ điều hành sẽ cấp cho tiến trình con một số định danh gọi là PID. Như vậy nếu có 2 tiến trình P0 và P1 cùng gọi hàm fork() đồng thời với nhau thì chuyện gì sẽ xảy ra?

01.



5.1.2. Bài toán cấp phát PID

- 02 tiến trình P0 và P1 đang tạo tiến trình con bằng cách gọi hàm `fork()`.
- Biến `next_available_pid()` được kernel sử dụng để tạo ra PID cho tiến trình mới.
- Tiến trình con của P0 và P1 đồng thời yêu cầu PID và nhận được kết quả như nhau.



- Cần có cơ chế để ngăn P0 và P1 truy cập biến `next_available_pid` cùng lúc, để tránh tình trạng một PID được cấp cho 2 tiến trình.



RACE CONDITION

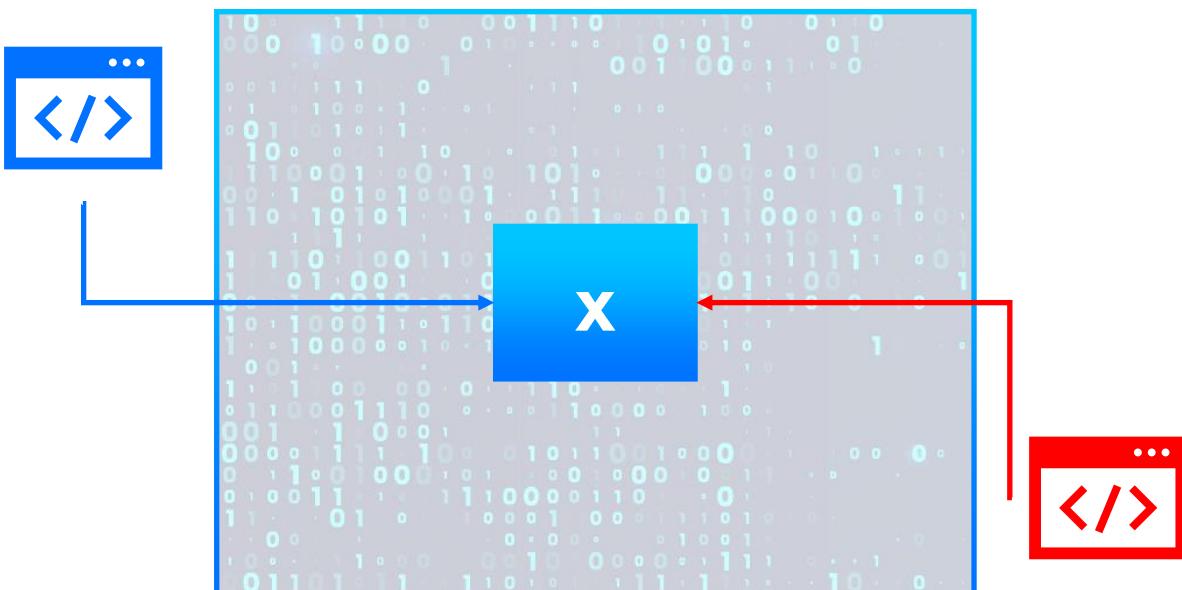
5.1.3. Race condition

01.



5.1.3. Race condition

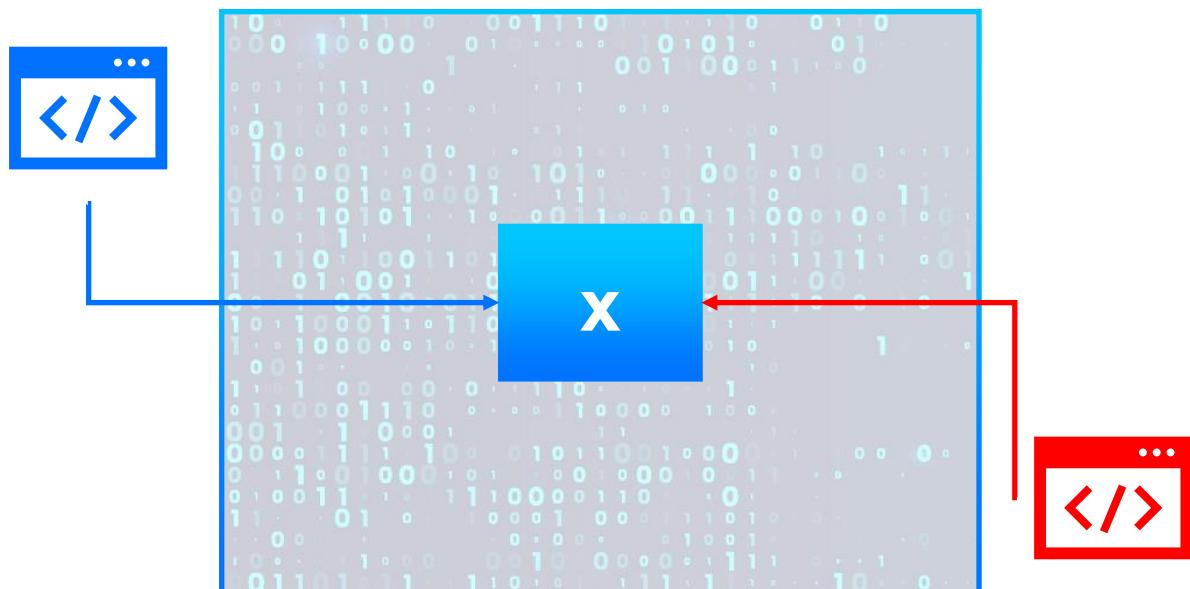
- **Race condition** là hiện tượng xảy ra khi các tiến trình cùng truy cập đồng thời vào dữ liệu được chia sẻ. Kết quả cuối cùng sẽ phụ thuộc vào thứ tự thực thi của các tiến trình đang chạy đồng thời với nhau.
- Trong bài toán Producer vs. Consumer dữ liệu được chia sẻ là biến count bị tác động đồng thời bởi cả 02 tiến trình Producer và Consumer. Trong bài toán cấp phát PID, dữ liệu được chia sẻ là biến next_available_pid bị tranh giành bởi tiến trình thực thi đồng thời là P0 và P1.





