

Ban học tập
Khoa Kỹ thuật máy tính

Training cuối học kỳ I năm học 2021-2022

HỆ ĐIỀU HÀNH



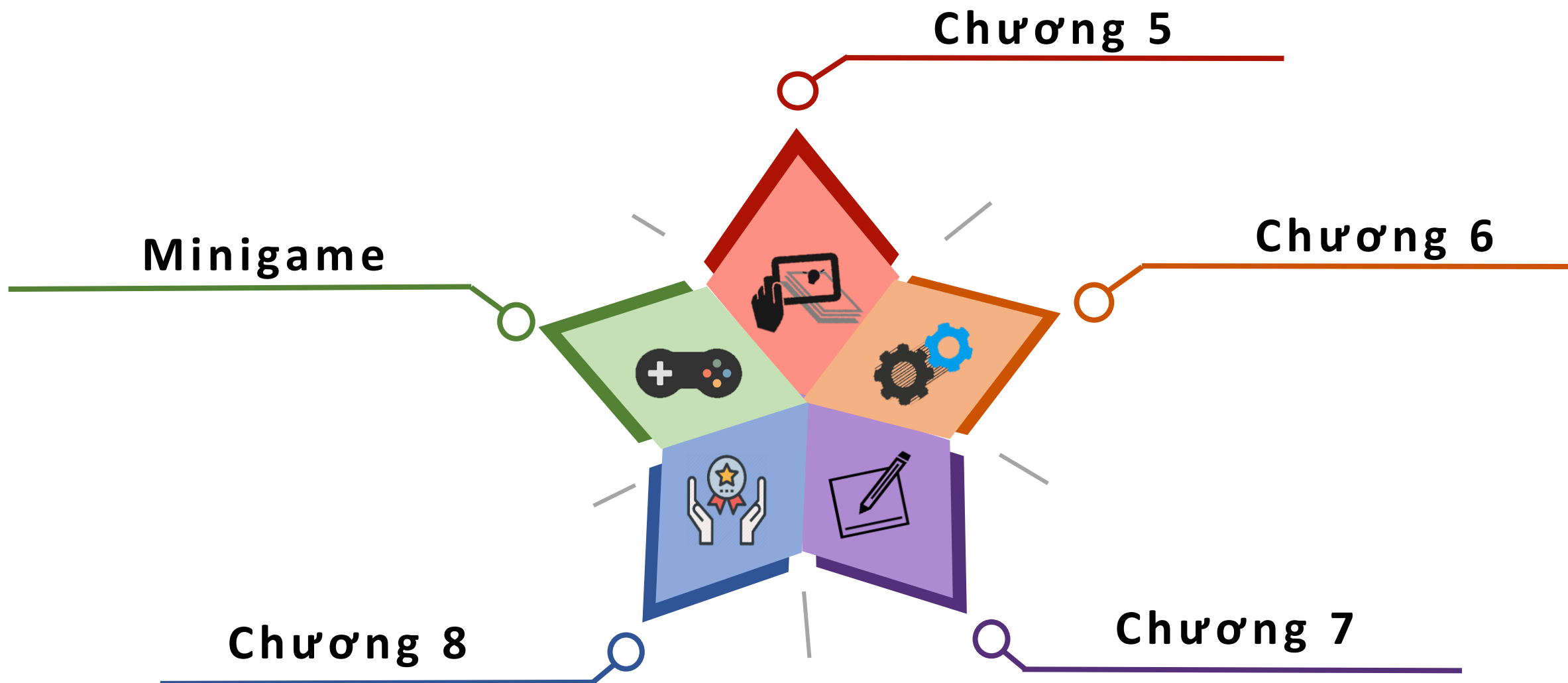
Email : bht.ktmt@gmail.com



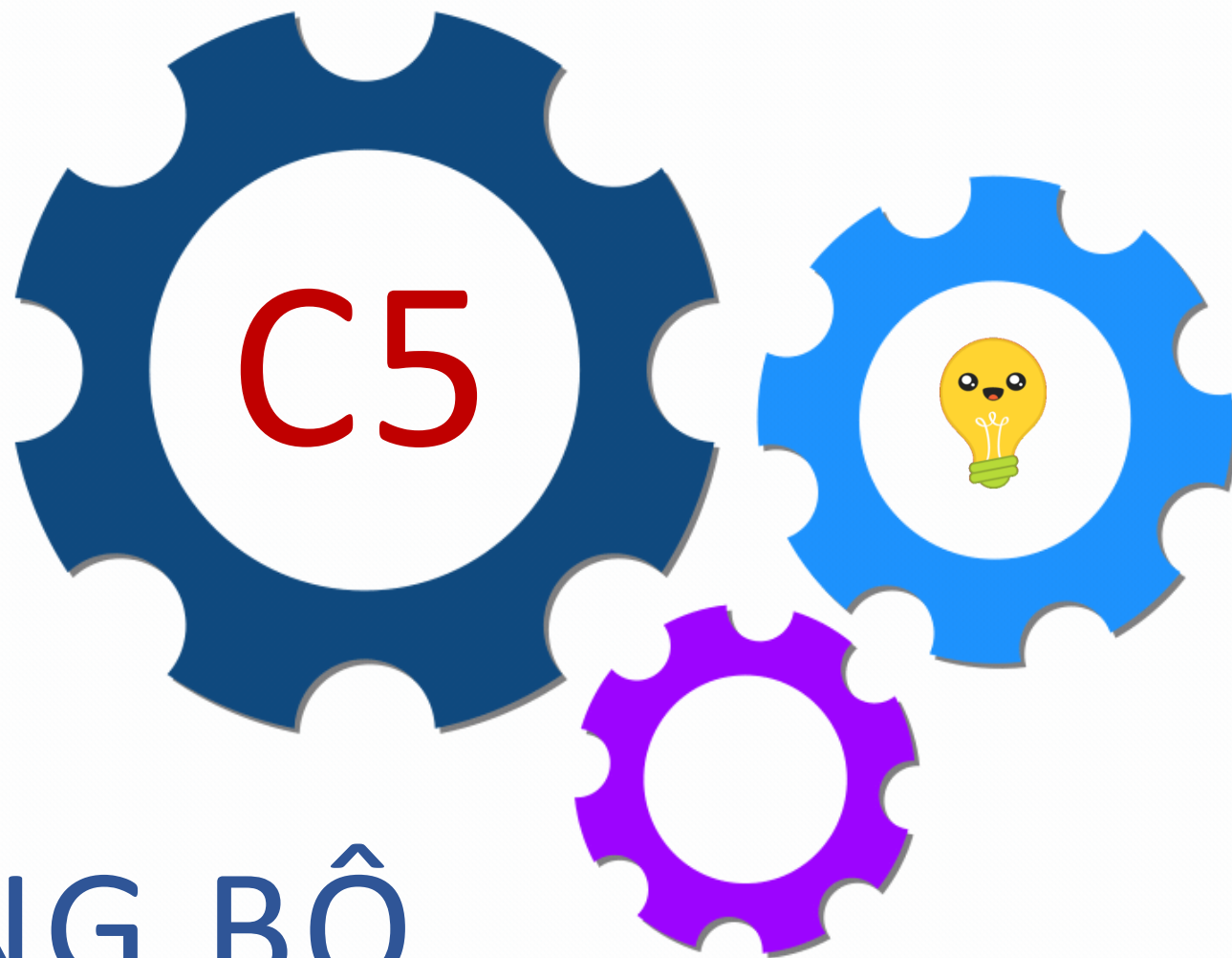
Fanpage : www.facebook.com/bht.ktmt



BAN HỌC TẬP
KHOA KỸ THUẬT MÁY TÍNH



MỤC LỤC



ĐỒNG BỘ

3 tính chất

Busy waiting

Sleep & wake-up

Phân loại giải pháp

Nhóm giải pháp **Sleep & Wakeup**

Semaphone

Monitor

Message

Nhóm giải pháp **Busy Waiting**

Sử dụng
các **biến cờ hiệu**

Sử dụng
việc **kiểm tra luân phiên**

Giải pháp của **Peterson**

Cấm ngắt

Chỉ thị TSL

Các giải pháp “Busy waiting”

1

Tiếp tục **tiêu thụ CPU** khi chờ vào vùng tranh chấp

Không đòi hỏi sự trợ giúp của hệ điều hành

2

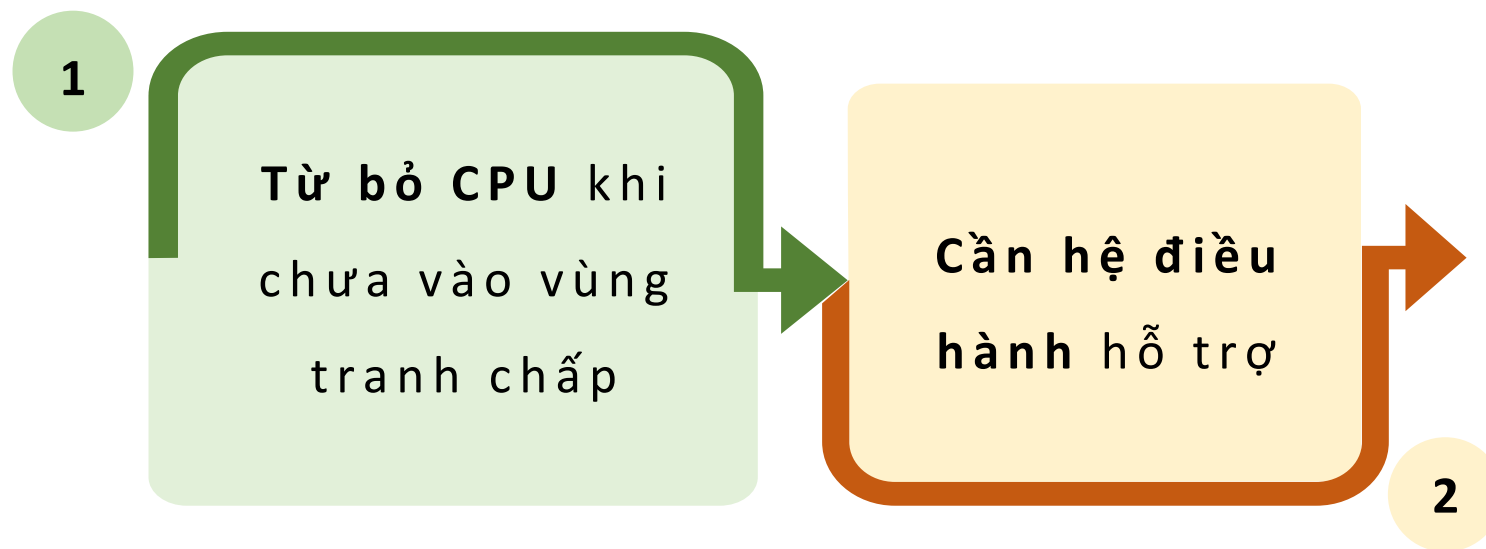
```
While (chưa có quyền) do nothing();
```

CS; —————> Khi đã có quyền thì vào CS

Từ bỏ quyền sử dụng CS

Sử dụng xong CS

Các giải pháp “Sleep & wake-up”



```
if (chưa có quyền) sleep();
```

CS; Khi đã có quyền thì vào CS

Wake-up (tiến trình khác)

Sử dụng xong CS



Busy waiting

Giải thuật **luân phiên**

Giải thuật **cờ hiệu**

Giải thuật **Peterson**

Giải thuật **Bakery**

Một số giải thuật khác

Bài tập



Đặc điểm

Biến chia sẻ: **turn** (khởi tạo = 0)
Nếu **turn = i** thì **Pi** được phép vào **CS**, với $i = 0$ hay 1

3 tính chất

Thỏa 1
Không thỏa 2 và 3

Minh họa

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

