



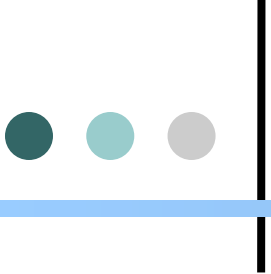
Chương 3

QUẢN LÝ TIỀN TRÌNH



Nội dung chương 3

1. Khái niệm về tiến trình (process).
2. Tiểu trình (thread).
3. Điều phối tiến trình.
4. Đồng bộ tiến trình.
5. Tình trạng tắc nghẽn (deadlock)



Đồng bộ tiến trình

❖ Liên lạc giữa các tiến trình

➤ Mục đích:

- ✓ Để chia sẻ thông tin như dùng chung file, bộ nhớ,...
- ✓ Hoặc hợp tác hoàn thành công việc

➤ Các cơ chế:

- ✓ Liên lạc bằng tín hiệu (Signal)
- ✓ Liên lạc bằng đường ống (Pipe)
- ✓ Liên lạc qua vùng nhớ chia sẻ (shared memory)
- ✓ Liên lạc bằng thông điệp (Message)
- ✓ Liên lạc qua socket

Đồng bộ tiến trình

❖ Liên lạc bằng tín hiệu (Signal)

Tín hiệu	Mô tả
SIGINT	Người dùng nhấn phím Ctl-C để ngắt xử lý tiến trình
SIGILL	Tiến trình xử lý một chỉ thị bất hợp lệ
SIGKILL	Yêu cầu kết thúc một tiến trình
SIGFPT	Lỗi chia cho 0
SIGSEGV	Tiến trình truy xuất đến một địa chỉ bất hợp lệ
SIGCLD	Tiến trình con kết thúc

Tín hiệu được gửi đi bởi:

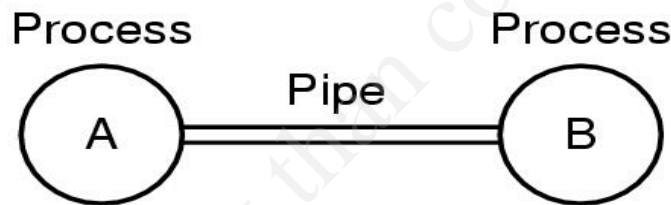
- Phần cứng
- Hệ điều hành:
- Tiến trình:
- Người sử dụng:

Khi tiến trình nhận tín hiệu:

- Gọi hàm xử lý tín hiệu.
- Xử lý theo cách riêng của tiến trình.
- Bỏ qua tín hiệu.

Đồng bộ tiến trình

❖ Liên lạc bằng đường ống (Pipe)



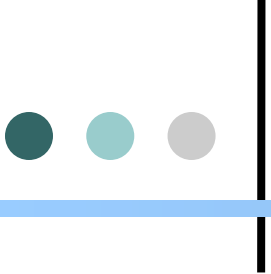
- Dữ liệu truyền: dòng các byte (FIFO)
- Tiến trình đọc pipe sẽ bị khóa nếu pipe trống, và đợi đến khi pipe có dữ liệu mới được truy xuất.
- Tiến trình ghi pipe sẽ bị khóa nếu pipe đầy, và đợi đến khi pipe có chỗ trống để chứa dữ liệu.

Đồng bộ tiến trình

- ❖ Liên lạc qua vùng nhớ chia sẻ (shared memory)
 - Vùng nhớ chia sẻ độc lập với các tiến trình
 - Tiến trình phải gắn kết vùng nhớ chung vào không gian địa chỉ riêng của tiến trình

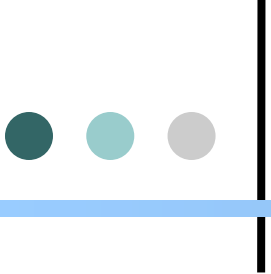


- Vùng nhớ chia sẻ là:
 - ✓ Phương pháp nhanh nhất để trao đổi dữ liệu giữa các tiến trình.
 - ✓ Cần được bảo vệ bằng những cơ chế đồng bộ hóa.
 - ✓ Không thể áp dụng hiệu quả trong các hệ phân tán



Đồng bộ tiến trình

- ❖ Liên lạc bằng thông điệp (Message)
 - Thiết lập một mối liên kết giữa hai tiến trình
 - Sử dụng các hàm send, receive do hệ điều hành cung cấp để trao đổi thông điệp
 - Cách liên lạc bằng thông điệp:
 - ✓ Liên lạc gián tiếp (indirect communication)
 - Send(A, message): gửi thông điệp tới port A
 - Receive(A, message): nhận thông điệp từ port A
 - ✓ Liên lạc trực tiếp (direct communication)
 - Send(P, message): gửi thông điệp đến process P
 - Receive(Q, message): nhận thông điệp từ process Q

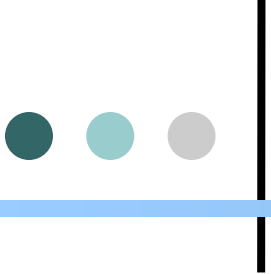


Đồng bộ tiến trình

Ví dụ: Bài toán nhà sản xuất - người tiêu thụ
(producer-consumer)

```
void nsx()  
{ while(1)  
  { tạo_sp();  
    send(ntt,sp); //gởi sp cho ntt  }}
```


```
void ntt()  
{ while(1)  
  { receive(nsx,sp); //ntt chờ nhận sp  
    tiêu_thụ(sp); }}
```

Đồng bộ tiến trình

❖ Liên lạc qua socket

- Mỗi tiến trình cần tạo một socket riêng
- Mỗi socket được kết buộc với một cổng khác nhau.
- Các thao tác đọc/ghi lên socket chính là sự trao đổi dữ liệu giữa hai tiến trình.
- Cách liên lạc qua socket:
 - ✓ **Liên lạc kiểu thư tín (socket đóng vai trò bưu cục)**
 - “tiến trình gởi” ghi dữ liệu vào socket của mình, dữ liệu sẽ được chuyển cho socket của “tiến trình nhận”
 - “tiến trình nhận” sẽ nhận dữ liệu bằng cách đọc dữ liệu từ socket của “tiến trình nhận”