5.1.3. Race condition

- Race condition có thể dẫn đến việc dữ liệu bị sai và không nhất quán (inconsistency).
- Để dữ liệu chia sẻ được nhất quán, cần bảo đảm sao cho tại mỗi thời điểm chỉ có một tiến trình được thao tác lên dữ liệu chia sẻ. Do đó, cần có cơ chế đồng bộ hoạt động của các tiến trình này.





VĂN ĐỀ VÙNG TRANH CHẤP

Vùng tranh chấp (hay còn gọi là critical section) là vùng code mà ở đó các tiến trình thực hiện tác động lên dữ liệu được chia sẻ.

02.



5.2 Vấn đề vùng tranh chấp

- Xem xét một hệ thống có n tiến trình $\{P_0, P_1, \dots, P_{n-1}\}$
- Mỗi tiến trình có một **vùng tranh chấp** là một **đoạn code**:
 - Thực hiện việc thay đổi giá trị của dữ liệu được chia sẻ (có thể là các biến, bảng dữ liệu, file,...)
 - Khi một tiến trình đang thực hiện vùng tranh chấp của mình thì các tiến trình khác **KHÔNG** được thực hiện vùng tranh chấp của chúng.
- Vấn đề vùng tranh chấp chính là thiết kế cách thức xử lý các vấn đề trên.



5.2 Vấn đề vùng tranh chấp

Mỗi tiến trình phải yêu cầu để được phép tiến vào vùng tranh chấp của mình thông qua entry section, sau đó thực thi vùng tranh chấp – **critical section** - rồi tiến đến exit section, và sau cùng là thực thi remainder section.

```
while(1) {
    entry section
    critical section
    exit section
    remainder section
}
```



LỜI GIẢI CHO BÀI TOÁN VÙNG TRANH CHẤP

5.3.1. Yêu cầu dành cho lời giải

Ván đè vùng tranh chấp là một ván đè phức tạp, do đó, ta cần có những yêu cầu cụ thể để đảm bảo rằng lời giải cho bài toán này có thể đáp ứng được các tiêu chuẩn như các tiến trình đều phải được thực thi, không bị xảy ra hiện tượng đói, dữ liệu không bị thiếu nhát quán hay không để xảy ra tình trạng deadlock.

03.



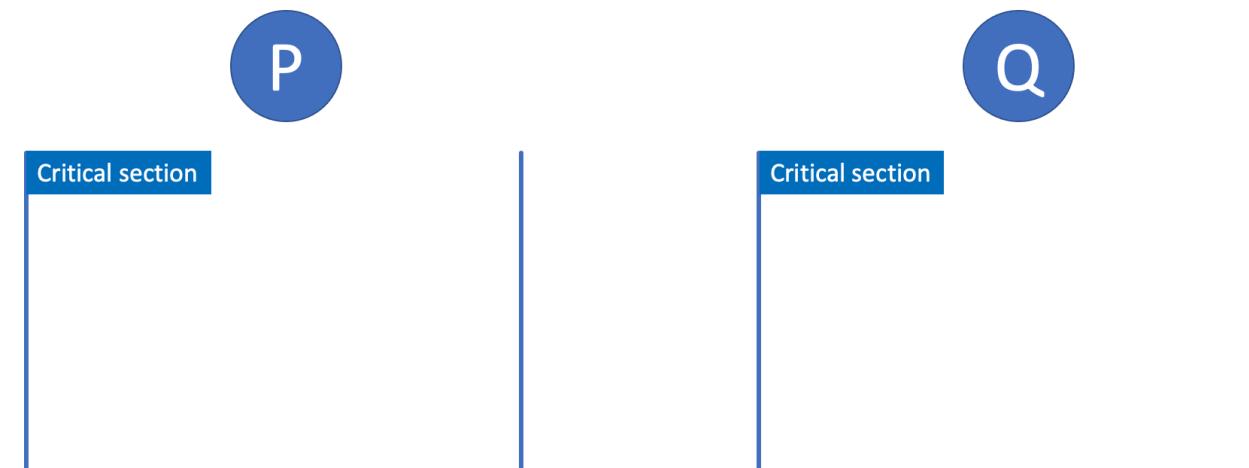
5.3.1. Yêu cầu dành cho lời giải

- Lời giải cho bài toán vùng tranh chấp phải đảm bảo 03 yêu cầu sau:
 - (1) **Mutual exclusion** (loại trừ tương hỗ): Khi một tiến trình P đang thực thi trong vùng tranh chấp (CS) của nó thì không có tiến trình Q nào khác đang thực thi trong CS của Q.
 - (2) **Progress** (tiến triển): Một tiến trình tạm dừng bên ngoài vùng tranh chấp không được ngăn cản các tiến trình khác vào vùng tranh chấp.
 - (3) **Bounded waiting** (chờ đợi giới hạn): Mỗi tiến trình chỉ phải chờ để được vào vùng tranh chấp trong một khoảng thời gian có hạn định nào đó. Không xảy ra tình trạng đói tài nguyên (starvation).



5.3.1. Yêu cầu dành cho lời giải

(1) **Mutual exclusion** (loại trừ tương hỗ): Khi một tiến trình P đang thực thi trong vùng tranh chấp (CS) của nó thì không có tiến trình Q nào khác đang thực thi trong CS của Q.

