



COMPUTER ENGINEERING



**UIT**  
TRƯỜNG ĐẠI HỌC  
CÔNG NGHỆ THÔNG TIN

# HỆ ĐIỀU HÀNH

## Chương 5 – Đồng bộ (2)

4/8/2024



# Ôn tập chương 5 (1)

- Khi nào thì xảy ra tranh chấp race condition?
- Vấn đề Critical Section là gì?
- Yêu cầu của lời giải cho CS problem?
- Có mấy loại giải pháp? Kể tên?



## Mục tiêu chương 5 (2)

- Hiểu được nhóm giải pháp Busy waiting bao gồm:
  - Các giải pháp phần mềm
  - Các giải pháp phần cứng



## Nội dung chương 5 (2)

- Các giải pháp phần mềm
  - Sử dụng giải thuật kiểm tra luân phiên
  - Sử dụng các biến cờ hiệu
  - Giải pháp của Peterson
  - Giải pháp Bakery
- Các giải pháp phần cứng
  - Cấp ngắt
  - Chỉ thị TSL



# Giải thuật 1

## ■ Biến chia sẻ

- `int turn;`                    `/* khởi đầu turn = 0 */`
- nếu `turn = i` thì  $P_i$  được phép vào critical section, với  $i = 0$  hay  $1$

## ■ Process $P_i$

```
do {  
    while (turn != i);  
    critical section  
    turn = j;  
    remainder section  
} while (1);
```

## ■ Thỏa mãn Mutual exclusion (1)

## ■ Nhưng không thỏa mãn yêu cầu về progress (2) và bounded waiting (3) vì tính chất strict alternation của giải thuật



# Giải thuật 1 (tt)

```
Process P0:  
do  
    while (turn != 0);  
        critical section  
    turn := 1;  
        remainder section  
while (1);
```

```
Process P1:  
do  
    while (turn != 1);  
        critical section  
    turn := 0;  
        remainder section  
while (1);
```

- Điều gì xảy ra nếu P0 có RS (remainder section) rất lớn còn P1 có RS nhỏ?



# Giải thuật 2

## ■ Biến chia sẻ

- `boolean flag[ 2 ]; /* khởi đầu flag[ 0 ] = flag[ 1 ] = false */`
- Nếu `flag[ i ] = true` thì  $P_i$  “sẵn sàng” vào critical section.

## ■ Process $P_i$

**do {**

**flag[ i ] = true; /\*  $P_i$  “sẵn sàng” vào CS \*/**

**while ( flag[ j ] ); /\*  $P_i$  “nhường”  $P_j$  \*/**

**critical section**

**flag[ i ] = false;**

**remainder section**

**} while (1);**

- Thỏa mãn Mutual exclusion (1)
- Không thỏa mãn progress. Vì sao?



# Giải thuật 3 (Peterson)

## ■ Biến chia sẻ

- Kết hợp cả giải thuật 1 và 2

## ■ Process $P_i$ , với $i = 0$ hoặc $i = 1$

do {

```
    flag[ i ] = true;      /* Process i sẵn sàng */  
    turn = j;              /* Nhường process j */  
    while (flag[ j ] and turn == j);
```

critical section

```
    flag[ i ] = false;
```

remainder section

```
} while (1);
```

## ■ Thoả mãn được cả 3 yêu cầu ?

⇒ giải quyết bài toán critical section cho 2 process





# Giải thuật 3 (Peterson) cho 2 tiến trình

Process  $P_0$

```
do {  
    /* 0 wants in */  
    flag[0] = true;  
    /* 0 gives a chance to 1 */  
    turn = 1;  
    while (flag[1] && turn == 1);  
        critical section  
    /* 0 no longer wants in */  
    flag[0] = false;  
        remainder section  
} while(1);
```

Process  $P_1$

```
do {  
    /* 1 wants in */  
    flag[1] = true;  
    /* 1 gives a chance to 0 */  
    turn = 0;  
    while (flag[0] && turn == 0);  
        critical section  
    /* 1 no longer wants in */  
    flag[1] = false;  
        remainder section  
} while(1);
```