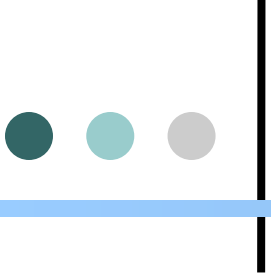


Đồng bộ tiến trình

❖ Liên lạc qua socket

➤ Cách liên lạc qua socket (tt):

- ✓ Liên lạc kiểu điện thoại (socket đóng vai trò tổng đài)
 - Hai tiến trình cần kết nối trước khi truyền/nhận dữ liệu và kết nối được duy trì suốt quá trình truyền nhận dữ liệu



Đồng bộ tiến trình

- ❖ Bảo đảm các tiến trình xử lý song song không tác động sai lệch đến nhau.
- ❖ **Yêu cầu độc quyền truy xuất (Mutual exclusion)**: tại một thời điểm, chỉ có một tiến trình được quyền truy xuất một tài nguyên không thể chia sẻ.
- ❖ **Yêu cầu phối hợp (Synchronization)**: các tiến trình cần hợp tác với nhau để hoàn thành công việc.
- ❖ Hai “bài toán đồng bộ” cần giải quyết:
bài toán “độc quyền truy xuất” (“**bài toán miền găng**”)
bài toán “phối hợp thực hiện”.

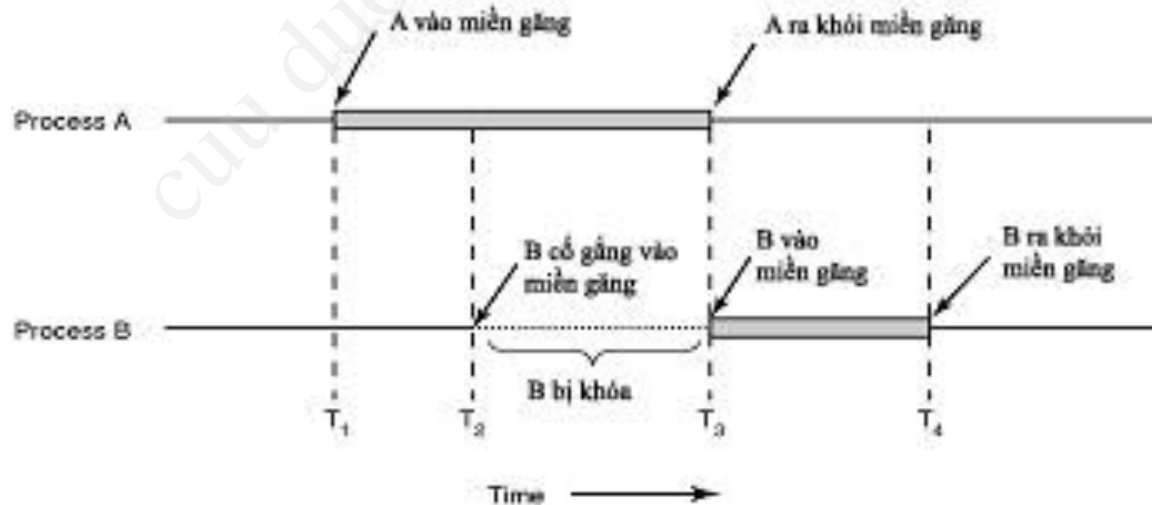
Đồng bộ tiến trình

❖ Miền găng (critical section)

- Đoạn mã của một tiến trình có khả năng xảy ra lỗi khi truy xuất tài nguyên dùng chung (biến, tập tin,...).

- Ví dụ:

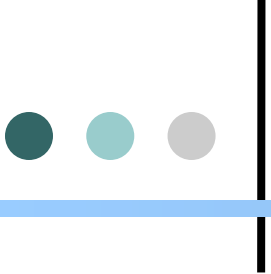
if (taikhoan >= tienrut) taikhoan = taikhoan - tienrut;
else Thông báo “không thể rút tiền !”;





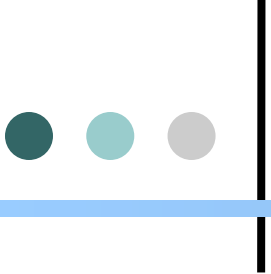
Đồng bộ tiến trình

- ❖ Các điều kiện cần khi giải quyết bài toán miền găng
 1. Không có giả thiết về tốc độ của các tiến trình, cũng như về số lượng bộ xử lý.
 2. Không có hai tiến trình cùng ở trong miền găng cùng lúc.
 3. Một tiến trình bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng.
 4. Không có tiến trình nào phải chờ vô hạn để được vào miền găng



Đồng bộ tiến trình

- ❖ Các nhóm giải pháp đồng bộ
 - Busy Waiting
 - Sleep And Wakeup
 - ✓ Semaphore
 - ✓ Monitor
 - Message.



Đồng bộ tiến trình

- ❖ Busy Waiting (bận thì đợi)
 - Giải pháp phần mềm
 - ✓ Thuật toán sử dụng biến cờ hiệu
 - ✓ Thuật toán sử dụng biến luân phiên
 - ✓ Thuật toán Peterson
 - Giải pháp phần cứng
 - ✓ Cấm ngắt
 - ✓ Sử dụng lệnh TSL (Test and Set Lock)

Đồng bộ tiến trình

- ❖ Thuật toán sử dụng biến cờ hiệu (*dùng cho nhiều tiến trình*)
 - ✓ lock=0 là không có tiến trình trong miền găng.
 - ✓ lock=1 là có một tiến trình trong miền găng.

```
lock=0;
while (1)
{
    while (lock == 1);
    lock = 1;
    critical-section ();
    lock = 0;
    noncritical-section();
}
```

Vi phạm : “Hai tiến trình có thể cùng ở trong miền găng tại một thời điểm”.

Đồng bộ tiến trình

❖ Thuật toán sử dụng biến luân phiên (*dùng cho 2 tiến trình*)

Hai tiến trình A, B sử dụng chung biến turn:

- ✓ turn = 0, tiến trình A được vào miền găng
- ✓ turn=1 thì B được vào miền găng

// tiến trình A

```
while (1) {  
    while (turn == 1);  
    critical-section ();  
    turn = 1;  
    Noncritical-section ();  
}
```

turn=0 thì A
được vào
miền găng

// tiến trình B

```
while (1) {  
    while (turn == 0);  
    critical-section ();  
    turn = 0;  
    Noncritical-section ();  
}
```

turn=1 thì B
được vào
miền găng

- Hai tiến trình chắc chắn không thể vào miền găng cùng lúc, vì tại một thời điểm turn chỉ có một giá trị.
- Vi phạm: một tiến trình có thể bị ngăn chặn vào miền găng bởi một tiến trình khác không ở trong miền găng.

