

Merry  
Christmas

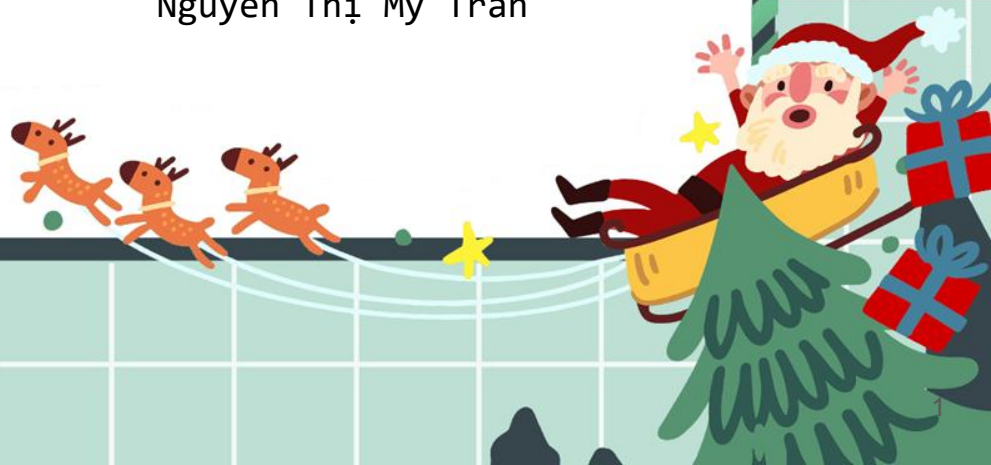


Khoa Khoa học  
và Kỹ thuật Thông tin

# Training Hệ Điều Hành – BHT HTTT

**Trainer :** Nguyễn Quốc Khánh

Nguyễn Thị Mỹ Trân



*Nội dung thi ( trắc nghiệm )*

Chương 5 : Đồng bộ (5.2 & 5.3)

Chương 6 : Deadlocks

Chương 7 : Quản lý bộ nhớ

Chương 8 : Bộ nhớ ảo

Chương 9 : HĐH Linux & Windows

## Chương 5 : Đồng bộ

Tranh chấp race condition

Vấn đề Critical Section

Giải pháp cho CS

- Nhóm giải pháp Busy waiting
- Nhóm giải pháp Sleep & Wakeup

Semaphore

Monitor

Merry Christmas

## Chương 5 : Đồng bộ Tranh chấp race condition

Tình trạng nhiều process truy xuất và thao tác đồng thời lên dữ liệu chia sẻ. Kết quả cuối cùng của việc truy xuất đồng thời này phụ thuộc thứ tự thực thi của các lệnh thao tác dữ liệu.

- Khảo sát các process/thread thực thi đồng thời và chia sẻ dữ liệu (qua shared memory, file).
- Nếu không có sự kiểm soát khi truy cập các dữ liệu chia sẻ thì có thể đưa đến ra trường hợp không nhất quán dữ liệu (data inconsistency).
- Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế bảo đảm sự thực thi có trật tự của các process đồng thời.



## Chương 5 : Đồng bộ

- Tranh chấp race condition
- Vấn đề Critical Section
- Giải pháp cho CS
  - Nhóm giải pháp Busy wating
  - Nhóm giải pháp Sleep & Wakeup

Semaphore

Monitor

Merry Christmas

## Chương 5 : Đồng bộ Vấn đề Critical Section

Trong mỗi process có những đoạn code có chứa các thao tác lên dữ liệu chia sẻ. Đoạn code này được gọi là vùng tranh chấp (critical section, CS).

➡ *Một CS trong một thời điểm nhất định, chỉ có một process được phép chạy*

**Vấn đề CS:** Tìm một cách thiết kế một giao thức (một cách thức) nào đó để các process có thể phối hợp với nhau hoàn thành nhiệm vụ của nó.

## Chương 5 : Đồng bộ

Tranh chấp race condition

Vấn đề Critical Section

Giải pháp cho CS

- Nhóm giải pháp Busy waiting
- Nhóm giải pháp Sleep & Wakeup

Semaphore

Monitor

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS

Lời giải phải thỏa ba tính chất:

- (1) Loại trừ tương hỗ (Mutual exclusion): Khi một process P đang thực thi trong vùng tranh chấp (CS) của nó thì không có process Q nào khác đang thực thi trong CS của Q.
- (2) Progress: Một tiến trình tạm dừng bên ngoài vùng tranh chấp không được ngăn cản các tiến trình khác vào vùng tranh chấp.
- (3) Chờ đợi giới hạn (Bounded waiting): Mỗi process 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).

Merry Christmas



## Chương 5 : Đồng bộ

Tranh chấp race condition

Vấn đề Critical Section

Giải pháp cho CS

- Nhóm giải pháp Busy waiting
- Nhóm giải pháp Sleep & Wakeup

Semaphore

Monitor

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS

Nhóm giải pháp *Busy Waiting*

- Tính chất:
  - Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng (thông qua việc kiểm tra điều kiện vào CS liên tục).
  - Không đòi hỏi sự trợ giúp của hệ điều hành.

```
While (chưa có quyền) do nothing() ;
```

```
CS;
```

```
Từ bỏ quyền sử dụng CS
```

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS

*Nhóm giải pháp Busy Waiting*

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

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS - Sử dụng kiểm tra Luân phiên

**Tiếp cận :** Hai tiến trình sử dụng chung biến *turn* (phản ánh phiên tiến trình nào được vào miền găng), được khởi động với giá trị 0. Nếu *turn* = 0, tiến trình A được vào miền găng. Nếu *turn* = 1, tiến trình A đi vào một vòng lặp chờ đến khi *turn* nhận giá trị 0. Khi tiến trình A rời khỏi miền găng, nó đặt giá trị *turn* về 1 để cho phép tiến trình B đi vào miền găng.