```
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);
```

turn = 1 -> P1 vào CS, P0 vào RS0.
RS0 quá lớn => P0 vẫn chưa vào vòng lặp mới, chưa vô được CS -> chưa gán turn tiếp theo = 1, trong khi RS1 nhanh hơn => P1 đang chờ ở vòng lặp tiếp theo cũng bị delay

Đặc điểm

Mảng boolean `flag[2]` (khởi tạo bằng `false`)
Nếu `flag[i] = true` thì P_i **sẵn sàng** vào CS

3 tính chất

Thỏa mãn 1

Không thỏa mãn 2

Minh họa

P_i đã sẵn sàng

Nếu P_j đang dùng thì chờ P_j dùng xong

Thực hiện xong thì gán `false`

Không thỏa mãn 2 vì chẳng hạn có 3 process,
 i_0 đang thực hiện nhưng lâu, i_1 chờ, i_2 vừa
thực hiện xong cũng chờ

=> nhiều flag cùng true thì bị delay mãi mãi

```
do {  
    flag[i] = true;  
    while ( flag[j] );  
        critical section  
    flag[i] = false;  
        remainder section  
} while (1);
```



Đặc điểm

Kết hợp 2 giải thuật trước: cả turn và flag

3 tính chất

Thỏa mãn all

Minh họa

P0 sẵn sàng, gán turn = 1 để nhường P1 nếu P1 sẵn sàng. P1 dùng xong vùng CS thì P0 vào CS. Ngược lại đối với P1

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

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

turn trong 1 thời điểm chỉ mang 1 giá trị => 1 thời điểm chỉ có 1 Process vào CS
P1 true cũng không ngăn được P0 vì muốn tiến vào CS còn cần phụ thuộc vào turn
Không xảy ra chờ đợi vô hạn vì có thể ngắt while bằng flag hoặc turn