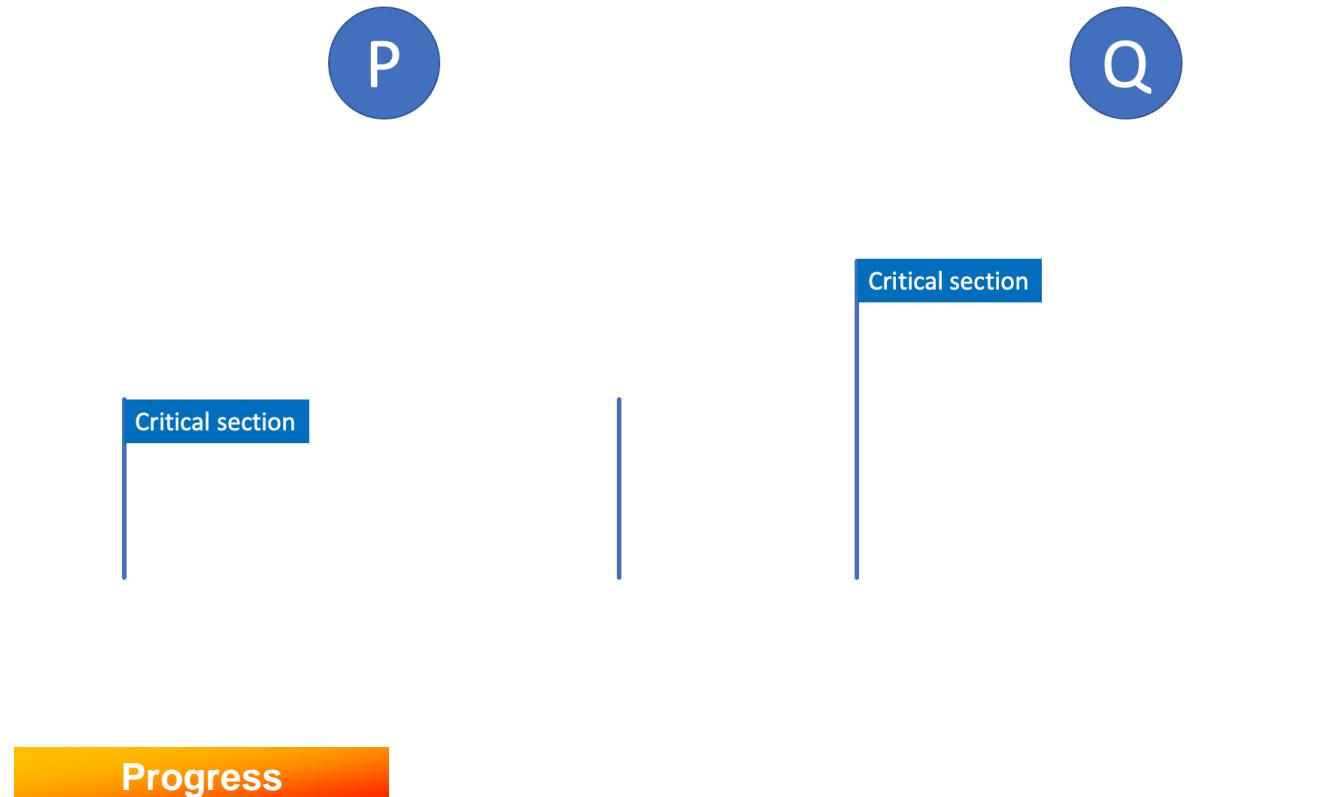


Mutual exclusion



5.3.1. Yêu cầu dành cho lời giải

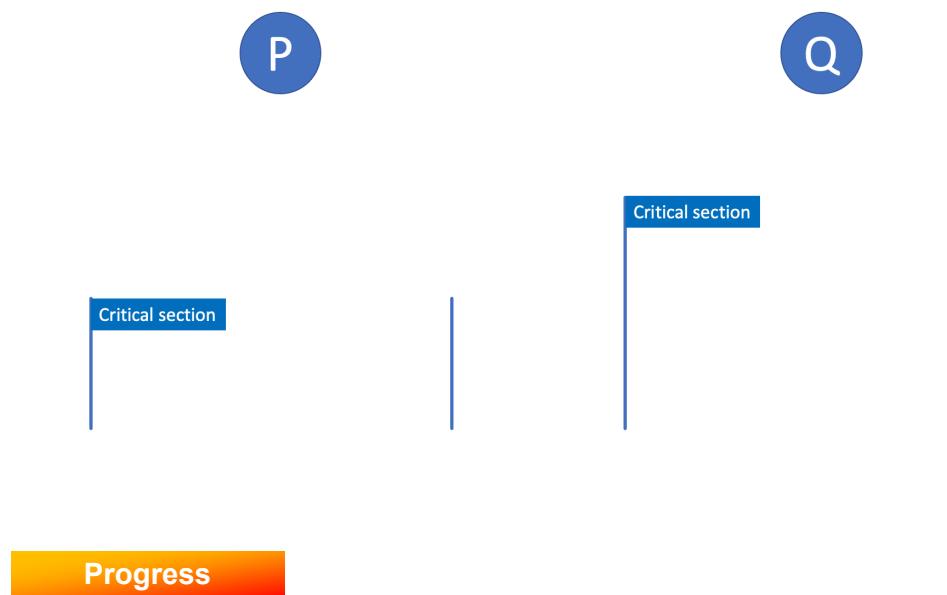
(2) **Progress** (tiến triển): Một tiến trình tạm dừng bên ngoài vùng tranh chấp không được ngăn cản các tiến trình khác vào vùng tranh chấp.