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);
```

Merry Christmas

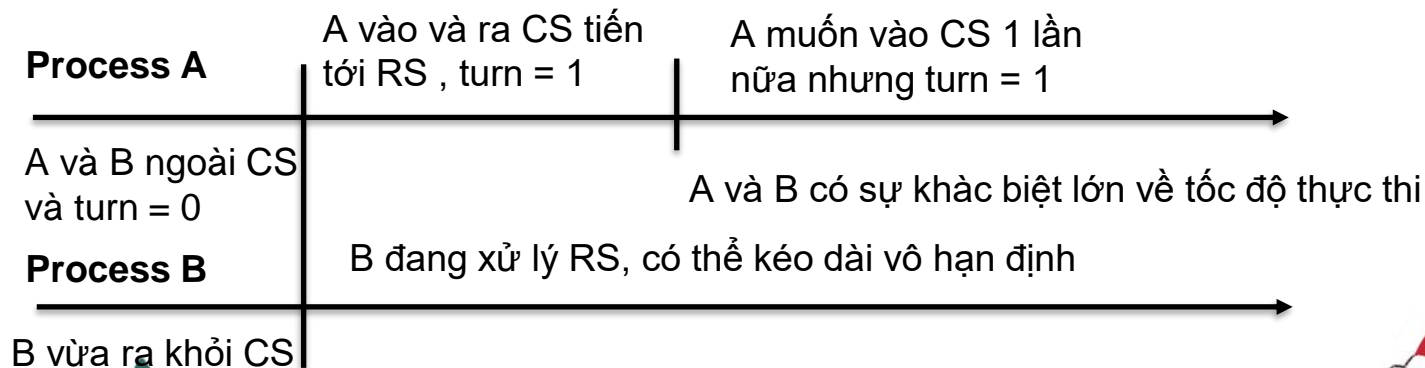


## Chương 5 : Đồng bộ Giải pháp cho CS - Sử dụng kiểm tra Luân phiên

### Đặc điểm:

Thỏa mãn yêu cầu **Mutual exclusion**

Không thỏa mãn yêu cầu về **progress** và **bounded wating** vì tính chất **strick alternation**. (sự khác biệt của tộc độ thực thi)



## Chương 5 : Đồng bộ Giải pháp cho CS – Sử dụng các biến cờ hiệu

### 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);`

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS – Sử dụng các biến cờ hiệu

### Đặc điểm:

- Thỏa mãn yêu cầu **Mutual exclusion**
- Không thỏa mãn yêu cầu về **progress** khi 2 tiến trình có tộc độ thực thi ngang nhau và biến `flag[]` có cùng giá trị.

P0

```
do
{
    flag[0] = true;
    while (flag[1]);
    CS;
    flag[0] = false;
    RS;
} while(1);
```

P1

```
do
{
    flag[1] = true;
    while (flag[0]);
    CS;
    flag[1] = false;
    RS;
} while(1);
```

## Chương 5 : Đồng bộ Giải pháp cho CS – Peterson

- Kết hợp kiểm tra luân phiên và biến cờ hiệu
- Giải quyết được 3 yêu cầu về CS
- Biến chia sẻ : turn và flag[]

```
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);
```



## Chương 5 : Đồng bộ Giải pháp cho CS – Peterson

- Kết hợp kiểm tra luân phiên và biến cờ hiệu
- Giải quyết được 3 yêu cầu về CS
- Biến chia sẻ : turn và flag[]

### Nhược điểm :

Busy waiting -> tiêu tốn CPU  
Chỉ giới hạn ở hai process

```
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);
```

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS - Peterson

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);
```

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS – Barkey

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. (Đôi 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 if  $a = c$  và  $b < d$

□  $\max(a_0, \dots, a_k)$  là con số  $b$  sao cho  $b \geq a_i$  với mọi  $i = 0, \dots, k$

```
/* 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);
```

Merry Christmas

**Chương 5 : Đồng bộ Giải pháp cho CS – Tổng kết**

	Cờ hiệu	Luân phiên	Peterson	Barkey
<b>Biến chia sẻ</b>	Turn	Flag[2]	Turn và Flag[2]	Choosing[N]. Num[n]]
<b>Process</b>	2	2	2	N
<b>Mutual Exclusion</b>	Có	Có	Có	Có
<b>Progress</b>	Không	Không	Có	Có
<b>Bounded wating</b>	Không	Không	Có	Có



## Chương 5 : Đồng bộ Giải pháp cho CS – Tổng kết

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

## *Chương 5 : Đồng bộ Giải pháp cho CS – Cấm ngắt*

- Cấm ngắt ( disable interrupts) : Cho phép tiến trình cấm ngắt tắt cả các tiến trình khác và phục hồi khi ra khỏi CS.
- Hệ thống không thể tạm dừng hoạt động của tiến trình đang xử lý để cấp phát CPU cho tiến trình khác.

## Chương 5 : Đồng bộ Giải pháp cho CS – Cấm ngắt

### Đặc điểm :

- Chỉ sử dụng cho hệ thống đơn tiến trình với điều kiện ngắt đồng hồ không xảy ra.
- Trên hệ thống đa nhiệm không thỏa mutual exclusion vì lệnh cấm ngắt chỉ có tác dụng trên bộ xử lý đang xử lý tiến trình
- Không được ưu chuộng vì thiếu thận trọng khi cho phép tiến trình người dùng được thực hiện lệnh cấm ngắt

Process  $P_i$ :

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

## Chương 5 : Đồng bộ Giải pháp cho CS – Chỉ thị

Nhiều máy tính cung cấp các chỉ thị đặc biệt (atomic) để giải quyết vấn đề đồng bộ hóa.