Đặc điểm

Mỗi process nhận một con số

Số nhỏ nhất sẽ vào CS

Nếu cùng số thì so sánh số thứ tự, $i < j$ thì i vào trước

Số num được cấp tăng dần

Minh họa

```
boolean    choosing[ n ];
int        num[ n ];
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);
```



Giải thuật Bakery (n process)

nếu Process vào
sau cùng thì sẽ
mang con số lớn
nhất trong số
các Process + 1

```
boolean    choosing[ n ];
int         num[ n ];
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));
    }
    num[ i ] = 0;
} while (1);
```

critical section

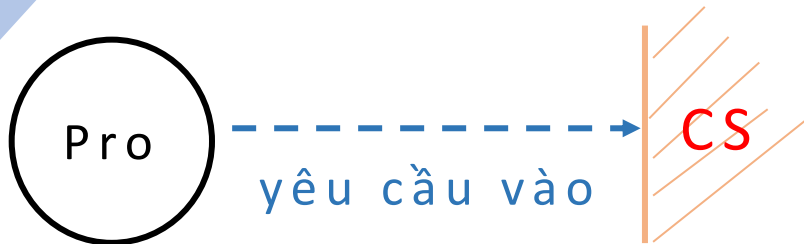
remainder section

nếu bắt gặp process j
đang thực hiện gán num
thì chờ để nó gán xong
mới so sánh

chờ nếu có pro có num nhỏ hơn nó -> số nhỏ hơn vào CS. nếu
process đã thực hiện xong (num = 0) hoặc process j được gán
1 num mới lớn hơn
=> break while, so sánh process khác



Software



Phải liên tục kiểm tra điều kiện

Tốn nhiều thời gian xử lý của CPU

Nếu thời gian xử lý trong vùng CS lớn
=> cần có cơ chế block process

Giải quyết tranh chấp

Hardware

Cấm ngắt
(disable interrupts)

Dùng các lệnh đặc biệt

Uniprocessor:
tính chất 1 **đảm bảo**

Multiprocessor:
tính chất 1 **không đảm bảo**

system clock không thể thực hiện

Chỉ cấm ngắt tại CPU thực thi lệnh
`disable_interrupts`

```
Process Pi:
do{
  disable_interrupts();
  critical section
  enable _interrupts();
  remainder section
} while (1);
```

Lệnh **TestAndSet**

- Đảm bảo tính chất 1
 - Quá trình chọn lựa process P_j vào CS kế tiếp là tùy ý
- ⇒ không bảo đảm điều kiện tính chất 3 => xảy ra **starvation**

Bị bỏ đói

Swap

- Biến chia sẻ **lock** (khởi tạo = false)
- Biến cục bộ **key**
- P_i nào thấy giá trị **lock = false** trước thì vào CS trước
- P_i vào CS sẽ cho **lock = true** để ngăn các P khác

Giải thuật **TestAndSet**

- Thỏa mãn 3 tính chất

Code	Giải thích
<pre> while (TRUE) { int j = 1-i; flag[i]= TRUE; turn = i; while (turn == j && flag[j]==TRUE); critical-section (); flag[i] = FALSE; Noncritical-section (); } </pre>	<p>Cho $j = 1 - i$, là số ngược lại của i (0 hoặc 1) Cờ $i = \text{true}$ Biến turn qui định i được vào Nếu cùng lúc có tiến trình j được gán turn và cờ j true thì chờ cho tới khi j dùng xong CS, hoặc nếu j đang dùng CS mà bị gán $\text{turn} = i$ thì i cũng vào CS Gán cờ = false sau khi dùng xong CS Thực hiện các tác vụ khác</p>

(1) Biến turn tại 1 thời điểm chỉ có thể $= i$ hoặc j , $\text{turn} = i \Rightarrow$ cho i vào CS ngay cả khi j đang ở CS \Rightarrow không thỏa mutual. Tuy nhiên, nếu P_j có thời gian thực hiện trong CS nhanh hơn từ lúc P_j vừa vào đến lúc $\text{turn} = i$ thì vẫn có khả năng chỉ có 1 tiến trình được vào CS \Rightarrow vẫn có thể thỏa

(2) P_j nếu không được vào vùng CS cũng không thể ngăn tiến trình i vào CS vì qui định tiến trình được vào $= \text{turn} \Rightarrow$ progress

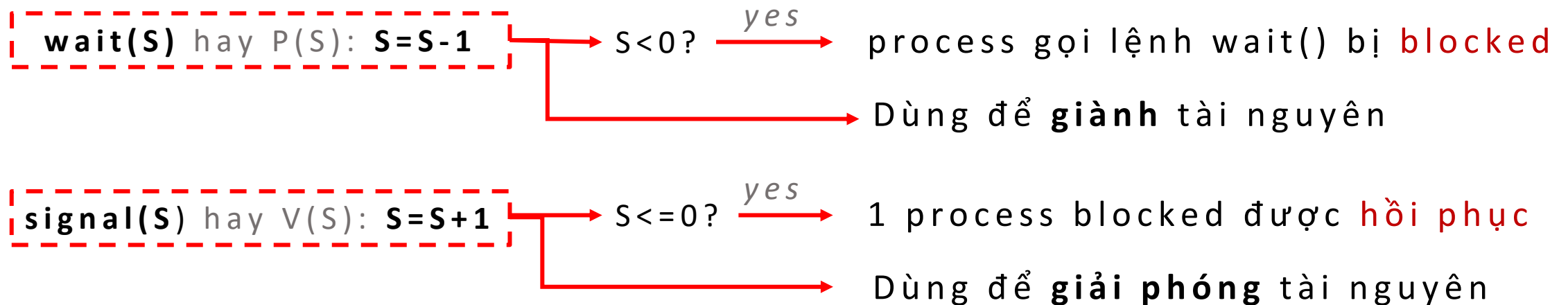
(3) Nếu một tiến trình bị kẹt trong vòng lặp while thì sẽ bị ngắt khỏi CS nếu tiến trình khác được gán $\text{turn} \Rightarrow$ bounded waiting

\Rightarrow Có thể thỏa hoặc không thỏa 3 yêu cầu

Sleep & wake-up

Semaphore

- Là công cụ đồng bộ cung cấp bởi OS, không đòi hỏi busy waiting
- Semaphore S là một biến số nguyên
- S có thể được truy xuất qua hai tác vụ atomic và mutual exclusive



- Các process bị blocked vì chờ cùng 1 sự kiện sẽ được đặt trong cùng **blocked queue** → Danh sách liên kết các PCB
- 2 loại
 - Counting semaphore → S là số nguyên
 - Binary semaphore → S = 0 hoặc 1