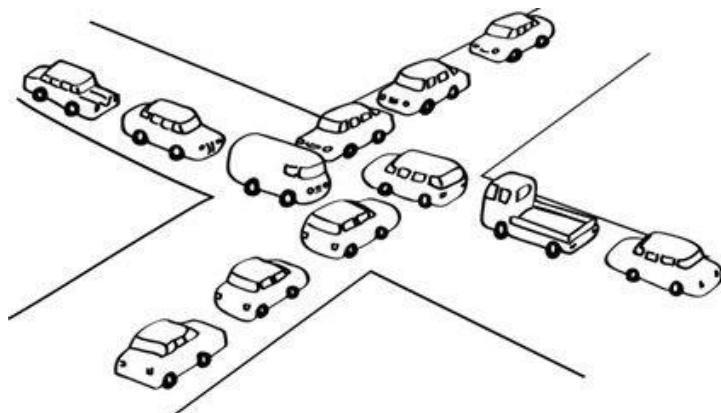


5.3.1. Yêu cầu dành cho lời giải

(2) **Progress** (tiến triển): Một tiến trình tạm dừng bên ngoài vùng tranh chấp không được ngăn cản các tiến trình khác vào vùng tranh chấp.



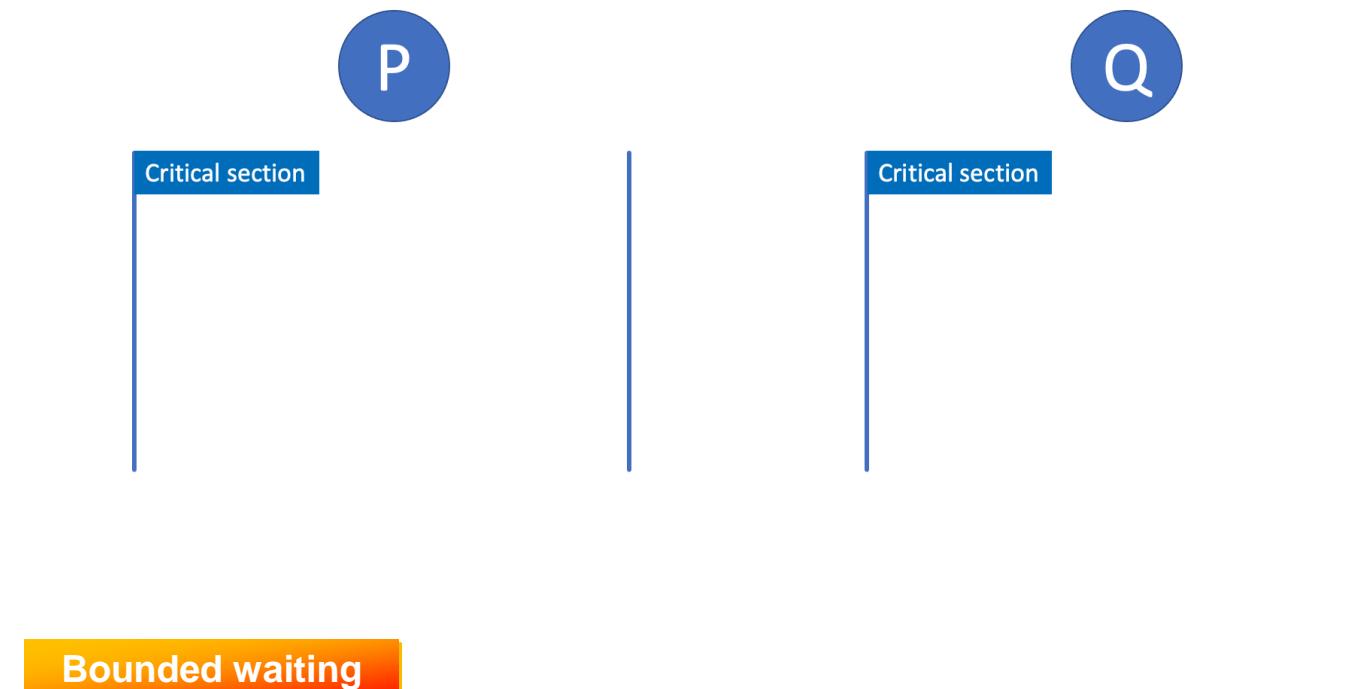
Hiện tượng tiến trình P chờ điều kiện từ tiến trình Q khi tiến trình Q cũng đang chờ điều kiện từ tiến trình P để được vào vùng tranh chấp được gọi là **deadlock**.





5.3.1. Yêu cầu dành cho lời giải

(3) **Bounded waiting** (chờ đợi giới hạn): Mỗi tiến trình chỉ phải chờ để được vào vùng tranh chấp trong một khoảng thời gian có hạn định nào đó. Không xảy ra tình trạng đói tài nguyên (starvation).





LỜI GIẢI CHO BÀI TOÁN VÙNG TRANH CHẤP

5.3.2. Phân loại giải pháp

Có nhiều hướng tiếp cận cho lời giải của bài toán vùng tranh chấp. Tùy thuộc vào tiêu chí mà chúng ta có thể phân loại thành các giải pháp phần mềm/phần cứng, các giải pháp đòi hỏi/không đòi hỏi sự hỗ trợ của hệ điều hành, các giải pháp yêu cầu sự chờ đợi của tiến trình,...

03.



5.3.2. Phân loại giải pháp

Phân loại theo sự hỗ trợ của phần cứng

Giải pháp phần mềm

- Không cần sự hỗ trợ từ phần cứng, có thể được thực hiện thông qua các kỹ thuật lập trình.
- Ví dụ: giải thuật Peterson, giải thuật Bakery, giải thuật Dekker.

Giải pháp dựa trên phần cứng

- Cần sự hỗ trợ của một vài phần cứng đặc biệt, ví dụ cung cấp cơ chế đơn nguyên cho một vài chỉ thị/lệnh nhất định.
- Ví dụ: Test & Set, Compare & Swap.



5.3.2. Phân loại giải pháp

Phân loại theo sự hỗ trợ của hệ điều hành

Busy waiting

- Không cần sự hỗ trợ của hệ điều hành.
- Sử dụng kỹ thuật lập trình để tiến trình/tiểu trình phải chờ đợi (trong khi liên tục kiểm tra điều kiện) để được vào vùng tranh chấp.