**Ví dụ :** TestAndSet(), Swap(), Lock() and Unlock(),...

**Đặc điểm :** là các dòng lệnh không thể bị ngắt giúp giảm nhẹ việc lập trình để giải quyết vấn đề nhưng không dễ dàng cài đặt sao cho xử lý không phân chia ( nhất là nhiều bộ xử lý )



## Chương 5 : Đồng bộ Giải pháp cho CS - TestAndSet()

### TestAndSet :

- Kiểm tra và cập nhật ( đọc và ghi ) một vùng nhớ trong một thao tác atomic ( không thể phân chia)
- Tương đồng với Kiểm tra luân phiên nhưng tốt hơn vì có sự trợ giúp của Hệ điều hành
- Áp dụng được cho Multiprocess ( tốt hơn cấm ngắt )

## Chương 5 : Đồng bộ Giải pháp cho CS - TestAndSet()

```
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);
```

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS - TestAndSet()

### TestAndSet :

Kiểm tra và cập nhật ( đọc và ghi ) một vùng nhớ trong một thao tác atomic ( không thể phân chia)

Tương đồng với Kiểm tra luân phiên nhưng tốt hơn vì có sự trợ giúp của Hệ điều hành

Áp dụng được cho Multiprocess ( tốt hơn cấm ngắt )

## Chương 5 : Đồng bộ Giải pháp cho CS - TestAndSet()

### TestAndSet :

#### Ưu điểm :

- Loại trừ tương hỗ hiệu quả
- Loại bỏ deadlock
- Đảm bảo yêu cầu progress

#### Nhược điểm :

- Không đảm bảo bounded waiting ( không có cơ chế lựa chọn process tiếp theo vào CS)
- Có thể gây ra tình trạng chết đói



## Chương 5 : Đồng bộ Giải pháp cho CS - Swap

### Swap :

Tránh sử dụng biến dùng chung (lock) như TestAndSet

Khởi tạo Lock : biến toàn cục

Mỗi process có biến key cục bộ -> Hoán đổi key và lock để xin quyền vào CS

Khi lock == False (không có process nào khác đang cấm ngắt) -> swap() phá vỡ vòng lặp while() -> process đi vào CS

Chưa giải quyết được Bounded Wating

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

## Chương 5 : Đồng bộ Giải pháp cho CS - Swap

### Swap :

Tránh sử dụng biến dùng chung (lock) như TestAndSet

Lock : biến toàn cục

Mỗi process có biến key cục bộ -> Hoán đổi key và lock để xin quyền vào CS

Khi lock == False (không có process nào khác đang cấm ngắt) -> swap() phá vỡ vòng lặp while() -> process đi vào CS

Chưa giải quyết được Bounded Wating

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)
```

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS – Unlock and Lock

### Unlock and Lock :

- Kết hợp TestAndSet và Swap
- Thêm 1 mảng waiting[n] lưu tất cả process đang đợi vào CS -> dễ dàng lựa chọn process kế tiếp vào CS -> cyclic order
- Thỏa mãn 3 yêu cầu của vấn đề CS.

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

```
bool waiting[ n ];  
bool lock;
```

## Chương 5 : Đồng bộ Giải pháp cho CS - Unlock and Lock

```
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)
```



*Chương 5 : Đồng bộ Giải pháp cho CS - Unlock and Lock*

	TestAndSet	Swap	Lock and Unlock
<b>Biến chia sẻ</b>	Lock	Lock	Lock
<b>Biến cục bộ</b>		Key	Key Waiting[n]
<b>Mutual exclusion</b>	Có	Có	Có
<b>Progress</b>	Có	Có	Có
<b>Bounded Waiting</b>	Không	Không	Có

## Chương 5 : Đồng bộ Giải pháp cho CS

### Nhận xét :

Busy wating :

- Cần vòng lặp để kiểm tra tiến trình được phép vào CS, nếu chưa tiến trình phải chờ.
- Liên tục kiểm tra điều kiện để phát hiện tiến trình thích hợp.

-> Các tiến trình tiêu thụ và chiếm rất nhiều CPU

-> Tránh giải pháp busy wating trong đồng bộ hóa tiến trình.

-> Tiến trình chưa đủ điều kiện -> blocked (từ bỏ quyền sử dụng CPU)

## Chương 5 : Đồng bộ Giải pháp cho CS

Nhóm giải pháp Sleep & Wakeup

- Tính chất:
  - Từ bỏ CPU khi chưa được vào CS.
  - Cần sự hỗ trợ từ hệ điều hành (để đánh thức process và đưa process vào trạng thái blocked).

```
if (chưa có quyền) Sleep() ;
```

```
CS;
```

```
Wakeup (somebody);
```

Merry Christmas

*Chương 5 : Đồng bộ Giải pháp cho CS*  
*Nhóm giải pháp Sleep & Wakeup*

- Bao gồm một vài loại :
  - Semaphore
  - Monitor
  - Message



## Chương 5 : Đồng bộ Giải pháp cho CS Semaphore

**SLEEP**: tạm dừng hoạt động của tiến trình (blocked) gọi nó và chờ đến khi một tiến trình khác đánh thức

**WAKEUP** : nhận tham số duy nhất – tiến trình được tái kích hoạt (về trạng thái ready)

Ý tưởng :

- Tiến trình chưa đủ điều kiện vào CS -> gọi **SLEEP** để tự khóa đến khi 1 tiến trình khác gọi **WAKEUP**
- Một tiến trình gọi **WAKEUP** khi ra khỏi miền găng để đánh thức tiến trình đang chờ

```
int busy;    // =1 nếu CS đang bị chiếm
int blocked; // số P đang bị khóa
do{
    if (busy){
        blocked = blocked + 1;
        sleep();
    }
    else busy =1;
    CS;
    busy = 0;
    if (blocked !=0){
        wakeup (process);
        blocked = blocked -1;
    }
    RS;
}while (1);
```

Merry Christmas

*Chương 5 : Đồng bộ Giải pháp cho CS*  
*Nhóm giải pháp Sleep & Wakeup*

- Bao gồm một vài loại :
  - Semaphore
  - Monitor
  - Message

## Chương 5 : Đồng bộ Giải pháp cho CS Semaphore

- Công cụ đồng bộ cung cấp bởi OS không đòi hỏi Busy wating
- Là một biến số nguyên có cấu trúc :