Đồng bộ tiến trình

❖ Thuật toán **Peterson** (dùng cho 2 tiến trình)

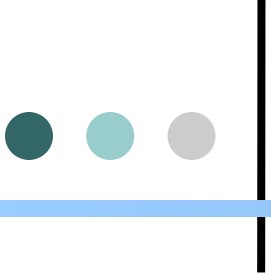
- Dùng chung hai biến turn và flag[2] (kiểu int).
 - ✓ $\text{flag}[0] = \text{flag}[1] = \text{FALSE}$
 - ✓ turn được khởi động là 0 hay 1.
- Nếu $\text{flag}[i] = \text{TRUE}$ ($i=0,1$) $\rightarrow P_i$ muốn vào miền găng và $\text{turn}=i$ là đến lượt P_i .
- Để có thể vào được miền găng:
 - ✓ P_i đặt trị $\text{flag}[i] = \text{TRUE}$ để thông báo nó muốn vào miền găng.
 - ✓ Đặt $\text{turn}=j$ để thử đề nghị tiến trình P_j vào miền găng.
- Nếu tiến trình P_j không quan tâm đến việc vào miền găng ($\text{flag}[j] = \text{FALSE}$), thì P_i có thể vào miền găng
 - ✓ Nếu $\text{flag}[j] = \text{TRUE}$ thì P_i phải chờ đến khi $\text{flag}[j] = \text{FALSE}$.
- Khi tiến trình P_i ra khỏi miền găng, nó đặt lại trị $\text{flag}[i]$ là FALSE.

Đồng bộ tiến trình

❖ Thuật toán **Peterson** (*đoạn code*)

```
// tiến trình P0 (i=0)
while (TRUE)
{
    flag [0]= TRUE; //P0 muốn vào
                    //miền găng
    turn = 1; //thu de nghi P1 vào
    while (turn==1 &&
           flag[1]==TRUE);
    //neu P1 muon vào thì P0 chờ
    critical_section();
    flag [0] = FALSE; //P0 ra ngoài mg
    noncritical_section ();
}
```

```
// tiến trình P1 (i=1)
while (TRUE)
{
    flag [1]= TRUE; //P1 muon vào
                    //miền găng
    turn = 0; //thử de nghi P0 vào
    while (turn == 0 && flag
           [0]==TRUE); //neu P0 muon vào
                    //thì P1 chờ
    critical_section();
    flag [1] = FALSE; //P1 ra ngoài mg
    Noncritical_section ();
}
```



Đồng bộ tiến trình

❖ Cấm ngắt

- Tiến trình cấm tất cả các ngắt trước khi vào miền găng, và phục hồi ngắt khi ra khỏi miền găng.
 - ✓ Không an toàn cho hệ thống
 - ✓ Không tác dụng trên hệ thống có nhiều bộ xử lý

Đồng bộ tiến trình

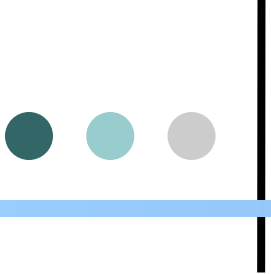
❖ Sử dụng lệnh TSL (*Test and Set Lock*)

```
boolean Test_And_Set_Lock (boolean lock){  
    boolean temp=lock;  
    lock = TRUE;  
    return temp; //trả về giá trị ban đầu của  
                //biến lock  
}
```

```
boolean lock=FALSE; //biến dùng chung  
while (TRUE) {  
    while (Test_And_Set_Lock(lock));  
    critical_section ();  
    lock = FALSE;  
    noncritical_section ();  
}
```

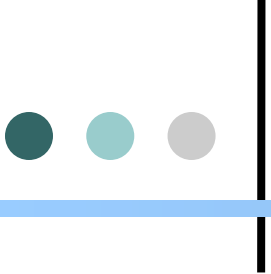
❑ Lệnh TSL cho phép kiểm tra và cập nhật một vùng nhớ trong một thao tác độc quyền.

❑ Hoạt động trên hệ thống có nhiều bộ xử lý.



Đồng bộ tiến trình

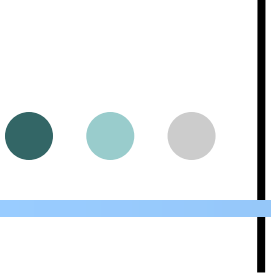
- ❖ Nhóm giải pháp “SLEEP and WAKEUP “ (ngủ và đánh thức)
 1. Sử dụng lệnh SLEEP VÀ WAKEUP
 2. Sử dụng cấu trúc Semaphore
 3. Sử dụng cấu trúc Monitors
 4. Sử dụng thông điệp



Đồng bộ tiến trình

❖ Sử dụng lệnh SLEEP VÀ WAKEUP

- SLEEP → “danh sách sẵn sàng”, lấy lại CPU cấp cho P khác.
- WAKEUP → HĐH chọn một P trong ready list, cho thực hiện tiếp.
- P chưa đủ điều kiện vào miền găng → gọi SLEEP để tự khóa, đến khi có P khác gọi WAKEUP để giải phóng cho nó.
- Một tiến trình gọi WAKEUP khi ra khỏi miền găng để đánh thức một tiến trình đang chờ, tạo cơ hội cho tiến trình này vào miền găng



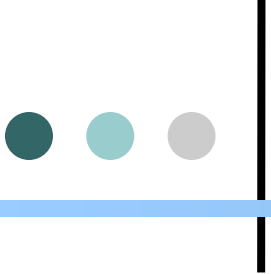
Đồng bộ tiến trình

❖ Sử dụng lệnh SLEEP VÀ WAKEUP

int busy=FALSE; // TRUE là có tiến trình trong miền găng, FALSE là không có

int blocked=0; // đếm số lượng tiến trình đang bị khóa

```
while(TRUE){  
    if (busy) {  
        blocked = blocked + 1;  
        sleep();  
    }else busy = TRUE;  
    critical-section ();  
    busy = FALSE;  
    if (blocked>0){  
        wakeup(); //đánh thức một tiến trình đang chờ  
        blocked = blocked - 1;  
    }  
    Noncritical-section ();  
}
```

Đồng bộ tiến trình

❖ Sử dụng cấu trúc Semaphore

- Biến semaphore s có các thuộc tính:
 - ✓ Một giá trị nguyên dương e ;
 - ✓ Một hàng đợi f : lưu danh sách các tiến trình đang chờ trên semaphore s .
- Hai thao tác trên semaphore s :
 - ✓ Down(s): $e=e-1$.
 - Nếu $e < 0$ thì tiến trình phải chờ trong f (sleep), ngược lại tiến trình tiếp tục.
 - ✓ Up(s): $e=e+1$.
 - Nếu $e \leq 0$ thì chọn một tiến trình trong f cho tiếp tục thực hiện (đánh thức).