Sleep & Wake up

- Cần hệ điều hành cung cấp cơ chế (qua system call) để:
 - *Tạm dừng (block) tiến trình*: đưa tiến trình gọi lệnh này vào trạng thái ngủ (**sleep**) nếu không được vào vùng tranh chấp.
 - *Đánh thức tiến trình*: khi một tiến trình ra khỏi vùng tranh chấp, tiến trình này có thể “đánh thức” (**wake up**) một tiến trình khác đang ngủ để tiến trình đó vào vùng tranh chấp.



LỜI GIẢI CHO BÀI TOÁN VÙNG TRANH CHẤP

5.3.3. Giải pháp cấm ngắt

Vô hiệu hóa (disable) ngắt sau khi tiến trình vào vùng tranh chấp và bật/kích hoạt lại (re-enable) nó trước khi rời khỏi vùng tranh chấp có phải là một giải pháp phù hợp?

03.



5.3.3. Cấm ngắt

- Entry section: vô hiệu hóa ngắt
- Exit section: kích hoạt ngắt
- Liệu rằng giải pháp này có thể giải quyết được bài toán?
 - Chuyện gì sẽ xảy ra nếu vùng tranh chấp là đoạn code chạy trong vòng 1 giờ?
 - Liệu có tiến trình nào bị đói không?
 - Nếu có 2 CPUs cùng chạy thì sao?



CÁC GIẢI PHÁP PHẦN MỀM

5.4.1. Giải pháp phần mềm 1

Ý tưởng của giải pháp này sử dụng biến một biến `turn` để kiểm tra xem tiến trình tới lượt thực hiện của tiến trình nào với sự hỗ trợ của 2 thao tác đơn nguyên là `load` và `store`.

04.

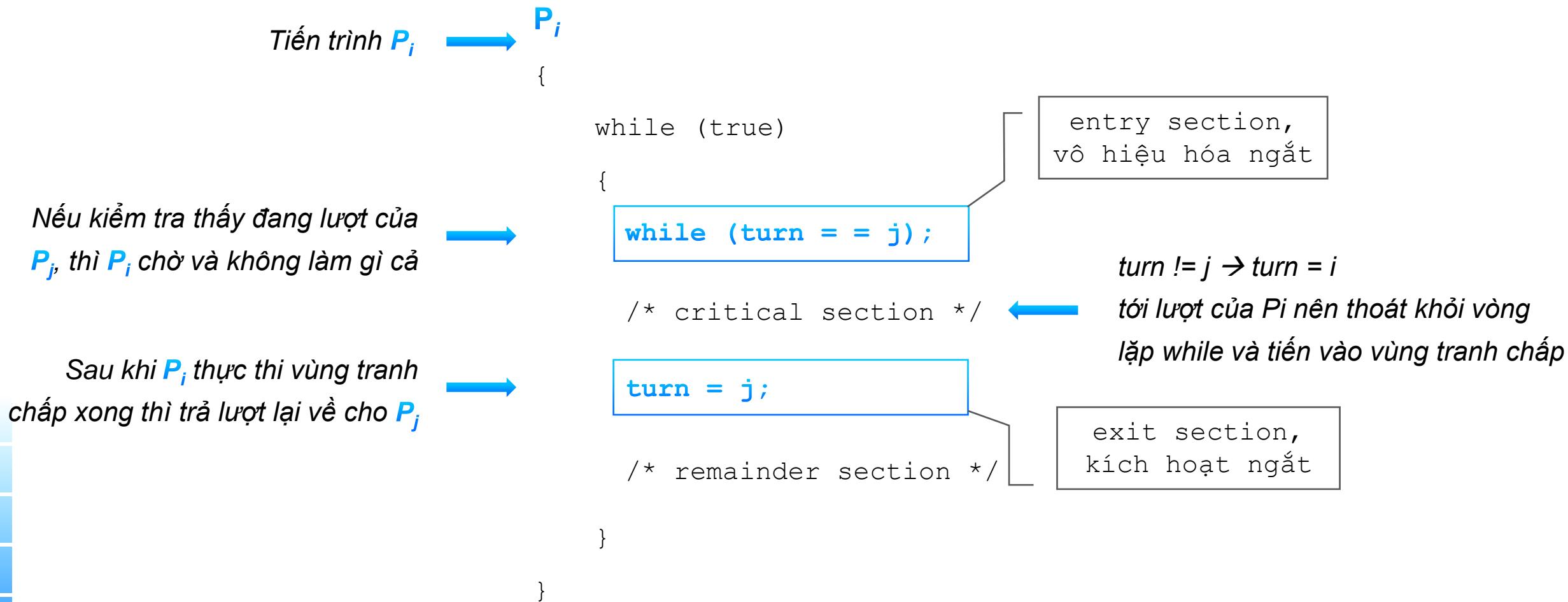


5.4.1. Giải pháp phần mềm 1

- Giải pháp dành cho 2 tiến trình.
- Giả sử 2 lệnh hợp ngũ load và store là 2 thao tác **đơn nguyên** (không thể bị cắt ngang).
- 2 tiến trình cùng chia sẻ một biến turn
 - int turn;
- Biến turn có tác dụng chỉ ra tiến trình nào tới lượt để vào vùng tranh chấp.
- Giá trị của turn sẽ được khởi tạo là *i*.



5.4.1. Giải pháp phần mềm 1





5.4.1. Giải pháp phần mềm 1

- Mutual exclusion được đảm bảo:

P_i chỉ được phép vào vùng tranh chấp khi:

turn = i

và turn không thể vừa bằng i , vừa bằng j được (nếu $i = 0$ và $j = 1$
thì turn không thể vừa bằng 0, vừa bằng 1)

- Kiểm tra Progress và Bounded waiting?