```
typedef struct {  
    int value;  
    struct process *L; /* process queue */  
} semaphore;
```


- Khi phải đợi thì process được đặt vào blocked queue – chứa các proces đang chờ đợi được thực thi -> **tránh busy wating**

## Chương 5 : Đồng bộ Giải pháp cho CS Semaphore

- Một semaphore S có 3 thao tác :
- Khởi tạo semaphore : giá trị khởi tạo là số lượng process được thực hiện trong CS tại 1 thời điểm.
- Wait(S) hay P(S) : giảm giá trị semaphore đi một đơn vị ( $S = S - 1$ ). Nếu S.value âm, process thực hiện lệnh wait() bị blocked cho đến khi được đánh thức.

```
void wait(semaphore S) {  
    S.value--;  
    if (S.value < 0) {  
        add this process to S.L;  
        block();  
    }  
}
```

Ready sang wating





## Chương 5 : Đồng bộ Giải pháp cho CS Semaphore

- Signal(S) hay V(S) : tăng giá trị S.value lên 1 ( $S = S+1$ ). Nếu giá trị S không dương (vẫn còn process đang bị blocked) thì lấy một process Q nào đó rồi gọi wakeup(Q) để đánh thức Q.

```
void signal(semaphore S) {  
    S.value++;  
    if (S.value <= 0) {  
        remove a process P from S.L;  
        wakeup(P);  
    }  
}
```

Waiting sang ready

remove a process P from S.L;  
wakeup(P);

## Chương 5 : Đồng bộ Giải pháp cho CS Semaphore

- Khi  $S.value \geq 0$ : số process có thể thực thi  $wait(S)$  mà không bị blocked =  $S.value$
- Khi  $S.value < 0$ : số process đang đợi trên S là  $|S.value|$
- Atomic và mutual exclusion: không được xảy ra trường hợp 2 process cùng đang ở trong thân lệnh  $wait(S)$  và  $signal(S)$  (cùng semaphore S) tại một thời điểm (ngay cả với hệ thống multiprocessor)

⇒ do đó, đoạn mã định nghĩa các lệnh  $wait(S)$  và  $signal(S)$  cũng chính là vùng tranh chấp

## *Chương 5 : Đồng bộ Giải pháp cho CS Semaphore*

### *Các loại Semaphore*

- Counting semaphore: một số nguyên có giá trị không hạn chế.
- Binary semaphore: có trị là 0 hay 1. Binary semaphore rất dễ hiện thực.
- Có thể hiện thực counting semaphore bằng binary semaphore.

## Chương 5 : Đồng bộ Giải pháp cho CS

### Critical Region (CR)

- Là một cấu trúc ngôn ngữ cấp cao (high-level language construct, được dịch sang mã máy bởi một compiler), thuận tiện hơn cho người lập trình.
- Một biến chia sẻ  $v$  kiểu dữ liệu  $T$ , khai báo như sau

**$v$ : shared  $T$ ;**

- Biến chia sẻ  $v$  chỉ có thể được truy xuất qua phát biểu sau

**region  $v$  when  $B$  do  $S$ ; /\*  $B$  là một biểu thức Boolean \*/**

- Ý nghĩa: trong khi  $S$  được thực thi, không có quá trình khác có thể truy xuất biến  $v$ .



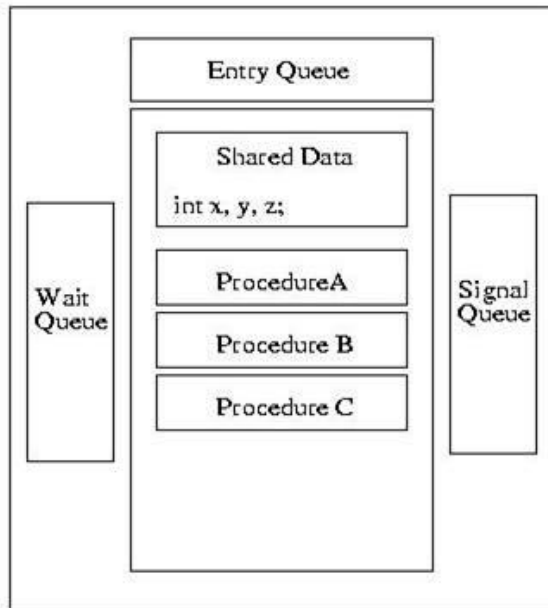
## Chương 5 : Đồng bộ Giải pháp cho CS Monitor

### Đặc điểm

- Là một công cụ đồng bộ tương tự CR và có chức năng tương tự như semaphore (có thể hiện thực bằng semaphore) nhưng dễ điều khiển hơn.
- Là một kiểu dữ liệu trừu tượng, lưu lại các biến chia sẻ và các phương thức dùng để thao tác lên các biến chia sẻ đó. Các biến chia sẻ trong monitor chỉ có thể được truy cập bởi các phương thức được định nghĩa trong monitor.
- Một monitor chỉ có 1 process hoạt động, tại một thời điểm bất kỳ đang xét. -> đảm bảo mutual exclusion.

## Chương 5 : Đồng bộ Giải pháp cho CS Monitor

Cấu trúc của một monitor :