Đồng bộ tiến trình

❖ Sử dụng cấu trúc Semaphore

```
Down(s) {
```

```
  e = e - 1;
```

```
  if(e < 0) {
```

```
    status(P)= blocked; //khóa P (trạng thái chờ)
```

```
    enter(P,f); //cho P vào hàng đợi f
```

```
  }
```

```
}
```

```
Up(s){
```

```
  e = e + 1;
```

```
  if(e <= 0 ){
```

```
    exit(Q,f); //lấy một tt Q ra khỏi hàng đợi f
```

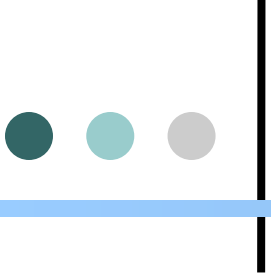
```
    status (Q) = ready; //chuyển Q sang trạng thái sẵn sàng
```

```
    enter(Q,ready-list); //đưa Q vào danh sách sẵn sàng
```

```
  }
```

```
}
```

P là tiến trình thực hiện thao tác Down(s) hay Up(s)



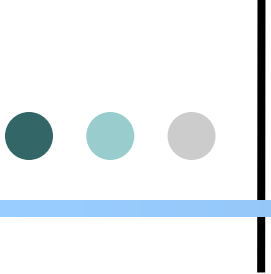
Đồng bộ tiến trình

❖ Sử dụng cấu trúc Semaphore

- Hệ điều hành cần cài đặt các thao tác Down, Up là độc quyền.
- Cấu trúc của semaphore:

```
class semaphore{  
    int e;  
    PCB * f; //ds riêng của semaphore  
public:  
    down();  
    up();  
};
```

- $|e|$ = số tiến trình đang chờ trên f.



Đồng bộ tiến trình

❖ Giải quyết bài toán miền găng bằng Semaphores

- Dùng một semaphore s , e được khởi gán là 1.
- Tất cả các tiến trình áp dụng cùng cấu trúc chương trình sau:

```
semaphore s=1; //nghĩa là e của s=1
```

```
while (1)
```

```
{
```

```
    Down(s);
```

```
    critical-section ();
```

```
    Up(s);
```

```
    Noncritical-section ();
```

```
}
```

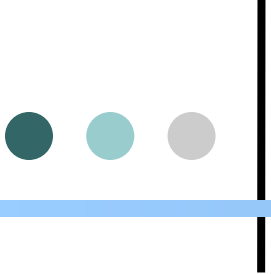
Đồng bộ tiến trình

❖ Giải quyết bài toán đồng bộ bằng Semaphores

- Hai tiến trình đồng hành P1 và P2, P1 thực hiện công việc 1, P2 thực hiện công việc 2.
- Cv1 làm trước rồi mới làm cv2, cho P1 và P2 dùng chung một semaphore s, khởi gán $e(s) = 0$:

semaphore s=0; //dùng chung cho hai tiến trình

```
P1:{  
    job1();  
    Up(s); //đánh thức P2  
}  
P2:{  
    Down(s); // chờ P1 đánh thức  
    job2();  
}
```

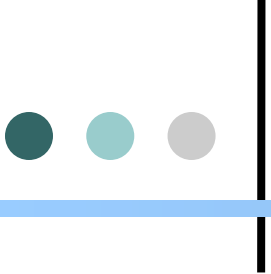


Đồng bộ tiến trình

❖ Vấn đề khi sử dụng semaphore

- Tiến trình quên gọi Up(s), và kết quả là khi ra khỏi miền găng nó sẽ không cho tiến trình khác vào miền găng!

```
e(s)=1;  
while (1)  
{  
    Down(s);  
    critical-section ();  
    Noncritical-section ();  
}
```



Đồng bộ tiến trình

❖ Vấn đề khi sử dụng semaphore

- Sử dụng semaphore có thể gây ra tình trạng tắc nghẽn.

P1:

{

down(s1); down(s2);

....

up(s1); up(s2);

}

P2:

{

down(s2); down(s1);

....

up(s2); up(s1);

}

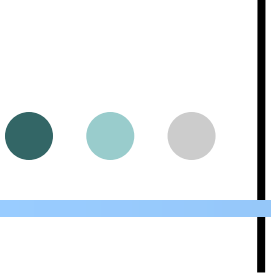
Hai tiến trình P1, P2 sử dụng chung 2 semaphore $s1=s2=1$

Nếu thứ tự thực hiện như sau:

P1: down(s1), P2: down(s2) ,

P1: down(s2), P2: down(s1)

Khi đó $s1=s2=-1$ nên P1,P2 đều chờ mãi



Đồng bộ tiến trình

❖ Sử dụng cấu trúc Monitors

- Hoare(1974) và Brinch & Hansen (1975) đã đề nghị một cơ chế cao hơn gọi à monitor được cung cấp bởi 1 số ngôn ngữ lập trình
- Monitor là một cấu trúc đặc biệt (lớp)
 - ✓ các phương thức độc quyền (critical-section)
 - ✓ các biến (được dùng chung cho các tiến trình)
 - Các biến trong monitor chỉ có thể được truy xuất bởi các phương thức trong monitor
 - ✓ Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor.
 - ✓ Biến điều kiện c
 - dùng để đồng bộ việc sử dụng các biến trong monitor.
 - Wait(c) và Signal(c):

Đồng bộ tiến trình

❖ Sử dụng cấu trúc Monitors

Wait(c)
{

Wait(c): chuyển trạng thái tiến trình gọi sang chờ (blocked) và đặt tiến trình này vào hàng đợi trên biến điều kiện c.

status(P)= blocked; //chuyển P sang trạng thái chờ
enter(P,f(c)); //đặt P vào hàng đợi f(c) của biến điều kiện c