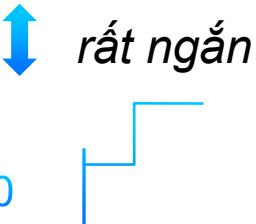


5.4.1. Giải pháp phần mềm 1

Kiểm tra Progress → Không đảm bảo

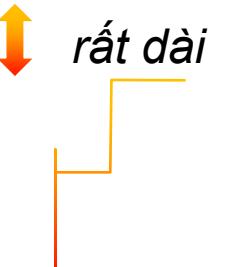
```
P0
{
    while (true
    {
        while (turn == 1);
        /* critical section */
        turn = 1;
        /* remainder section */
    }
}
```

P0 phải chờ P1 trả turn = 0



```
P1
{
    while (true
    {
        while (turn == 0);
        /* critical section */
        turn = 0;
        /* remainder section */
    }
}
```

P1 tốn thời gian chạy RS, không vào CS nhưng cần P0 vào CS do turn vẫn đang bằng 1





5.4.1. Giải pháp phần mềm 1

Kiểm tra Bounded Waiting → Không đảm bảo

P₀

```
{\n    while (true\n    {\n        while (turn == 1);\n        /* critical section */\n        turn = 1;\n        /* remainder section */\n    }\n}
```

P₀ không biết pháo chờ bao lâu để được vào CS

P₁

```
{\n    while (true\n    {\n        while (turn == 0);\n        /* critical section */\n        turn = 0;\n        /* remainder section */\n    }\n}
```

P₁ tốn nhiều thời gian chạy RS



rất ngắn



rất dài



CÁC GIẢI PHÁP PHẦN MỀM

5.4.2. Giải pháp phần mềm 2

Ý tưởng của giải pháp này sử dụng một mảng flag[] để kiểm tra xem tiến trình tới lượt thực hiện của tiến trình nào với sự hỗ trợ của 2 thao tác đơn nguyên là load và store.

04.



5.4.2. Giải pháp phần mềm 2

- Giải pháp dành cho 2 tiến trình.
- Giả sử 2 lệnh hợp ngũ load và store là 2 thao tác **đơn nguyên** (không thể bị cắt ngang).
- 2 tiến trình cùng chia sẻ một biến flag
 - boolean flag[2];
- Mảng flag [] được dùng để xác định liệu tiến trình đã sẵn sàng để vào vùng tranh chấp chưa.
 $flag[i] = true;$ cho biết là P_i đã sẵn sàng để vào vùng tranh chấp.
- Giá trị của flag [i] sẽ được khởi tạo là false.



5.4.2. Giải pháp phần mềm 2

Nếu kiểm tra thấy đang lượt của P_j , thì P_i chờ và không làm gì cả

Sau khi P_i thực thi vùng tranh chấp xong thì trả lượt lại về cho P_j

Tiến trình P_i → P_i

{

while (true)

{

flag[i] = true;

while (flag[j]);



P_i sẵn sàng để vào vùng tranh chấp

/* critical section */

flag[i] = false;

/* remainder section */

}

}



5.4.2. Giải pháp phần mềm 2

- Mutual exclusion, Progress và Bounded waiting có được đảm bảo?



CÁC GIẢI PHÁP PHẦN MỀM

5.4.3. Giải pháp Peterson

Giải pháp phần mềm 1 và 2 đã đưa ra ý tưởng về cách đảm bảo mutual exclusion tuy vẫn chưa thực hiện tốt việc đảm bảo progress và bounded waiting. Khắc phục các nhược điểm của các giải pháp trên, Peterson đã đề xuất một giải pháp đảm bảo được cả 03 yêu cầu về lời giải của bài toán vùng tranh chấp.

04.



5.4.3. Giải pháp Peterson

- Giải pháp dành cho 2 tiến trình.
- Giả sử 2 lệnh hợp ngũ load và store là 2 thao tác **đơn nguyên** (không thể bị cắt ngang).
- 2 tiến trình cùng chia sẻ hai biến:

```
int turn;  
boolean flag[2];
```

- Biến `turn` có tác dụng chỉ ra tiến trình nào tới lượt để vào vùng tranh chấp.
- Mảng `flag[]` được dùng để xác định liệu tiến trình đã sẵn sàng để vào vùng tranh chấp chưa.

`flag[i] = true;` cho biết là P_i đã sẵn sàng để vào vùng tranh chấp.



5.4.3. Giải pháp Peterson

Tiến trình P_i → P_i
{

P_i sẵn sàng để vào vùng tranh chấp
Nhường lượt cho P_j
Nếu như P_j sẵn sàng và đang lượt
của P_j thì P_i chờ



```
while (true) {  
  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
  
    /* critical section */  
  
    flag[i] = false;  
  
    /* remainder section */  
}  
}
```



P_i bỏ trạng thái sẵn sàng
vào vùng tranh chấp



5.4.3. Giải pháp Peterson

- Mutual exclusion được đảm bảo:

P_i chỉ được phép vào vùng tranh chấp khi:

hoặc $\text{flag}[j] = \text{false}$ hoặc $\text{turn} = i$

và turn không thể vừa bằng i , vừa bằng j được (nếu $i = 0$ và $j = 1$ thì turn không thể vừa bằng 0, vừa bằng 1)

- Kiểm tra Progress và Bounded waiting?



5.4.3. Giải pháp Peterson

Kiểm tra Progress → Đảm bảo

```
P0
{
    while (true) {
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn
               == 1);
        /* critical section */
        flag[0] = false;
        /* remainder section */
    }
}
```

```
P1
{
    while (true) {
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn
               == 0);
        /* critical section */
        flag[1] = false;
        /* remainder section */
    }
}
```

P₁ không được vào CS:

flag [0] và turn = 0

P₀ không thực hiện CS:

- Entry section:

turn = 1

- Exit section:

flag [0] = false

➤ Nếu P₀ không vào CS,
P₀ KHÔNG ngăn cản P₁ vào
CS và ngược lại.