Sleep & wake-up

Semaphore

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

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

Dùng cơ chế FIFO khi lấy ra từ hàng đợi



Sleep & wake-up

Semaphore

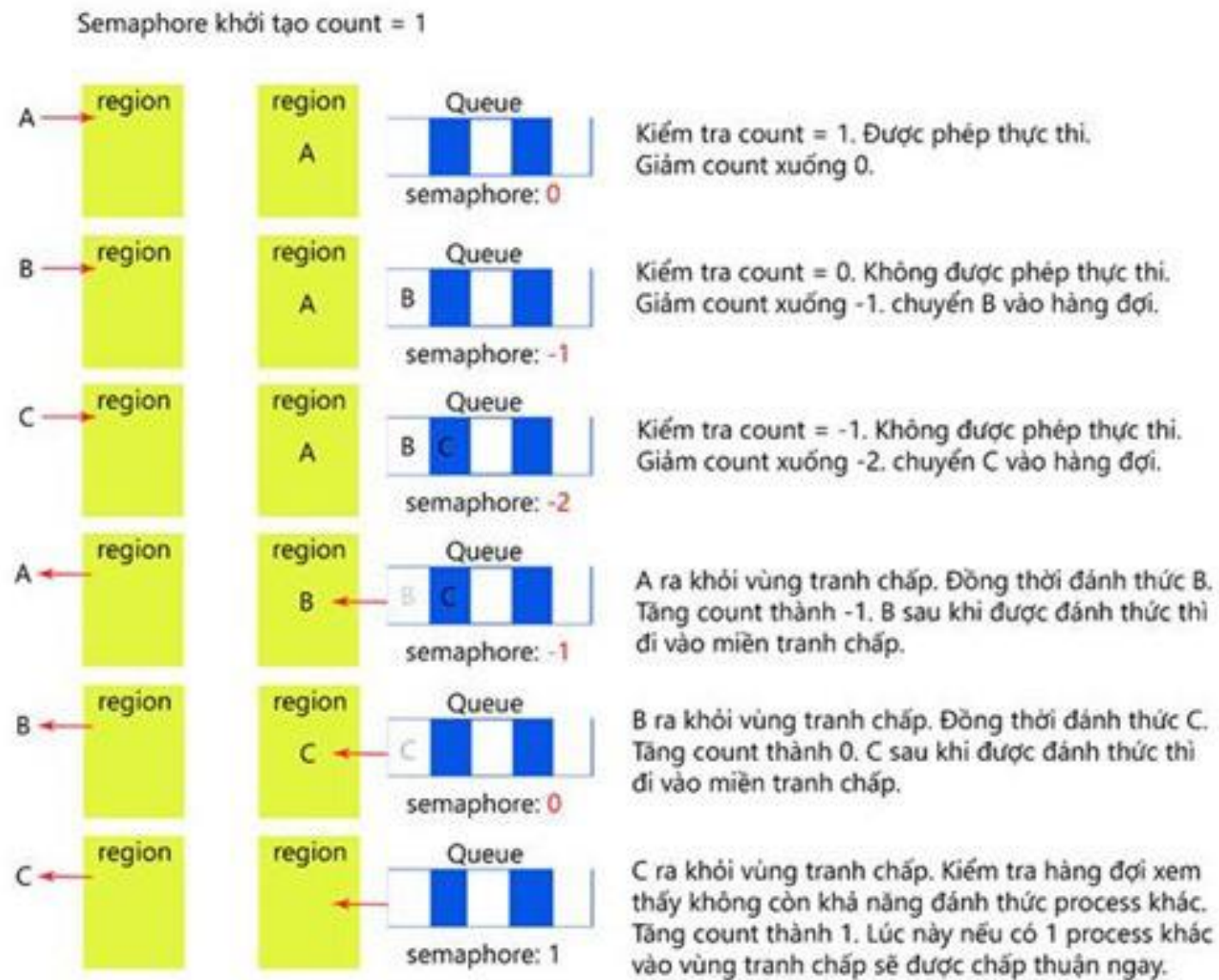


Figure: Counting Semaphore (non priority)



Sleep & wake-up

Semaphore

- Semaphore bảo đảm mutual exclusion và phối hợp đồng bộ các process
- 1 process bị “**die**” có thể khiến các process khác cùng sử dụng biến semaphore

Cấu trúc:

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

Hàm được sử dụng:

- **block()**: tạm treo process nào thực thi lệnh này, trạng thái process từ **running** -> **waiting**
- **wakeup(P)**: hồi phục process P đang blocked, trạng thái process từ **waiting** -> **ready**



Sleep & wake-up

Semaphore

VD1

- Dùng cho n process
- Khởi tạo $S.value = 1 \Rightarrow$ chỉ có 1 process được vào CS
- Để cho phép k process vào CS, khởi tạo $S.value = k$

```
semaphore mutex; //khởi tạo mutex.value=1
```

Process P_i :

```
do {
```

```
    wait(mutex);
```

```
    critical section
```

```
    signal(mutex);
```

```
    remainder section
```

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

Process i tiến vào \rightarrow thấy $value < 0 \Rightarrow$ block() đứng yên một chỗ

Khi 1 process trong CS thực hiện xong, ra khỏi CS \Rightarrow gọi signal()

\Rightarrow 1 process đang bị block sẽ tiếp tục chạy



Sleep & wake-up

Semaphore

VD2

- Dùng cho 2 process: P1 và P2
- Yêu cầu: lệnh S1 trong P1 cần được thực thi trước lệnh S2 trong P2
- Dùng semaphore synch để đồng bộ
- Khởi động semaphore:

`synch.value = 0`

Để đồng bộ hoạt động theo yêu cầu, P1 phải định nghĩa như sau:

```
S1;  
signal(synch);
```

P2 định nghĩa như sau:

```
wait(synch);  
S2;
```



Sleep & wake-up

Semaphore

VD3

- Xét 2 process xử lý 2 đoạn chương trình:

Tiến trình P1 {A1, A2}

Tiến trình P2 {B1, B2}

- Đồng bộ hóa hoạt động của 2 tiến trình sao cho cả A1 và B1 đều hoàn tất trước khi A2 và B2 bắt đầu.
- Khởi tạo semaphore $s1.v = s2.v = 0$

Để đồng bộ hoạt động theo yêu cầu, P1 phải định nghĩa như sau:

```
A1;  
signal(s1);  
wait(s2);  
A2;
```

P2 định nghĩa như sau:

```
B1;  
signal(s2);  
wait(s1);  
B2;
```

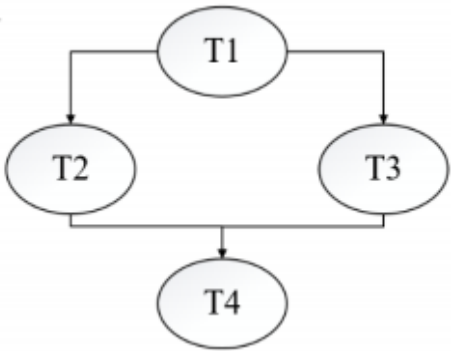


Hệ thống có 4 tiểu trình T1, T2, T3, T4. Quan hệ giữa các tiểu trình này được biểu diễn như sơ đồ, mũi tên từ Tx sang Ty nghĩa là Tx phải kết thúc trước khi Ty bắt đầu thực thi. Giả sử tất cả các tiểu trình đã được khởi tạo và sẵn sàng để thực thi. Dùng semaphore để đồng bộ hoạt động của các tiểu trình cho đúng với sơ đồ.