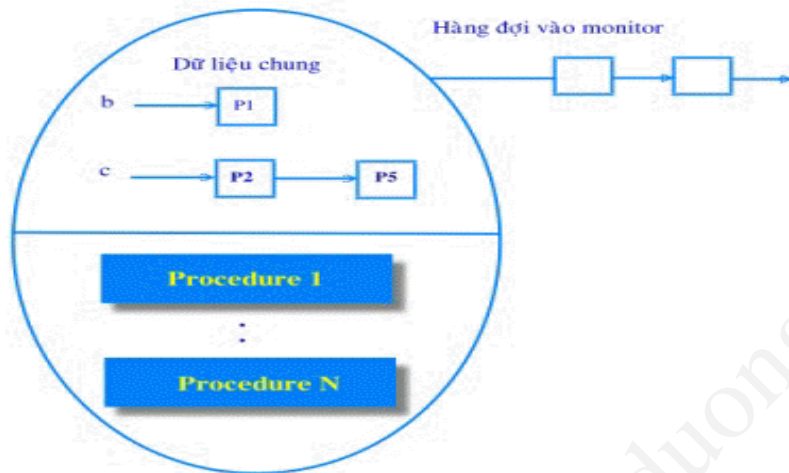
}
Signal(c)
{

Signal(c): khi có một tiến trình đang chờ trong hàng đợi c, kích hoạt tiến trình đó và tiến trình gọi sẽ rời khỏi monitor.

if (f(c) != NULL){
 exit(Q,f(c)); //Lấy tiến trình Q đang chờ trên c
 status(Q) = ready; //chuyển Q sang trạng thái sẵn sàng
 enter(Q,ready-list); //đưa Q vào danh sách sẵn sàng.
}
}

Đồng bộ tiến trình

❖ Sử dụng cấu trúc Monitors



monitor <tên monitor> //khai báo monitor
dùng chung cho các tiến trình

```
{  
    <cac bien dung chung>;  
    <cac bien điều kiện>;  
    <cac phuong thuc độc quyền>;  
}
```

//tiến trình Pi:

```
while (1) //cấu trúc tiến trình thứ i  
{
```

```
    Noncritical-section ();
```

```
    <ten monitor>.Phương thức_i; //thực  
    hiện công việc độc quyền thứ i
```

```
    Noncritical-section ();
```

```
}
```

critical-section



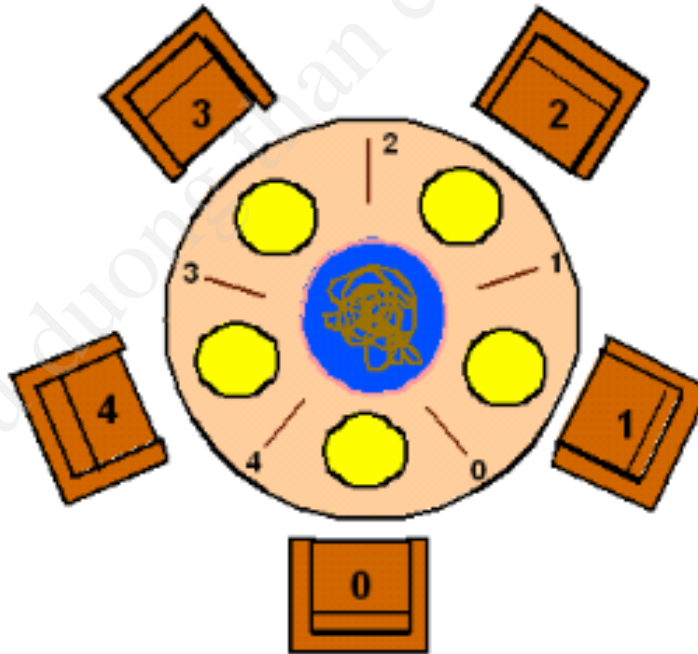
Đồng bộ tiến trình

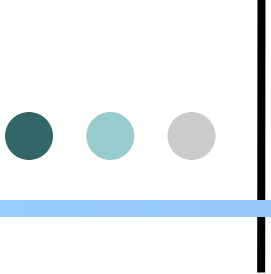
❖ Sử dụng cấu trúc Monitors

- Nguy cơ thực hiện đồng bộ hóa sai giảm rất nhiều.
- Ít có ngôn ngữ hỗ trợ cấu trúc monitor.

Đồng bộ tiến trình

❖ Monitors và Bài toán 5 triết gia ăn tối





Đồng bộ tiến trình

❖ Monitors và Bài toán 5 triết gia ăn tối

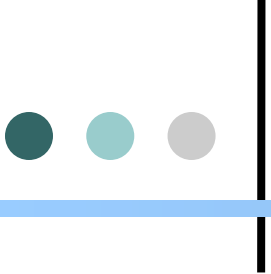
```
monitor philosopher{  
    enum {thinking, hungry, eating} state[5]; // các biến dùng chung  
                                                //cho các triết gia  
    condition self[5]; //các biến điều kiện dùng để đồng bộ việc ăn tối  
    //cac pt doc quyen (cac mien gang), viec doc quyen do nnlt ho  
    tro.  
    void init(); //phương thức khởi tạo  
    void test(int i); //dùng kiểm tra điều kiện trước khi triết gia thứ i ăn  
    void pickup(int i); //phương thức lấy đĩa  
    void putdown(int i); //phương thức trả đĩa  
}
```

Đồng bộ tiến trình

❖ Monitors và Bài toán 5 triết gia ăn tối

```
void philosopher() //phương thức khởi tạo (constructor)
{
    //gán trạng thái ban đầu cho các triết gia là "đang suy nghĩ"
    for (int i = 0; i < 5; i++) state[i] = thinking;
}

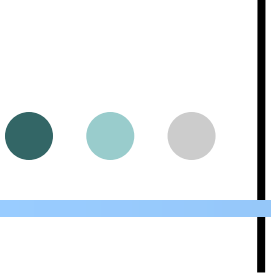
void test(int i) { //nếu tg_i đói và các tg bên trái, bên phải không đang
                  //ăn thì cho tg_i ăn
    if ( (state[i] == hungry) && (state[(i + 4) % 5] != eating) &&(state[(i
+ 1) % 5] != eating)) {
        self[i].signal();//đánh thức tg_i, nếu tg_i đang chờ
        state[i] = eating; //ghi nhận tg_i đang ăn
    }
}
```