## Giải thuật 3: Tính đúng đắn

- Giải thuật 3 thỏa mutual exclusion, progress, và bounded waiting
- Mutual exclusion được đảm bảo bởi vì
  - P0 và P1 đều ở trong CS nếu và chỉ nếu  $\text{flag}[0] = \text{flag}[1] = \text{true}$  và  $\text{turn} = i$  cho mỗi  $P_i$  (không thể xảy ra)
- Chứng minh thỏa yêu cầu về progress và bounded waiting
  - $P_i$  không thể vào CS nếu và chỉ nếu bị kẹt tại vòng lặp `while()` với điều kiện  $\text{flag}[j] = \text{true}$  và  $\text{turn} = j$
  - Nếu  $P_j$  không muốn vào CS thì  $\text{flag}[j] = \text{false}$  và do đó  $P_i$  có thể vào CS



## Giải thuật 3: Tính đúng đắn (tt)

- Nếu  $P_j$  đã bật  $\text{flag}[j] = \text{true}$  và đang chờ tại  $\text{while}()$  thì có chỉ hai trường hợp là  $\text{turn} = i$  hoặc  $\text{turn} = j$
- Nếu  $\text{turn} = i$  và  $P_i$  vào CS. Nếu  $\text{turn} = j$  thì  $P_j$  vào CS nhưng sẽ bật  $\text{flag}[j] = \text{false}$  khi thoát ra  $\rightarrow$  cho phép  $P_i$  vào CS
- Nhưng nếu  $P_j$  có đủ thời gian bật  $\text{flag}[j] = \text{true}$  thì  $P_j$  cũng phải gán  $\text{turn} = i$
- Vì  $P_i$  không thay đổi trị của biến  $\text{turn}$  khi đang kẹt trong vòng lặp  $\text{while}()$ ,  $P_i$  sẽ chờ để vào CS nhiều nhất là sau một lần  $P_j$  vào CS (bounded waiting)



# Giải thuật bakery: n process

- Trước khi vào CS, process  $P_i$  nhận một con số. Process nào giữ con số nhỏ nhất thì được vào CS
- Trường hợp  $P_i$  và  $P_j$  cùng nhận được một chỉ số:
  - Nếu  $i < j$  thì  $P_i$  được vào trước. (Đối xứng)
- Khi ra khỏi CS,  $P_i$  đặt lại số của mình bằng 0
- Cơ chế cấp số cho các process thường tạo các số theo cơ chế tăng dần, ví dụ 1, 2, 3, 3, 3, 3, 4, 5,...
- Kí hiệu
  - $(a,b) < (c,d)$  nếu  $a < c$  hoặc nếu  $a = c$  và  $b < d$
  - $\max(a_0, \dots, a_k)$  là con số  $b$  sao cho  $b \geq a_i$  với mọi  $i = 0, \dots, k$



# Giải thuật bakery: n process (tt)

```
/* shared variable */
boolean    choosing[ n ]; /* initially, choosing[ i ] = false */
int        num[ n ];      /* initially, num[ i ] = 0 */

do {
    choosing[ i ] = true;
    num[ i ]      = max(num[0], num[1],..., num[n - 1]) + 1;
    choosing[ i ] = false;
    for (j = 0; j < n; j++) {
        while (choosing[ j ]);
        while ((num[ j ] != 0) && (num[ j ], j) < (num[ i ], i));
    }
    critical section
    num[ i ] = 0;
    remainder section
} while (1);
```



# Từ software đến hardware

- Khuyết điểm của các giải pháp software:
  - Các process khi yêu cầu được vào vùng tranh chấp đều phải liên tục kiểm tra điều kiện (busy waiting), tốn nhiều thời gian xử lý của CPU
  - Nếu thời gian xử lý trong vùng tranh chấp lớn, một giải pháp hiệu quả nên có cơ chế block các process cần đợi.
- Các giải pháp phần cứng:
  - Cấm ngắt (disable interrupts)
  - Dùng các lệnh đặc biệt