Điều kiện:

- T1 kết thúc -> T2 thực thi
- T1 kết thúc -> T3 thực thi
- T2 và T3 kết thúc -> T4 thực thi

3 điều kiện => 3 semaphore
s1 = 0;
s2 = 0;
s3 = 0;



Các lệnh wait() và signal() của các semaphore

```
Void T1(void)
{ CS;
//T1 kết thúc, thực thi T2 &T3
    signal(s1);
    signal(s1);
}
```

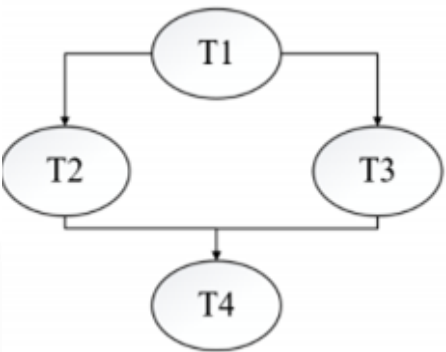
```
Void T2(void)
{ wait(s1); //đợi T1
    CS;
    signal(s2); // T2
    kết thúc, thực thi T4
}
```

```
Void T3(void)
{
    Wait(s1);
    CS;
    Signal(s3);
}
```

```
Void T4(void)
{
    Wait(s2);
    Wait(s3);
    CS ;
}
```



Hệ thống có 4 tiểu trình T1, T2, T3, T4. Quan hệ giữa các tiểu trình này được biểu diễn như sơ đồ, mũi tên từ Tx sang Ty nghĩa là Tx phải kết thúc trước khi Ty bắt đầu thực thi. Giả sử tất cả các tiểu trình đã được khởi tạo và sẵn sàng để thực thi. Dùng semaphore để đồng bộ hoạt động của các tiểu trình cho đúng với sơ đồ.



Giải pháp hơn **tối ưu** chỉ dùng **2 semaphore**

=> khởi tạo sem1 = sem2 = 0;

```
void T1(void)
{

//T1 thực thi

signal(sem1)
signal(sem1)
}
```

```
void T2(void)
{

wait(sem1)

//T2 thực thi

signal(sem2)

}
```

```
void T3(void)
{

wait(sem1)

//T3 thực thi

signal(sem2)

}
```

```
void T4(void)
{

wait(sem2)
wait(sem2)
//T4 thực thi

}
```




Sleep & wake-up

Semaphore

Nhận xét

- $S.value \geq 0$: số process có thể thực thi = $S.value$
 - $S.value < 0$: 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 $signal(S)$ tại một thời điểm
- ⇒ đoạn code của lệnh $wait(S)$ và $signal(S)$ cũng là **vùng tranh chấp**

- Vùng tranh chấp thường rất nhỏ, chỉ khoảng 10 lệnh
- Giải pháp:
 - Uniprocessor: dùng cấm ngắt
 - Multiprocessor: giải pháp software (giải thuật Dekker, Peterson,...) hoặc giải pháp hardware (TestAndSet, Swap,...)
- Vì CS rất nhỏ nên chi phí cho busy waiting sẽ rất thấp



Sleep & wake-up

Semaphore

Deadlock chờ đợi vô hạn định một sự kiện không bao giờ xảy ra

Starvation tiến trình có thể không bao giờ được lấy ra khỏi hàng đợi

S và Q là hai biến semaphore được khởi tạo = 1

P0

wait(S);

wait(Q);

signal(S);

signal(Q);

P1

wait(Q);

wait(S);

signal(Q);

signal(S);

P0 gọi wait(S), rồi P1 gọi wait(Q) => $S.value = 1 - 1 = 0$, $Q.value = 1 - 1 = 0$

Tiếp tục cho P0 gọi wait(Q) => bị blocked, P1 gọi wait(S) => bị blocked

=> không thể gọi được hàm signal nào



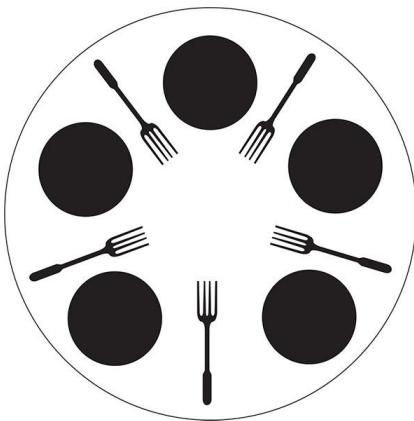
Sleep & wake-up

3 bài toán kinh điển



Bounded Buffer
Problem

Nhà máy sản xuất đưa
hàng vào kho và người
mua lấy hàng từ kho



Dining-Philosophers
Problem

Chia đĩa cho các
triết gia



Readers and Writers
Problem

Writer và Reader không
được truy xuất
database cùng lúc



Bounded Buffer Problem

3 bài toán kinh điển

Tạo 1 item

Nhà máy

Khách hàng

giảm value của full, nếu value full < 0 thì bị block (người không được mua hàng)

Nếu mutex < 0 thì nhà máy đang can thiệp kho, chờ đến khi nhà máy xong thì mới được mua hàng

Tăng mutex để cho người khác vào

Tăng empty, nếu empty <= 0 thì đánh thức 1 item khác (cho phép thêm hàng vào kho)

Xem lại wait&signal slide 17

kiểm tra số buffer empty, nếu < 0 thì bị item đó bị block (không còn chỗ trống để đưa hàng vào), nếu số buffer còn trống > 0 thì gọi wait(mutex) để kiểm tra có ai đang can thiệp kho không

nếu mutex.value < 0 thì có người can thiệp kho, chờ đến khi người đó làm xong và gọi signal(mutex) thì nhà máy sẽ được wake-up và vào CS. Thực hiện xong thì nhà máy gọi signal(mutex) để tăng mutex value cho người khác vào.

gọi signal(full) -> tăng giá trị của full, nếu full <= 0 -> wake-up cho 1 item khác vào kho

```
do {
    nextp = new_item();
    ...
    wait(empty);
    wait(mutex);
    ...
    insert_to_buffer(nextp);
    ...
    signal(mutex);
    signal(full);
} while (1);
```

```
do {
    wait(full);
    wait(mutex);
    ...
    nextc = get_buffer_item(out);
    ...
    signal(mutex);
    signal(empty);
    ...
    consume_item(nextc);
} while (1);
```

Semaphore full, empty, mutex;
Khởi tạo:
full = 0; //số buffers đầy
empty = n; // số buffers trống
mutex = 1; //1 thời điểm chỉ có 1 bên được can thiệp kho



Triết gia thứ i :

```
do {
```

```
    wait(chopstick [i])
```

```
    wait(chopstick [(i + 1) % 5])
```

```
    ...
```

```
    eat
```

```
    ...
```

```
    signal(chopstick [i]);
```

```
    signal(chopstick [(i + 1) % 5]);
```

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

5 triết gia ngồi ăn

Mỗi người cần 2 chiếc đũa để ăn

Trên bàn chỉ có 5 đũa

- Khởi tạo value của từng đũa là 1

- Triết gia thứ i ăn được khi cây đũa thứ i và thứ $i+1$ chưa được ai dùng

Nếu cây đũa thứ i có value < 0 thì triết gia bị block

Kiểm tra thêm cây đũa thứ $i+1$

Nếu cả i và $i+1$ đều chưa có người dùng thì cho triết gia dùng cả đôi để ăn

Ăn xong thì tăng value i và $i+1$ lên 1, báo hiệu đã sẵn sàng cho người khác sử dụng.