Đồng bộ tiến trình

❖ Monitors và Bài toán 5 triết gia ăn tối

```
void pickup(int i)
{
    state[i] = hungry; //ghi nhận tg_i đói
    test(i); //kiểm tra đk trước khi cho tg_i ăn
    if (state[i] != eating) self[i].wait(); //đòi tài nguyên
}
void putdown(int i)
{
    state[i] = thinking; //ghi nhận tg_i đang suy nghĩ
    test((i+4) % 5); //kt tg bên phải, nếu hợp lệ thì cho tg này ăn
    test((i+1) % 5); //kt tg bên trái nếu hợp lệ thì cho tg này ăn
}
```



Đồng bộ tiến trình

❖ Monitors và Bài toán 5 triết gia ăn tối

// Cấu trúc tiến trình Pi thực hiện việc ăn của triết gia thứ i

philosopher pp; //biến monitor dùng chung

Pi:

while(1) {

 Noncritical-section();

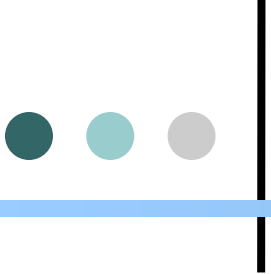
 pp.pickup(i); //pickup là miền găng và được truy xuất độc quyền

 eat(); //triết gia ăn

 pp.putdown(i); //putdown là miền găng và được truy xuất độc quyền

 Noncritical-section();

}



Đồng bộ tiến trình

❖ Sử dụng thông điệp

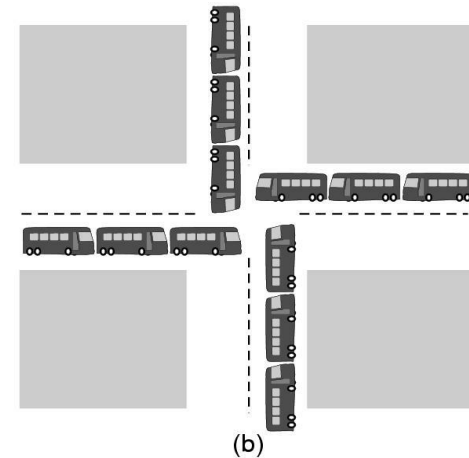
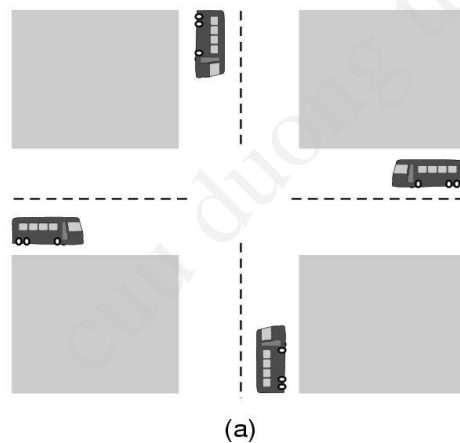
- Một tiến trình kiểm soát việc sử dụng tài nguyên và nhiều tiến trình khác yêu cầu tài nguyên.

```
while (1) {  
    Send(process controller, request message); //gửi thông điệp cần tài nguyên  
                                                //và chuyển sang blocked  
    Receive(process controller, accept message); //nhận thông điệp  
    critical-section(); //đọc quyền sử dụng tài nguyên chung  
    Send(process controller, end message); //gửi tin kết thúc sử dụng tn.  
    Noncritical-section();  
}
```

- Trong những hệ thống phân tán, cơ chế trao đổi thông điệp sẽ đơn giản hơn và được dùng để giải quyết bài toán đồng bộ hóa.

Tình trạng tắc nghẽn (deadlock)

- ❖ Một tập hợp các tiến trình gọi là ở tình trạng tắc nghẽn nếu mỗi tiến trình trong tập hợp đều chờ đợi tài nguyên mà tiến trình khác trong tập hợp đang chiếm giữ.





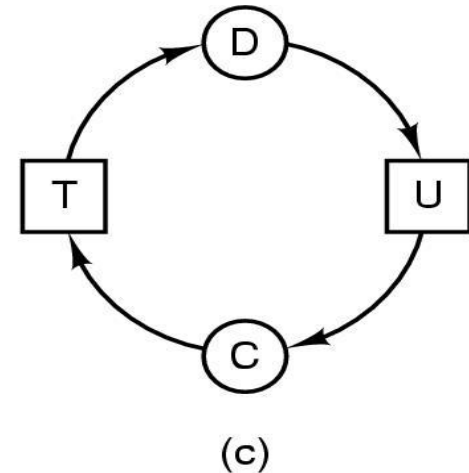
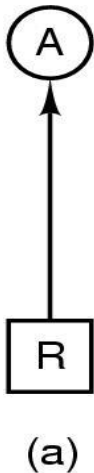
Tình trạng tắc nghẽn (deadlock)

❖ Điều kiện xuất hiện tắc nghẽn

1. **Điều kiện 1:** Có sử dụng tài nguyên không thể chia sẻ.
2. **Điều kiện 2:** Sự chiếm giữ và yêu cầu thêm tài nguyên không thể chia sẻ.
3. **Điều kiện 3:** Không thu hồi tài nguyên từ tiến trình đang giữ chúng.
4. **Điều kiện 4:** Tồn tại một chu trình trong đồ thị cấp phát tài nguyên.