```
monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        . . .
    }

    function P2 ( . . . ) {
        . . .
    }

    .
    .
    .
    function Pn ( . . . ) {
        . . .
    }

    initialization_code ( . . . ) {
        . . .
    }
}
```

## *Chương 5 : Đồng bộ Giải pháp cho CS Monitor*

### **Condition variable :**

- Nếu có nhiều hơn 1 process trong monitor ta cần phải khai báo biến điều kiện :  
condition a, b;
- Các biến điều kiện đều cục bộ và chỉ được truy cập bên trong monitor.

## Chương 5 : Đồng bộ Giải pháp cho CS Monitor

### Thao tác trên biến điều kiện

- a.wait: process gọi tác vụ này sẽ bị “block trên biến điều kiện” a  
Process này chỉ có thể tiếp tục thực thi khi có process khác thực hiện tác vụ a.signal
- a.signal: phục hồi quá trình thực thi của process bị block trên biến điều kiện a.
  - Nếu có nhiều process: chỉ chọn một
  - Nếu không có process: không có tác dụng



Merry Christmas



## Chương 5 : Đồng bộ Giải pháp cho CS

### Phân biệt semaphore và monitor

Các yếu tố so sánh	Semaphore	Monitor
Cấu trúc cơ bản	Semaphores là một số nguyên S. Hay nói cách khác, semaphore chỉ là một biến đếm, nhưng được bảo đảm cho việc truy xuất đa luồng (thread-safe).	Monitor là một kiểu dữ liệu trừu tượng (abstract data type).
Mức độ trừu tượng	Cấp độ trừu tượng thấp hơn Monitor.	Cấp độ trừu tượng cao hơn Semaphore. Monitor có thể được cài đặt bằng Semaphore.
Cách thức hoạt động	Giá trị của Semaphore S biểu thị số lượng tài nguyên có sẵn trong hệ thống (available resources).	Monitor chứa các biến dùng chung và các phương thức thao tác lên các biến dùng chung đó.

Merry Christmas

## Chương 5 : Đồng bộ Giải pháp cho CS

### Phân biệt semaphore và monitor

Truy cập	Khi bất kỳ tiến trình nào cần truy cập vào tài nguyên dùng chung (shared resources), tiến trình đó sẽ phải gọi wait() với semaphore S. Sau khi xong việc truy xuất tài nguyên dùng chung, nó sẽ trả lại tài nguyên rồi gọi signal() với semaphore S.	Khi bất kỳ tiến trình nào cần truy cập các biến dùng chung, tiến trình đó cần phải truy cập biến đó thông qua các phương thức trong Monitor.
Các biến điều kiện (Condition Variable)	Semaphore không có biến điều kiện.	Monitor có các biến điều kiện.
An toàn	Semaphore cấp thấp hơn và yêu cầu người sử dụng cẩn thận trong việc quản lý tài nguyên.	Monitor cấp cao hơn, tự động quản lý việc signal() và wait(), khiến cho việc sử dụng đơn giản và an toàn hơn

Merry Christmas

## Chương 5 : Đồng bộ Bài tập

Lệnh TestAndSet được xếp vào nhóm nào trong các nhóm giải pháp đồng bộ dưới đây?

- A. Busy waiting sử dụng phần mềm
- B. Busy waiting sử dụng phần cứng**
- C. Sleep & Wake up sử dụng phần mềm
- D. Sleep & Wake up sử dụng phần cứng

Merry Christmas

## Chương 5 : Đồng bộ Bài tập

Chọn phát biểu **ĐÚNG** trong các phát biểu dưới đây? (Cuối kỳ HK1 2019 – 2020)

**A. Đoạn mã định nghĩa các lệnh wait(S) và signal(S) cũng là các vùng tranh chấp.**

B. Lệnh wait(S) sẽ làm tăng giá trị của semaphore S thêm 1 đơn vị.

C. Lệnh signal(S) sẽ làm giảm giá trị của semaphore S đi 1 đơn vị.

D. Có thể hiện thực binary semaphore bằng counting semaphore.

E. Counting semaphore là semaphore có giá trị tối đa là 1.



## Chương 5 : Đồng bộ Bài tập

Giải pháp đồng bộ của Peterson là sự kết hợp của việc sử dụng các biến cờ hiệu với giải pháp nào?

- A. Cấm ngắt
- B. Giải thuật kiểm tra luân phiên**
- C. Lệnh swap
- D. Monitor

## Chương 5 : Đồng bộ Bài tập

Trong giải pháp đồng bộ sử dụng semaphore, để cho phép tối đa 5 tiến trình vào miền găng, cần khởi tạo semaphore với giá trị bằng bao nhiêu?

- A. 4
- B. 5
- C. 6
- D. 10

Khởi tạo semaphore : giá trị khởi tạo là số lượng process được thực hiện trong CS tại 1 thời điểm.

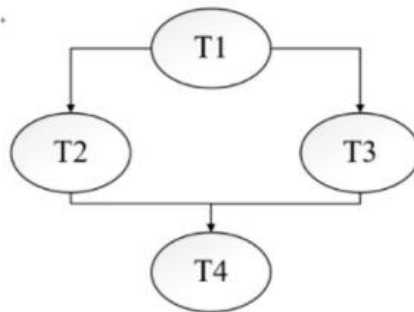
## Chương 5 : Đồng bộ Bài tập

Chọn phát biểu SAI trong các phát biểu dưới đây? (Cuối kỳ HK1 2019 – 2020)

- A. Critical region là một cấu trúc ngôn ngữ cấp cao.
- B. Nếu sử dụng semaphore không đúng thì có thể xảy ra tình trạng deadlock hoặc starvation.  
starvation.
- C. Monitor có thể được hiện thực bằng semaphore.
- D. Nhóm giải pháp đồng bộ “Sleep & Wakeup” không cần sự hỗ trợ của hệ điều hành.