Triết gia thứ i :

```
do {  
    wait(chopstick [i])  
    wait(chopstick [(i + 1) % 5])  
    ...  
    eat  
    ...  
    signal(chopstick [i]);  
    signal(chopstick [(i + 1) % 5]);  
} while (1);
```

Deadlock

khi tất cả đồng thời cầm lên chiếc đũa bên tay trái \Rightarrow mỗi người giữ 1 cây \Rightarrow không ai ăn được để signal

Giải quyết

Cách 1: Tối đa 4 người ngồi \Rightarrow Mỗi người cầm 1 chiếc \Rightarrow vẫn dư 1 chiếc để 1 người ăn xong trước

Cách 2: Triết gia ngồi ở vị trí lẻ cầm đũa bên trái trước, rồi mới đến đũa bên phải, triết gia ở vị trí chẵn cầm đũa bên phải trước, rồi mới đến đũa bên trái \Rightarrow triết gia lẻ cầm chiếc đũa của triết gia chẵn thì triết gia chẵn phải đợi



Reader Writer Problem

3 bài toán kinh điển

Writer không được cập nhật dữ liệu khi có một Reader đang truy xuất CSDL
1 thời điểm -> chỉ cho phép 1 Writer sửa đổi CSDL

Khai báo:

```
semaphore mutex = 1;  
semaphore wrt = 1;  
int readcount = 0;
```

Writer

```
wait(wrt);  
...  
writing is performed  
...  
signal(wrt);
```

Reader process

```
wait(mutex);  
readcount++;  
if (readcount == 1)  
    wait(wrt);  
signal(mutex);  
...  
reading is performed  
...  
wait(mutex);  
readcount--;  
if (readcount == 0)  
    signal(wrt);  
signal(mutex);
```

Block nếu có người đang truy xuất CSDL

Bị block nếu có writer đang viết

Tăng mutex để không cho người khác Read

Giảm mutex. Mutex = 0 thì người khác trong hàng đợi được phép read

Ngừng đọc thì cộng wrt lên 1, cho phép writer



Sleep & wake-up

Monitor

Khái niệm

Là một cấu trúc ngôn ngữ cấp cao, có chức năng như semaphore nhưng dễ điều khiển hơn

Có thể hiện thực bằng semaphore

Gồm

- Một hoặc nhiều thủ tục (procedure)
- Một đoạn code khởi tạo (initialization code)
- Các biến dữ liệu cục bộ (local data variable)

Đặc tính

- Dùng các thủ tục của monitor để truy xuất biến local
- Process “vào monitor” bằng cách gọi một trong các thủ tục đó
- Chỉ có một process có thể vào monitor tại một thời điểm

Chương 6

DEADLOCKS





- Deadlock: đờn một sự kiện không bao giờ xảy ra
- Có nhiều hơn một tiến trình bị liên quan
- Tiến trình trì hoãn vô hạn định nếu nó bị trì hoãn một khoảng thời gian dài lặp đi lặp lại nhưng hệ thống lại đáp ứng các tiến trình khác

Khái niệm

Ví dụ 1:

Hệ thống có 2 file trên đĩa

P1 và P2 mỗi tiến trình mở một file và yêu cầu mở file kia

Ví dụ 2:

Bài toán các triết gia ăn tối


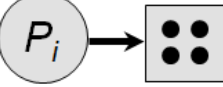
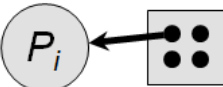
Mỗi người cầm 1 chiếc đũa và chờ chiếc còn lại

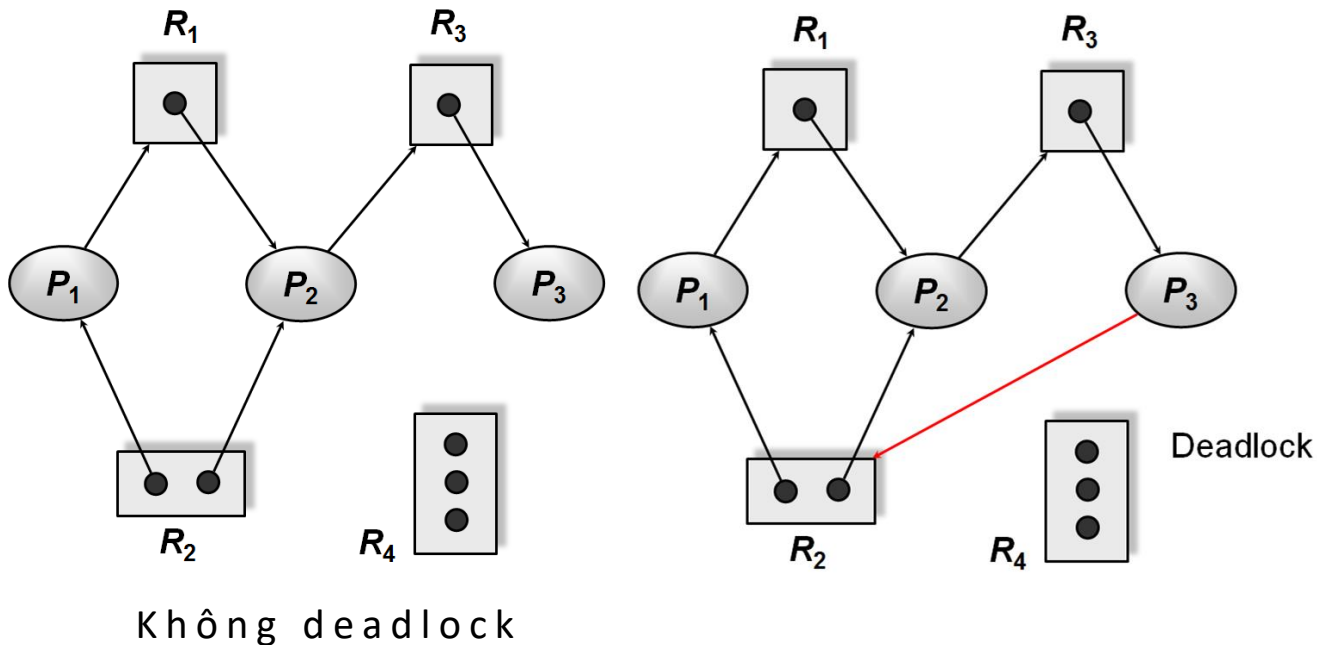
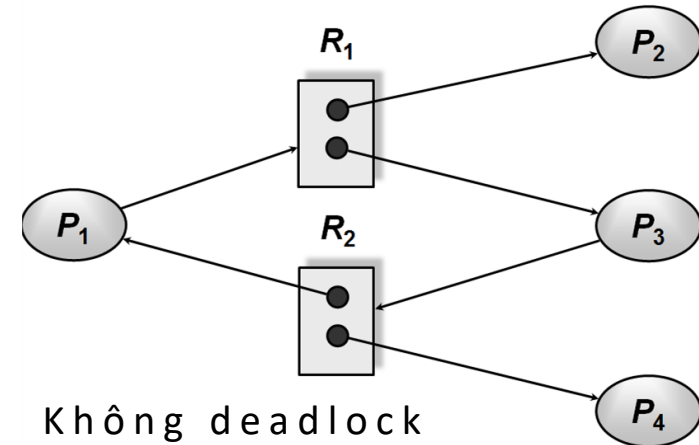