Tình trạng tắc nghẽn (deadlock)

❖ Đồ thị cấp phát tài nguyên



Tình trạng tắc nghẽn (deadlock)

A
Request R
Request S
Release R
Release S

(a)

B
Request S
Request T
Release S
Release T

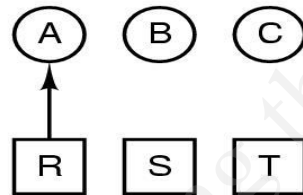
(b)

C
Request T
Request R
Release T
Release R

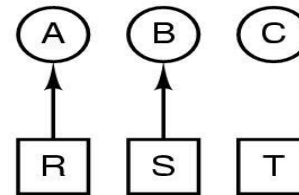
(c)

1. A requests R
 2. B requests S
 3. C requests T
 4. A requests S
 5. B requests T
 6. C requests R
- deadlock

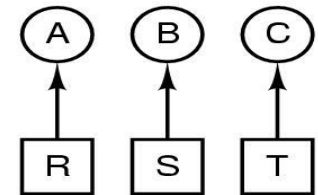
(d)



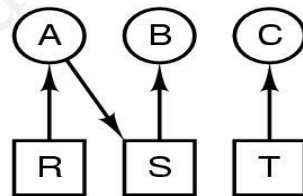
(e)



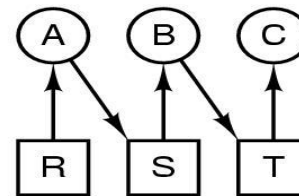
(f)



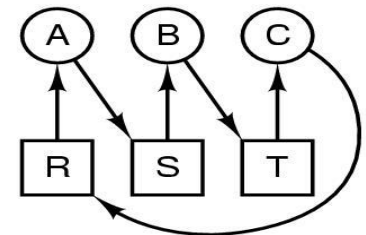
(g)



(h)



(i)



(j)



Tình trạng tắc nghẽn (deadlock)

- ❖ Các phương pháp xử lý tắc nghẽn và ngăn chặn tắc nghẽn
 - Sử dụng một thuật toán cấp phát tài nguyên -> không bao giờ xảy ra tắc nghẽn.
 - Cho phép xảy ra tắc nghẽn -> tìm cách sửa chữa tắc nghẽn.
 - Bỏ qua việc xử lý tắc nghẽn, xem như hệ thống không bao giờ xảy ra tắc nghẽn.

Tình trạng tắc nghẽn (deadlock)

❖ Ngăn chặn tắc nghẽn

- Điều kiện 1 gần như không thể tránh được.
- Để điều kiện 2 không xảy ra:
 - ✓ Tiến trình phải yêu cầu tất cả các tài nguyên cần thiết trước khi cho bắt đầu xử lý.
 - ✓ Khi tiến trình yêu cầu một tài nguyên mới và bị từ chối
 - Giải phóng các tài nguyên đang chiếm giữ
 - Tài nguyên cũ được cấp lại cùng với tài nguyên mới.
- Để điều kiện 3 không xảy ra:
 - ✓ Thu hồi tài nguyên từ các tiến trình bị khoá và cấp phát trở lại cho tiến trình khi nó thoát khỏi tình trạng bị khoá.
- Để điều kiện 4 không xảy ra:
 - ✓ Khi tiến trình đang chiếm giữ tài nguyên R_i thì chỉ có thể yêu cầu các tài nguyên R_j nếu $F(R_j) > F(R_i)$.



Tình trạng tắc nghẽn (deadlock)

- ❖ Giải thuật cấp phát tài nguyên tránh tắc nghẽn
 - Giải thuật xác định trạng thái an toàn
 - Giải thuật Banker

Tình trạng tắc nghẽn (deadlock)

❖ Giải thuật xác định trạng thái an toàn

```
int NumResources;           //số tài nguyên
int NumProcs;               //số tiến trình trong hệ thống
int Available[NumResources] //số lượng tài nguyên còn tự do
int Max[NumProcs, NumResources];
    //Max[p,r]= nhu cầu tối đa của tiến trình p về tài nguyên r
int Allocation[NumProcs, NumResources];
    //Allocation[p,r] = số tài nguyên r đã cấp phát cho tiến trình p
int Need[NumProcs, NumResources];
    //Need[p,r] = Max[p,r] - Allocation[p,r]= số tài nguyên r mà
    tiến trình p còn cần sử dụng
int Finish[NumProcs] = false; //true nếu process P thực thi xong;
```

Tình trạng tắc nghẽn (deadlock)

❖ Giải thuật xác định trạng thái an toàn

B1.

If $\exists i$:

Finish[i] = false //tiến trình i chưa thực thi xong

Need[i,j] <= Available[j], $\forall j$ //Yêu cầu của process i được thỏa

Do B2 else goto B3

B2. Cấp phát mọi tài nguyên mà tiến trình i cần

Allocation[i,j]= Allocation[i,j]+Need[i,j]; $\forall j$

need[i,j]=0 ; $\forall j$

Available[j]= Available[j] - Need[i,j];

Finish[i] = true;

Available[j]= Available[j] + Allocation[i,j];

goto B1

B3. $\forall i$ If Finish[i] = true -> « hệ thống ở trạng thái an toàn », else
« không an toàn »

Tình trạng tắc nghẽn (deadlock)

- ❖ Giải thuật Banker: P_i yêu cầu k_r thể hiện của tài nguyên r

B1. If $k_r \leq \text{Need}[i, r] \forall r$, goto B2, else « lỗi »

B2. If $k_r \leq \text{Available}[r] \forall r$, goto B3;
else P_i « wait »

B3. $\forall r$:

$\text{Available}[r] = \text{Available}[r] - k_r$;
 $\text{Allocation}[i, r] = \text{Allocation}[i, r] + k_r$;
 $\text{Need}[i, r] = \text{Need}[i, r] - k_r$;

B4: Kiểm tra trạng thái an toàn của hệ thống.

Tình trạng tắc nghẽn (deadlock)

- Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

	Max			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	4	1	2
P2	6	1	3	2	1	1			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

- Nếu tiến trình **P2** yêu cầu **4 R1, 1 R3**. hãy cho biết yêu cầu này có thể đáp ứng mà không xảy ra **deadlock** hay không?

Tình trạng tắc nghẽn (deadlock)

- Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

B0: Tính Need là nhu cầu còn lại về mỗi tài nguyên j của mỗi tiến trình i:

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	4	1	2
P2	4	0	2	2	1	1			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Tình trạng tắc nghẽn (deadlock)

➤ Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

B1+B2: yêu cầu tài nguyên của P2 thoả đk ở B1, B2.

B3: Thử cấp phát cho P2, cập nhật tình trạng hệ thống

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	0	1	1
P2	0	0	1	6	1	2			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Tình trạng tắc nghẽn (deadlock)

➤ Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

B4: Kiểm tra trạng thái an toàn của hệ thống.