## Chương 5 : Đồng bộ Bài tập

9. Xét một hệ thống có 4 tiểu trình T1, T2, T3, T4. Quan hệ giữa các tiểu trình này được biểu diễn như sơ đồ bên dưới, với mũi tên từ tiểu trình (Tx) sang tiểu trình (Ty) có nghĩa là tiểu trình Tx phải kết thúc quá trình hoạt động của nó trước khi tiểu trình Ty bắt đầu thực thi. Giả sử tất cả các tiểu trình đã được khởi tạo và sẵn sàng để thực thi. Nếu sử dụng semaphore để đồng bộ hoạt động của các tiểu trình thì phải cần ít nhất bao nhiêu semaphore? (G2)



A. 1

B. 2

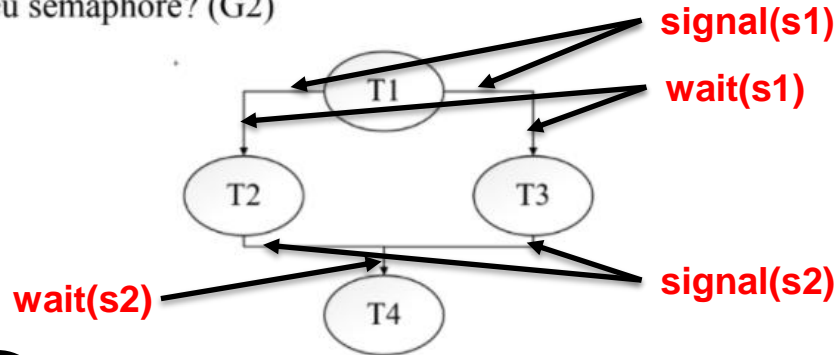
C. 3

D. 4



## Chương 5 : Đồng bộ Bài tập

9. Xét một hệ thống có 4 tiểu trình T1, T2, T3, T4. Quan hệ giữa các tiểu trình này được biểu diễn như sơ đồ bên dưới, với mũi tên từ tiểu trình (Tx) sang tiểu trình (Ty) có nghĩa là tiểu trình Tx phải kết thúc quá trình hoạt động của nó trước khi tiểu trình Ty bắt đầu thực thi. Giả sử tất cả các tiểu trình đã được khởi tạo và sẵn sàng để thực thi. Nếu sử dụng semaphore để đồng bộ hoạt động của các tiểu trình thì phải cần ít nhất bao nhiêu semaphore? (G2)



A. 1

2

C. 3

D. 4

Merry Christmas

*Nội dung thi ( trắc nghiệm )*

Chương 5 : Đồng bộ (5.2 & 5.3)

Chương 6 : Deadlock

Chương 7 : Quản lý bộ nhớ

Chương 8 : Bộ nhớ ảo

Chương 9 : HĐH Linux & Windows

## Chương 6 : DeadLock

- **Deadlock:** Hai hay nhiều process đang chờ đợi vô hạn định một sự kiện không bao giờ xảy ra (vd: sự kiện do một trong các process đang đợi tạo ra).
- **Starvation (indefinite blocking):** Một tiến trình có thể không bao giờ được lấy ra khỏi hàng đợi mà nó bị treo trong hàng đợi đó.



Merry Christmas



## Chương 6 : DeadLock

Điều kiện cần để xảy ra deadlock :

**Mutual exclusion ( độc quyền truy cập )** : tài nguyên chỉ được chia sẻ cho duy nhất 1 process

**Hold and wait ( giữ và chờ )** : mỗi tiến trình đang giữ ít nhất một tài nguyên và đợi thêm tài nguyên khác do tiến trình khác giữ.

**No preemption ( không trưng dụng )** : hệ thống không thể thu hồi tài nguyên mà process không muốn trả

**Circular wait ( chu trình đợi )** : tồn tại một tập  $\{P_0, \dots, P_n\}$  các quá trình đang đợi sao cho :

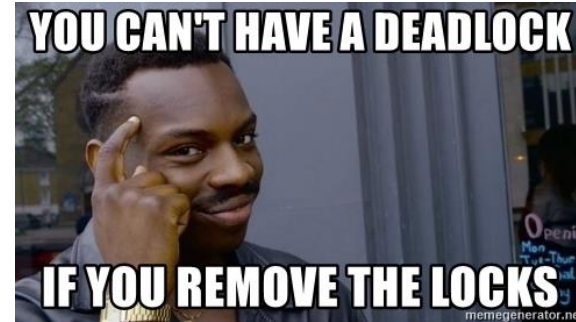
- $P_0$  đợi một tài nguyên mà
- $P_1$  đang giữ  $P_1$  đợi một tài nguyên mà  $P_2$  đang giữ
- $P_n$  đợi một tài nguyên mà  $P_0$  đang giữ



## Chương 6 : DeadLock

### Giải quyết Deadlock :

- Ngăn deadlock : không cho phép (ít nhất) 1 trong 4 điều kiện cần xảy ra.
- Tránh deadlock : thu thập nhu cầu tài nguyên của các tiến trình trước khi thực thi, cấp phát cẩn thận.
- Cho phép deadlock, sau đó tiến hành phát hiện, sửa chữa, phục hồi
- Bỏ qua mọi vấn đề -> Deadlock không được phát hiện -> Giảm hiệu suất hệ thống -< Ngưng hoạt động



Merry Christmas

## Chương 6 : DeadLock - Safe and Unsafe

### Trạng thái Safe :

- Hệ thống có **trạng thái an toàn** khi mà số lượng tài nguyên còn lại (Available) của hệ thống có thể đáp ứng được tất cả các process và tạo ra 1 chuỗi an toàn
- Một chuỗi quá trình  $\langle P_1, P_2, \dots, P_n \rangle$  là một chuỗi an toàn nếu
  - Với mọi  $i = 1, \dots, n$  yêu cầu tối đa về tài nguyên của  $P_i$  có thể được thỏa bởi
    - Tài nguyên mà hệ thống đang có sẵn sàng
    - Cùng với tài nguyên mà tất cả các  $P_j$  ( $j < i$ ) đang giữ
- Một trạng thái của hệ thống được gọi là **không an toàn** (unsafe) nếu không tồn tại một chuỗi an toàn

## Chương 6 : DeadLock - Safe and Unsafe

- VD : Hệ thống có 12 tap drive và 3 tiến trình P0,P1,P2 :
  - Tại thời điểm t0 :

	Cần tối đa	Đang giữ	Cần thêm
P0	10	5	5
P1	4	2	2
P2	9	2	7

- Còn 3 tap drive sẵn sàng.
- Có chuỗi <P1,P0,P2> là chuỗi an toàn -> Hệ thống an toàn.

## Chương 6 : DeadLock - Safe and Unsafe

### Trạng thái Safe :

- Hệ thống có **trạng thái an toàn** khi mà số lượng tài nguyên còn lại (Available) của hệ thống có thể đáp ứng được tất cả các process và tạo ra 1 chuỗi an toàn
- Một chuỗi quá trình  $\langle P_1, P_2, \dots, P_n \rangle$  là một chuỗi an toàn nếu
  - Với mọi  $i = 1, \dots, n$  yêu cầu tối đa về tài nguyên của  $P_i$  có thể được thỏa bởi
    - Tài nguyên mà hệ thống đang có sẵn sàng
    - Cùng với tài nguyên mà tất cả các  $P_j$  ( $j < i$ ) đang giữ