Điều kiện xảy ra deadlock

- 1 Ít nhất một tài nguyên được giữ theo **nonsharable mode**
- 2 Giữ và chờ cấp thêm tài nguyên
- 3 Không trưng dụng: tài nguyên không thể bị lấy lại mà chỉ có thể được trả lại từ tiến trình đang giữ tài nguyên đó khi nó muốn
- 4 Chu trình đợi: tồn tại vòng tròn các quá trình đang đợi sao cho
 - P0 đợi một tài nguyên mà P1 giữ
 - P1 đợi một tài nguyên mà P2 giữ
 - ...
 - Pn đợi một tài nguyên mà P0 giữ

Đồ thị cấp phát tài nguyên (RAG)

- Process i P_i
- Loại tài nguyên R_j với 4 thực thể 
- P_i yêu cầu một thực thể của R_j 
- P_i đang giữ một thực thể của R_j 



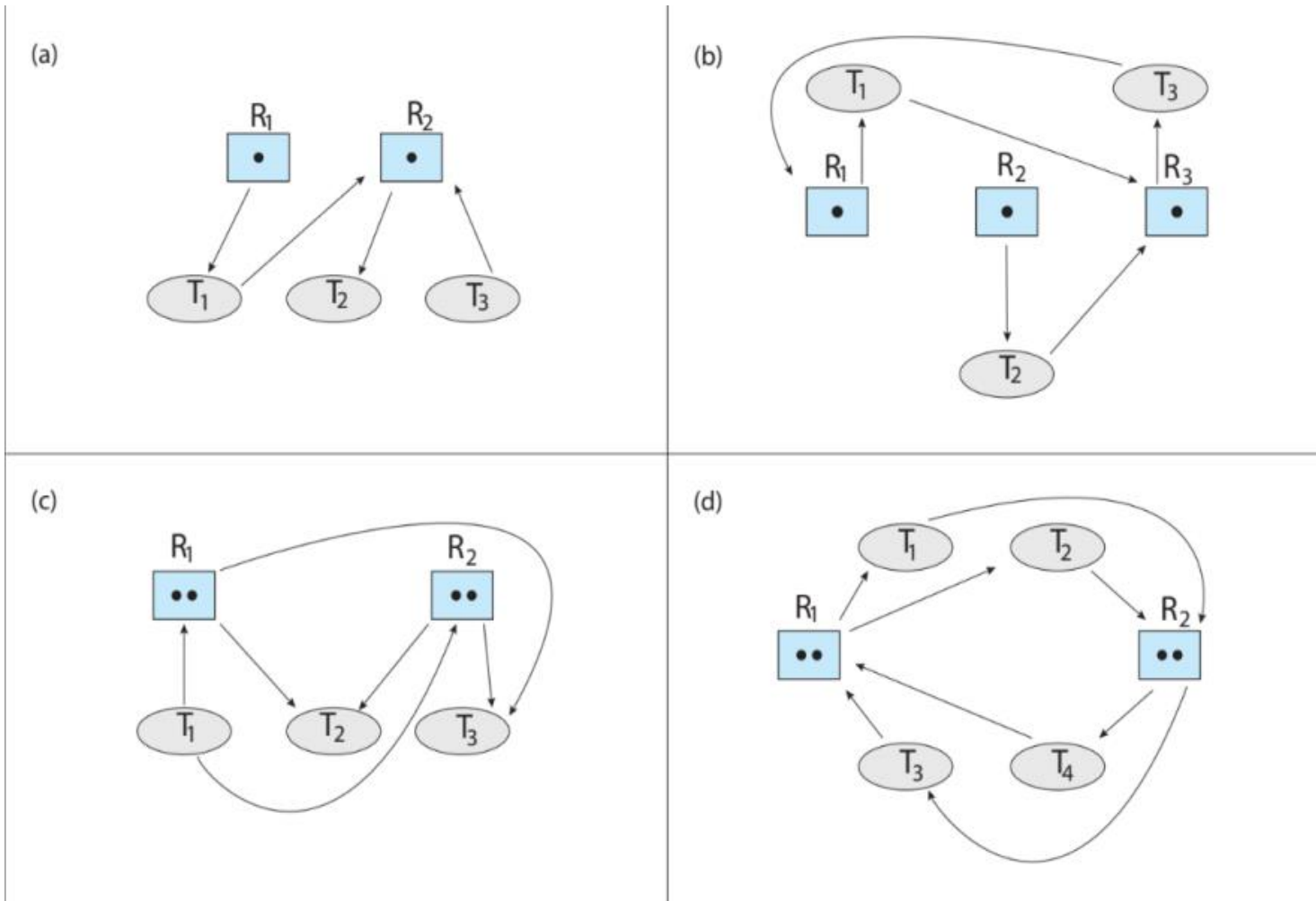
- Không chu trình -> không deadlock
- Chứa một (hay nhiều) chu trình:
 - ✓ Mỗi loại tài nguyên chỉ có **1 thực thể** -> **deadlock**
 - ✓ Mỗi loại tài nguyên có **nhiều thực thể** -> **có thể deadlock**



Đồ thị cấp phát tài nguyên (RAG)

Đồ thị nào có deadlock?