Lần lượt chọn tiến trình để thử cấp phát:

Chọn P2, thử cấp phát, g/s P2 thực thi xong thu hồi

$Available[j] = Available[j] + Allocation[i,j];$

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	6	2	3
P2	0	0	0	0	0	0			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Tình trạng tắc nghẽn (deadlock)

- Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

+ Chọn P1									
	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	0	0	0	0	0	7	2	3
P2	0	0	0	0	0	0			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Tình trạng tắc nghẽn (deadlock)

❖ Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

+ Chọn P3:									
	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	0	0	0	0	0	9	3	4
P2	0	0	0	0	0	0			
P3	0	0	0	0	0	0			
P4	4	2	0	0	0	2			

Tình trạng tắc nghẽn (deadlock)

❖ Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

+ Chọn P4:

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	0	0	0	0	0	9	3	6
P2	0	0	0	0	0	0			
P3	0	0	0	0	0	0			
P4	0	0	0	0	0	0			

- Mọi tiến trình đã được cấp phát tài nguyên với yêu cầu cao nhất, nên trạng thái của hệ thống là an toàn, do đó có thể cấp phát các tài nguyên theo yêu cầu của P2.



Tình trạng tắc nghẽn (deadlock)

❖ Giải thuật phát hiện tắc nghẽn

1. Đối với Tài nguyên chỉ có một thể hiện
2. Đối với Tài nguyên có nhiều thể hiện



Tình trạng tắc nghẽn (deadlock)

1. Đối với Tài nguyên chỉ có một thể hiện

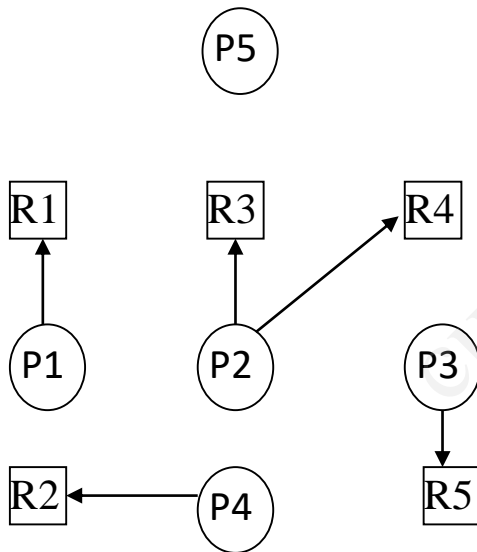
- Dùng đồ thị đợi tài nguyên (wait-for graph)
 - ✓ Được xây dựng từ đồ thị cấp phát tài nguyên (resource-allocation graph) bằng cách bỏ những đỉnh biểu diễn loại tài nguyên.
 - ✓ Cạnh từ P_i tới P_j : P_j đang đợi P_i giải phóng một tài nguyên mà P_j cần.
- Hệ thống bị tắc nghẽn nếu và chỉ nếu đồ thị đợi tài nguyên có chu trình.

Tình trạng tắc nghẽn (deadlock)

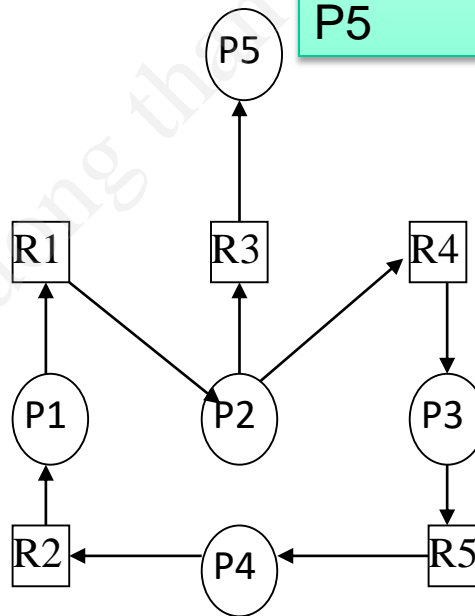
1. Đối với Tài nguyên chỉ có một thể hiện

➤ Ví dụ

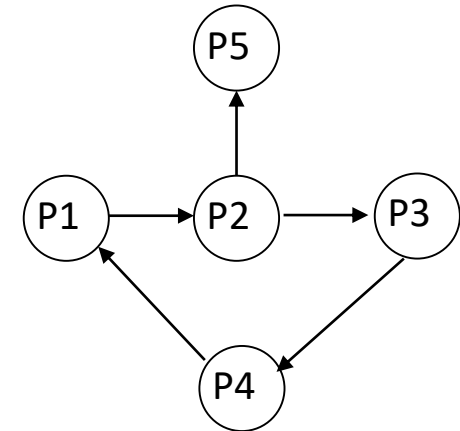
Tiến trình	Yêu cầu	Chiếm giữ
P1	R1	R2
P2	R3, R4	R1
P3	R5	R4
P4	R2	R5
P5		R3



đồ thị cấp phát tài nguyên



đồ thị thử cấp phát tài nguyên theo yêu cầu



đồ thị đợi tài nguyên



Tình trạng tắc nghẽn (deadlock)

2. Đối với Tài nguyên có nhiều thể hiện

Bước 1: Chọn Pi đầu tiên sao cho có yêu cầu tài nguyên **có thể** được đáp ứng,

nếu không có thì hệ thống bị tắc nghẽn.

Bước 2: **Thử cấp phát** tài nguyên cho Pi và kiểm tra trạng thái hệ thống,

nếu hệ thống an toàn thì tới Bước 3,

ngược lại thì quay lên Bước 1 tìm Pi kế tiếp.

Bước 3: Cấp phát tài nguyên cho Pi. Nếu tất cả Pi được đáp ứng thì hệ thống không bị tắc nghẽn, ngược lại quay lại Bước 1.

Tình trạng tắc nghẽn (deadlock)

❖ Hiệu chỉnh tắc nghẽn

- Hủy tiến trình trong tình trạng tắc nghẽn
 - ✓ Hủy đến khi không còn chu trình gây tắc nghẽn
 - ✓ Dựa vào các yếu tố như độ ưu tiên, thời gian đã xử lý, số tài nguyên đang chiếm giữ, hoặc yêu cầu thêm...
- Thu hồi tài nguyên
 - ✓ **Chọn lựa một nạn nhân**: tiến trình nào sẽ bị thu hồi tài nguyên? và thu hồi những tài nguyên nào?
 - ✓ **Trở lại trạng thái trước tắc nghẽn (rollback)**: khi thu hồi tài nguyên, cần phục hồi trạng thái của tiến trình về trạng thái gần nhất khi chưa bị tắc nghẽn.
 - ✓ **Tình trạng «đói tài nguyên»**: cần bảo đảm không có tiến trình nào luôn bị thu hồi tài nguyên?