5.4.3. Giải pháp Peterson

Kiểm tra Bounded Waiting → Đảm bảo

```
P0
{
    while (true) {
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn
               == 1);
        /* critical section */
        flag[0] = false;
        /* remainder section */
    }
}
```

```
P1
{
    while (true) {
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn
               == 0);
        /* critical section */
        flag[1] = false;
        /* remainder section */
    }
}
```

- P₁ phải chờ tối đa là 1 lần P₀ vào vùng tranh chấp.
- CS thường rất nhỏ nên thời gian chờ đợi sẽ rất ngắn.



CÁC GIẢI PHÁP PHẦN MỀM

5.4.4. Giải pháp Peterson và kiến trúc hiện đại

Mặc dù giải pháp Peterson đã được chứng minh là hiệu quả ở trên, tuy nhiên trên các kiến trúc hiện đại thì việc này chưa được đảm bảo. Trên các hệ thống hiện đại, để cải thiện hiệu suất của hệ thống thì bộ vi xử lý và/hoặc trình biên dịch có thể sắp xếp lại các thao tác độc lập với nhau. Hãy cùng quan sát xem liệu việc này sẽ tác động như thế nào đến giải pháp Peterson trong phần này.

04.



5.4.4. Giải pháp Peterson và kiến trúc hiện đại

- Để cải thiện hiệu suất, vi xử lý và/hoặc trình biên dịch sẽ sắp xếp lại các thao tác mà độc lập với nhau.
- Việc hiểu vì sao giải pháp Peterson không hoạt động trên kiến trúc hiện đại sẽ giúp hiểu rõ hơn về race condition.
- Với các tiến trình đơn tiểu trình thì việc thực hiện các lệnh sẽ không có gì thay đổi.
- Với các tiến trình đa tiểu trình, việc sắp xếp lại các thao tác có thể dẫn đến kết quả không nhất quán hoặc không dự đoán được.



5.4.4. Giải pháp Peterson và kiến trúc hiện đại

Ví dụ về kiến trúc hiện đại

- Có 2 tiêu trình cùng chia sẻ dữ liệu:

```
boolean flag = false;  
int x = 0;
```

- Thread1 thực hiện:

```
while (!flag);  
print x;
```

- Thread2 thực hiện:

```
x = 100;  
flag = true;
```

- Kết quả kỳ vọng được in ra là:

100



5.4.4. Giải pháp Peterson và kiến trúc hiện đại

Ví dụ về kiến trúc hiện đại

- Tuy nhiên, bởi vì biến flag và biến x là độc lập với nhau nên các thao tác:

```
flag = true;
```

```
x = 100;
```

có thể bị **sắp xếp lại** thứ tự thực hiện.

- Trong trường hợp này, kết quả có thể được in ra là:

```
0
```

Khởi tạo:

```
boolean flag = false
```

```
int x = 0
```

Thread1:

```
while (!flag);  
print x;
```

Thread2 (original):

```
x = 100;  
flag = true;
```

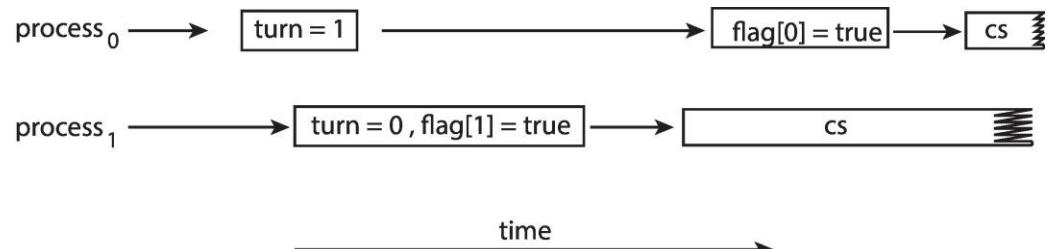
Thread2 (reordered):

```
flag = true;  
x = 100;
```



5.4.4. Giải pháp Peterson và kiến trúc hiện đại

Xét lại giải pháp Peterson



ORIGINAL

```
P0
{
    while (true) {
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn == 1);
        /* critical section */
        flag[0] = false;
        /* remainder section */
    }
}
```

```
P1
{
    while (true) {
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn == 0);
        /* critical section */
        flag[1] = false;
        /* remainder section */
    }
}
```

- Việc gán flag[] và turn bị **sắp xếp lại** thứ tự thực thi.
- P₀ và P₁ cùng vào CS.
- Để đảm bảo giải pháp Peterson hoạt động chính xác trên kiến trúc máy tính hiện đại, ta phải sử dụng **Memory Barrier**.



CÁC HỖ TRỢ TỪ PHẦN CỨNG

5.5.1. Memory Barrier

Một Memory Barrier (lớp bảo vệ bộ nhớ) là một lệnh bắt buộc bất kỳ thay đổi nào trên bộ nhớ phải được lan truyền đến tất cả các bộ xử lý.

05.



5.5.1. Memory Barrier

Memory model (mô hình bộ nhớ)

- **Memory model** trong hệ điều hành là mô hình hoạt động của bộ nhớ trong hệ thống, bao gồm cách thức quản lý và truy xuất đến các vùng nhớ được cấp phát cho các tiến trình và luồng trong hệ thống. Memory model định nghĩa các quy tắc và ràng buộc cho việc sử dụng bộ nhớ, đảm bảo tính đúng đắn, an toàn và hiệu quả của các hoạt động trên bộ nhớ.
- Hai mô hình bộ nhớ phổ biến bao gồm:
 - **Mô hình bộ nhớ được sắp xếp mạnh:** các thay đổi bộ nhớ trên một bộ xử lý sẽ được các bộ xử lý khác biết ngay lập tức.
 - **Mô hình bộ nhớ được sắp xếp yếu:** các thay đổi bộ nhớ trên một bộ xử lý CÓ THỂ sẽ KHÔNG được các bộ xử lý khác biết ngay lập tức.