- Một trạng thái của hệ thống được gọi là **không an toàn** (unsafe) nếu không tồn tại một chuỗi an toàn



## *Chương 6 : Deadlock – Giải thuật tránh deadlock*

- Mỗi tài nguyên chỉ có một thực thể
  - Giải thuật đồ thị cấp phát tài nguyên
- Mỗi tài nguyên có nhiều thực thể
  - Giải thuật Banker

## Chương 6 : DeadLock – Giải thuật Banker

- Được sử dụng trong các hệ thống ngân hàng để đảm bảo rằng ngân hàng sẽ không chi tiền đến mức mà nó không thể đáp ứng được nhu cầu của tất cả các khách hàng.

### Điều kiện của giải thuật :

- Mỗi tiến trình phải khai báo số lượng thực thể tối đa của mỗi loại tài nguyên mà nó cần (không được vượt quá số lượng tài nguyên hiện đang có trong hệ thống)
- Khi tiến trình yêu cầu tài nguyên thì có thể phải đợi
- Khi tiến trình đã có được đầy đủ tài nguyên thì phải hoàn trả trong một khoảng thời gian hữu hạn nào đó

## Chương 6 : DeadLock - Giải thuật Banker

### Cấu trúc dữ liệu cho giải thuật Banker :

Gọi m là số loại tài nguyên đang xét trong giải thuật Banker.

Gọi n là số tiến trình đang xét trong giải thuật Banker.

- Mảng Available có m phần tử : Từng phần tử Available[j] chứa một con số tương ứng với số thể hiện của loại tài nguyên j đó.
- Mảng Max[i,j] = k : Tiến trình Pi yêu cầu tối đa k thực thể của loại tài nguyên Rj.
- Allocation[i,j] = k : Pi đã được cấp phát k thực thể của Rj.
- Need[i,j] = k : Pi cần thêm k thực thể của Rj.
- Need[i,j] = Max[i,j] – Allocation[i,j].

Ký hiệu  $Y \leq X \Leftrightarrow Y[i] \leq X[i]$ , ví dụ  $(0, 3, 2, 1) \leq (1, 7, 3, 2)$

Merry Christmas

## Chương 6 : Deadlock - Giải thuật Banker - Các bước thực hiện

1. Gọi **Work** và **Finish** là hai vector độ dài là  $m$  và  $n$ . Khởi tạo

$\text{Work} = \text{Available}$

$\text{Finish}[i] = \text{false}, i = 0, 1, \dots, n-1$

2. Tìm  $i$  thỏa

(a)  $\text{Finish}[i] = \text{false}$

(b)  $\text{Need}_i \leq \text{Work}$  (hàng thứ  $i$  của  $\text{Need}$ )

Nếu không tồn tại  $i$  như vậy, đến bước 4.

3.  $\text{Work} = \text{Work} + \text{Allocation}_i$

$\text{Finish}[i] = \text{true}$

quay về bước 2

4. Nếu  $\text{Finish}[i] = \text{true}, i = 1, \dots, n$ , thì hệ thống đang ở trạng thái safe

Merry Christmas



## Chương 6 : Deadlock - Giải thuật Banker

- 5 tiến trình  $P_0, \dots, P_4$
- 3 loại tài nguyên:
  - A (10 thực thể), B (5 thực thể), C (7 thực thể)
- Sơ đồ cấp phát trong hệ thống tại thời điểm  $T_0$

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Merry Christmas

## Chương 6 : Deadlock - Giải thuật Banker

### Bước 1 :

- Khởi tạo Work = Available ( vector độ dài m – số tài nguyên đang xét = (3,3,2)
- Finish[n] ( vector độ dài n – số tiến trình ) và đặt giá trị là False.

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

## Chương 6 : Deadlock - Giải thuật Banker

### BƯỚC 2 :

Tìm  $i$  thỏa  $\text{Finish}[i] = \text{false}$  và  $\text{Need}_i \leq \text{Work}$  (hàng thứ  $i$  của Need)

Có P1 và P3 thỏa  $\rightarrow$  Chọn P1 , chuỗi safe <P1>

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Merry Christmas

## Chương 6 : Deadlock - Giải thuật Banker

### BƯỚC 3 :

Work = Work + Allocation<sub>i</sub>    Finish[*I*] = **true**

Quay về bước 2

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3				7	4	3
P1	2	0	0	3	2	2	5	3	2	1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Merry Christmas



## Chương 6 : Deadlock - Giải thuật Banker

### BƯỚC 2 :

Tìm  $i$  thỏa  $Finish[i] = \text{false}$  và  $Need_i \leq Work$  (hàng thứ  $i$  của Need)

Có P3 và P4 thỏa  $\rightarrow$  Chọn P3 , chuỗi safe  $\langle P1, P3 \rangle$

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3				7	4	3
P1	0	0	0	3	2	2	5	3	2	1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

## Chương 6 : Deadlock - Giải thuật Banker

### BƯỚC 3 :

Work = Work + Allocation<sub>i</sub>     Finish[3] = **true**

Quay về bước 2

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3				7	4	3
P1	0	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	0	0	0	2	2	2	7	4	3	0	1	1
P4	0	0	2	4	3	3				4	3	1

Merry Christmas

## Chương 6 : Deadlock - Giải thuật Banker

### BƯỚC 3 :

Work = Work + Allocation<sub>i</sub>    Finish[0] = **true**