# Cấm ngắt

- Trong hệ thống uniprocessor:  
mutual exclusion được đảm bảo
  - Nhưng nếu system clock được cập nhật do interrupt thì...
- Trong hệ thống multiprocessor:  
mutual exclusion không được đảm bảo
  - Chỉ cấm ngắt tại CPU thực thi lệnh `disable_interrupts`
  - Các CPU khác vẫn có thể truy cập bộ nhớ chia sẻ

Process  $P_i$ :

```
do {  
    disable_interrupts();  
    critical section  
    enable_interrupts();  
    remainder section  
} while (1);
```



# Lệnh TestAndSet

- Đọc và ghi một biến trong một thao tác atomic (không chia cắt được)

```
boolean TestAndSet( boolean
*target){
    boolean rv = *target;
    *target = true;
    return rv;
}
```

- Shared data:  
    boolean lock = false;

- Process  $P_i$ :

```
do {
    while (TestAndSet(&lock));
        critical section
    lock = false;
        remainder section
} while (1);
```





# Lệnh TestAndSet

- Mutual exclusion được bảo đảm: nếu  $P_i$  vào CS, các process  $P_j$  khác đều đang busy waiting
- Khi  $P_i$  ra khỏi CS, quá trình chọn lựa process  $P_j$  vào CS kế tiếp là tùy ý  $\Rightarrow$  không bảo đảm điều kiện bounded waiting. Do đó có thể xảy ra starvation (bị bỏ đói)
- Các processor (ví dụ Pentium) thông thường cung cấp một lệnh đơn là  $\text{Swap}(a, b)$  có tác dụng hoán chuyển nội dung của  $a$  và  $b$ .
  - $\text{Swap}(a, b)$  cũng có ưu nhược điểm như TestAndSet



# Swap và mutual exclusion

- Biến chia sẻ **lock** được khởi tạo giá trị false
- Mỗi process  $P_i$  có biến cục bộ **key**
- Process  $P_i$  nào thấy giá trị **lock** = **false** thì được vào CS.
  - ▶ Process  $P_i$  sẽ loại trừ các process  $P_j$  khác khi thiết lập **lock = true**

```
void Swap(boolean *a,  
           boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

- Biến chia sẻ (khởi tạo là **false**)  
**bool lock;**  
**bool key;**

- Process  $P_i$

```
do {  
    key = true;  
    while (key == true)  
        Swap(&lock, &key);  
        critical section  
    lock = false;  
        remainder section  
} while (1)
```

Không thỏa mãn bounded waiting



# Giải thuật dùng TestAndSet thoả mãn 3 yêu cầu

- Cấu trúc dữ liệu dùng chung (khởi tạo là false)

bool waiting[ n ];

bool lock;

- Mutual exclusion:  $P_i$  chỉ có thể vào CS nếu và chỉ nếu hoặc waiting[ i ] = false, hoặc key = false

- key = false chỉ khi TestAndSet (hay Swap) được thực thi

- Process đầu tiên thực thi TestAndSet mới có key == false; các process khác đều phải đợi

- waiting[ i ] = false chỉ khi process khác rời khỏi CS

- Chỉ có một waiting[ i ] có giá trị false

- Progress: chứng minh tương tự như mutual exclusion

- Bounded waiting: waiting in the cyclic order



# Giải thuật dùng TestAndSet thoả mãn 3 yêu cầu (tt)

do {

```
waiting[ i ] = true;  
key = true;  
while (waiting[ i ] && key)  
    key = TestAndSet(lock);  
waiting[ i ] = false;
```

*critical section*

```
j = (i + 1) % n;  
while ( (j != i) && !waiting[ j ] )  
    j = (j + 1) % n;  
if (j == i)  
    lock = false;  
else  
    waiting[ j ] = false;
```

*remainder section*

} while (1)



# Tóm tắt lại nội dung buổi học

- Các giải pháp phần mềm
  - Sử dụng giải thuật kiểm tra luân phiên
  - Sử dụng các biến cờ hiệu
  - Giải pháp của Peterson
  - Giải pháp Bakery
- Các giải pháp phần cứng
  - Cấp ngắt
  - Chỉ thị TSL



# THẢO LUẬN