5.5.1. Memory Barrier

- **Memory barrier** là một chỉ thị (instruction) mà bắt buộc mọi thay đổi trong bộ nhớ phải được truyền tải (hiển thị) đến tất cả bộ xử lý khác.
- Khi một chỉ thị memory barrier được thực hiện, hệ thống sẽ đảm bảo là tất cả thao tác `load` (nạp dữ liệu) và `store` (ghi dữ liệu) đều đã được hoàn thành trước khi các thao tác `load` và `store` sau đó được thực hiện.
- Do đó, kể cả khi các lệnh bị sắp xếp lại, memory barrier bảo đảm rằng các thao tác ghi dữ liệu đều đã được hoàn thành trong bộ nhớ và được truyền tải đến các bộ xử lý khác trước khi các thao tác nạp dữ liệu hoặc ghi dữ liệu được thực thi trong tương lai.



5.5.1. Memory Barrier

Ví dụ về Memory Barrier

- Xét lại ví dụ trước đó, chúng ta có thể dùng memory barrier để Thread1 chắc chắn in ra 100.

Thread1:

```
while (!flag)  
    memory_barrier();  
  
print x;
```

Thread2:

```
x = 100;  
memory_barrier();  
  
flag = true;
```

- Với Thread1, ta dùng `memory_barrier()` để đảm bảo rằng giá trị của `flag` được đọc trước khi đọc giá trị của `x`.
- Với Thread 2, ta dùng `memory_barrier()` để đảm bảo thao tác gán `x = 100` diễn ra trước khi gán `flag = true`.



CÁC HỖ TRỢ TỪ PHẦN CỨNG

5.5.2. Lệnh phần cứng: test_and_set

5.5.3. Lệnh phần cứng: compare_and_swap

5.5.4. Biến đơn nguyên

Sinh viên tự nghiên cứu các mục trên và trình bày tại lớp.

05.



MUTEX LOCKS

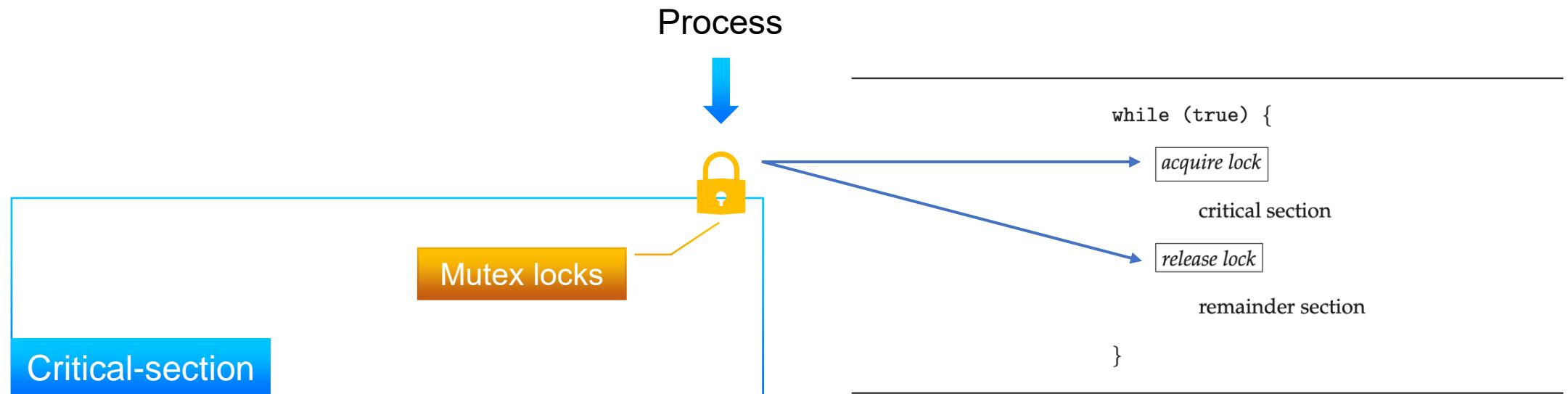
5.6.1. Định nghĩa mutex locks

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 phân 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 chờ.



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.

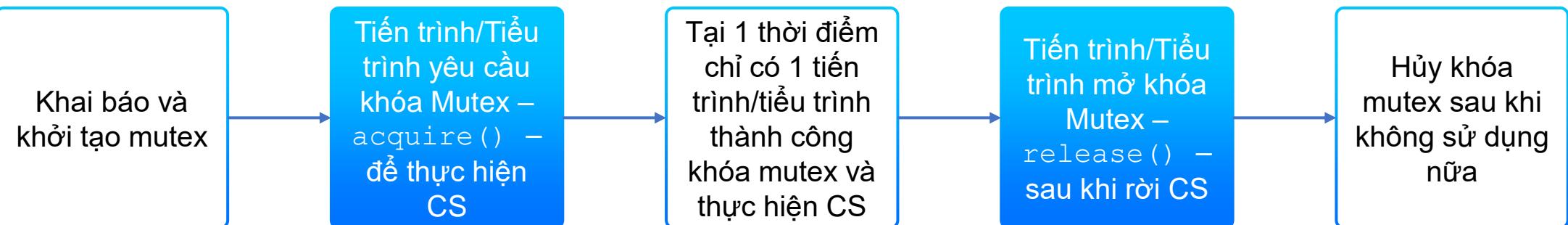


5.6.3. Cách sử dụng mutex locks

```
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).





Tóm tắt lại nội dung buổi học

- Race condition
- Vấn đề vùng tranh chấp
- Lời giải cho vấn đề vùng tranh chấp
- Các giải pháp phần mềm
- Giải pháp phần cứng
- Mutex locks

THẢO LUẬN