Quay về bước 2

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	5	3	7	4	3
P1	0	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	0	0	0	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Merry Christmas

## Chương 6 : Deadlock - Giải thuật Banker

### BƯỚC 3 :

Work = Work + Allocation<sub>i</sub>     Finish[2] = **true**

Quay về bước 2

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	0	0	7	5	3				7	4	3
P1	0	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2	10	5	5	6	0	0
P3	0	0	0	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Merry Christmas



## Chương 6 : Deadlock - Giải thuật Banker

### BƯỚC 3 :

Work = Work + Allocation<sub>i</sub>      Finish[4] = **true**

Quay về bước 2

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	0	0	7	5	3				7	4	3
P1	0	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	0	0	0	2	2	2				0	1	1
P4	0	0	2	4	3	3	10	5	7	4	3	1

Merry Christmas

## Chương 6 : DeadLock - Giải thuật Banker

Không còn process nào thỏa mãn yêu cầu

Finish[i] = true -> hệ thống đang ở trạng thái safe -> Chuỗi an toàn <P1,P3,P0,P2,P4>

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	0	0	7	5	3				7	4	3
P1	0	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	0	0	0	2	2	2				0	1	1
P4	0	0	2	4	3	3	10	5	7	4	3	1

## Chương 6 : DeadLock - Giải thuật Banker

### Phục hồi khi deadlock xảy ra :

Khi xảy ra deadlock giải thuật phát hiện deadlock xác định rằng deadlock đang tồn tại trong hệ thống

Báo cho người vận hành

Hệ thống tự động phục hồi bằng cách bẻ gãy chu trình deadlock :

- Huỷ bỏ một trong các process đang tham gia vào deadlock để phá Circular Wait.
- Trưng dụng (Preempt) một vài tài nguyên đang bị nắm giữ bởi process này cho process kia dùng trước.

## Chương 5 : Đồng bộ Bài tập

“Các tiến trình cần cung cấp thông tin về tài nguyên nó cần để hệ thống cấp phát tài nguyên một cách thích hợp” là đặc điểm của phương pháp giải quyết deadlock nào?

- A. Ngăn deadlock
- B. Tránh deadlock**
- C. Bỏ qua deadlock
- D. Phát hiện deadlock và phục hồi



## Chương 5 : Đồng bộ Bài tập

“Không cho phép (ít nhất) một trong 4 điều kiện cần cho deadlock xảy ra” là đặc điểm của phương pháp giải quyết deadlock nào?

- A. Ngăn deadlock
- B. Tránh deadlock
- C. Bỏ qua deadlock
- D. Phát hiện deadlock và phục hồi

Merry Christmas

## Chương 5 : Đồng bộ Bài tập

Lựa chọn nào dưới đây **KHÔNG** phải là điều kiện cần để thực hiện giải thuật Banker?

- A. Mỗi tiến trình phải khai báo số lượng thực thể tối đa của mỗi loại tài nguyên mà nó cần.
- B. Khi yêu cầu tài nguyên, tiến trình không được giữ tài nguyên nào.
- C. Khi tiến trình đã có được đầy đủ tài nguyên thì phải hoàn trả trong một khoảng thời gian hữu hạn nào đó.
- D. Khi tiến trình yêu cầu tài nguyên thì nó có thể phải đợi.

## Chương 5 : Đồng bộ Bài tập

Chọn phát biểu SAI trong các phát biểu bên dưới?

- A. Nếu hệ thống đang ở trạng thái an toàn thì không có deadlock trong hệ thống.
- B. Nếu hệ thống đang ở trạng thái không an toàn thì có deadlock trong hệ thống.**
- C. Nếu đồ thị cấp phát tài nguyên không chứa chu trình thì không có deadlock trong hệ thống.
- D. Nếu đồ thị cấp phát tài nguyên có một chu trình thì deadlock có thể xảy ra trong hệ thống.

## Chương 5 : Đồng bộ Bài tập

Cho các giải pháp sau:

- (1) Báo người vận hành. (2) Cung cấp thêm tài nguyên.
- (3) Chấm dứt một hay nhiều tiến trình. (4) Lấy lại tài nguyên từ một hay nhiều tiến trình.

Khi xảy ra deadlock, các giải pháp nào có thể được sử dụng để phục hồi hệ thống?

- A. (1), (2), (3)      B. (1), (3), (4)      C. (2), (3), (4)      D. (1), (2), (4)



## Chương 5 : Đồng bộ Bài tập

Giải pháp tập làm việc được sử dụng để giải quyết vấn đề gì?

- A. Phát hiện deadlock
- B. Trì trệ trên toàn bộ hệ thống do hoán chuyển trang nhớ
- C. Đồng bộ hoạt động giữa các tiến trình
- D. Thay thế trang nhớ

Merry Christmas

## Chương 5 : Đồng bộ Bài tập

Cho các đồ thị cấp phát tài nguyên sau, trong đó  $T_1, T_2, T_3, T_4$  là các tiến trình còn  $R_1, R_2, R_3$  là loại tài nguyên. Hỏi đồ thị nào có deadlock xảy ra?

- A. (a)(b) B. (c) (d)  
C. (a)(c) D. **(b)(d)**