Bài tập nhận biết deadlock



A. Đồ thị (a), (b)

B. Đồ thị (c), (d)

C. Đồ thị (b), (d)

D. Đồ thị (b), (c), (d)



Phương pháp giải quyết deadlocks

- Ngăn deadlock
- Tránh deadlock
- Cho phép hệ thống vào trạng thái deadlock, nhưng sau đó phát hiện deadlock và phục hồi hệ thống
- ‘Bơ’ deadlock



- ✓ Khá nhiều hệ điều hành sử dụng phương pháp này
- ✓ Deadlock không được phát hiện => giảm hiệu suất của hệ thống
=> hệ thống có thể ngưng hoạt động



Phương pháp giải quyết deadlocks

Ngăn deadlock

Không cho phép (ít nhất) 1 trong 4 điều kiện xảy ra deadlock

- (1) Ngăn mutual exclusion
 - Với tài nguyên không chia sẻ: không thể
 - Với tài nguyên chia sẻ: không cần thiết
- (2) Giữ và chờ tài nguyên
 - Tiến trình yêu cầu toàn bộ tài nguyên cần thiết một lần. đủ tài nguyên => cấp phát, không đủ => tiến trình bị block
 - Tiến trình phải trả lại tài nguyên trước khi yêu cầu cấp phát
- (3) Ngăn không trưng dụng *A có tài nguyên, A yêu cầu tài nguyên khác nhưng chưa được cấp*
 - Cách 1: Hệ thống lấy lại mọi tài nguyên A đang giữ
 - Cách 2: Hệ thống sẽ xem tài nguyên A yêu cầu
 - được giữ bởi một tiến trình khác đang đợi thêm tài nguyên => hệ thống lấy lại và cấp cho A
 - được giữ bởi tiến trình không đợi tài nguyên, A phải đợi => tài nguyên của A bị lấy
- (4) Ngăn chu trình đợi *gán số thứ tự* cho tất cả các tài nguyên => tiến trình chỉ có thể yêu cầu thực thể của một loại tài nguyên theo thứ tự tăng dần
Tiến trình yêu cầu một thực thể của loại tài nguyên R_j thì phải *trả lại các tài nguyên R_i có số thứ tự lớn hơn R_j*



Tránh deadlock

cung cấp thông tin về tài nguyên để hệ thống cấp phát tài nguyên hợp lý

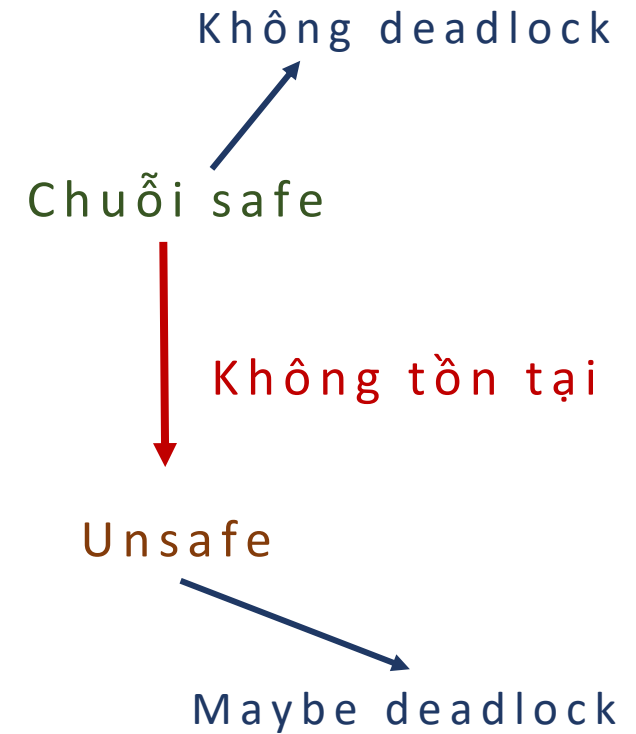
- Ngăn deadlock sử dụng tài nguyên không hiệu quả
- Tránh deadlock vẫn đảm bảo hiệu suất sử dụng tài nguyên tối đa
- Yêu cầu mỗi tiến trình khai báo số lượng tài nguyên tối đa cần để thực hiện công việc
- Giải thuật tránh deadlock kiểm tra trạng thái cấp phát tài nguyên
- Trạng thái cấp phát tài nguyên được định nghĩa dựa trên số tài nguyên còn lại, số tài nguyên đã được cấp phát, yêu cầu các tiến trình

Chuỗi an toàn

chuỗi $\langle P_1, P_2, \dots, P_n \rangle$

yêu cầu tài nguyên của P_i có thể được thỏa bởi

- Tài nguyên hệ thống đang có sẵn
- Tài nguyên mà tất cả các P_j ($j < i$) đang giữ



Ví dụ Hệ thống có 12 tape drive và 3 tiến trình P_0, P_1, P_2
Tại thời điểm t_0

	Cần tối đa	Đang giữ	Cần thêm
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Còn 3 tape drive sẵn sàng

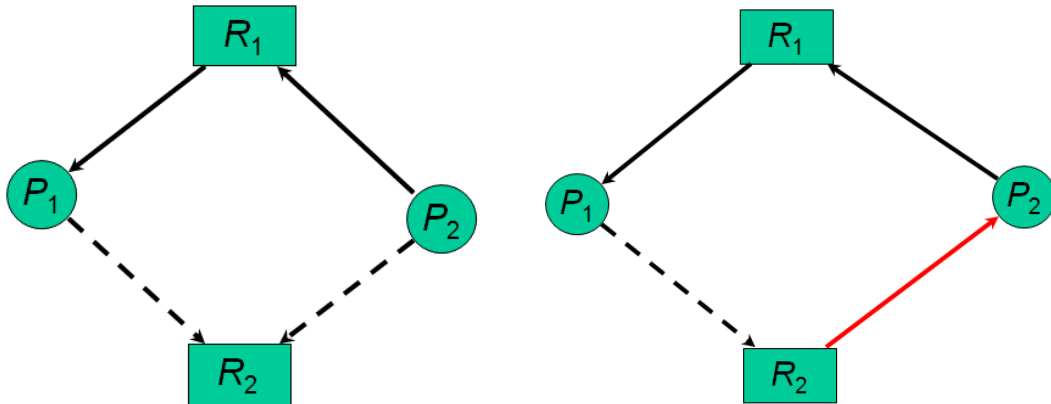
Chuỗi $\langle P_1, P_0, P_2 \rangle$ là chuỗi an toàn \rightarrow hệ thống an toàn

Các giải thuật tránh deadlocks

Giải thuật đồ thị cấp phát tài nguyên

Mỗi tài nguyên có 1 thực thể

Không có điều kiện



Giải thuật Banker

Mỗi tài nguyên có nhiều thực thể

tiến trình phải khai báo số thực thể tối đa của mỗi loại tài nguyên nó cần

tiến trình đã có đủ tài nguyên thì phải hoàn trả trong một khoảng thời gian

Sơ đồ cấp phát

	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

Các giải thuật tránh deadlocks

Các bước thực hiện giải thuật Banker

Tìm $Need = Max - Allocation$

Tìm tiến trình P_i thỏa:

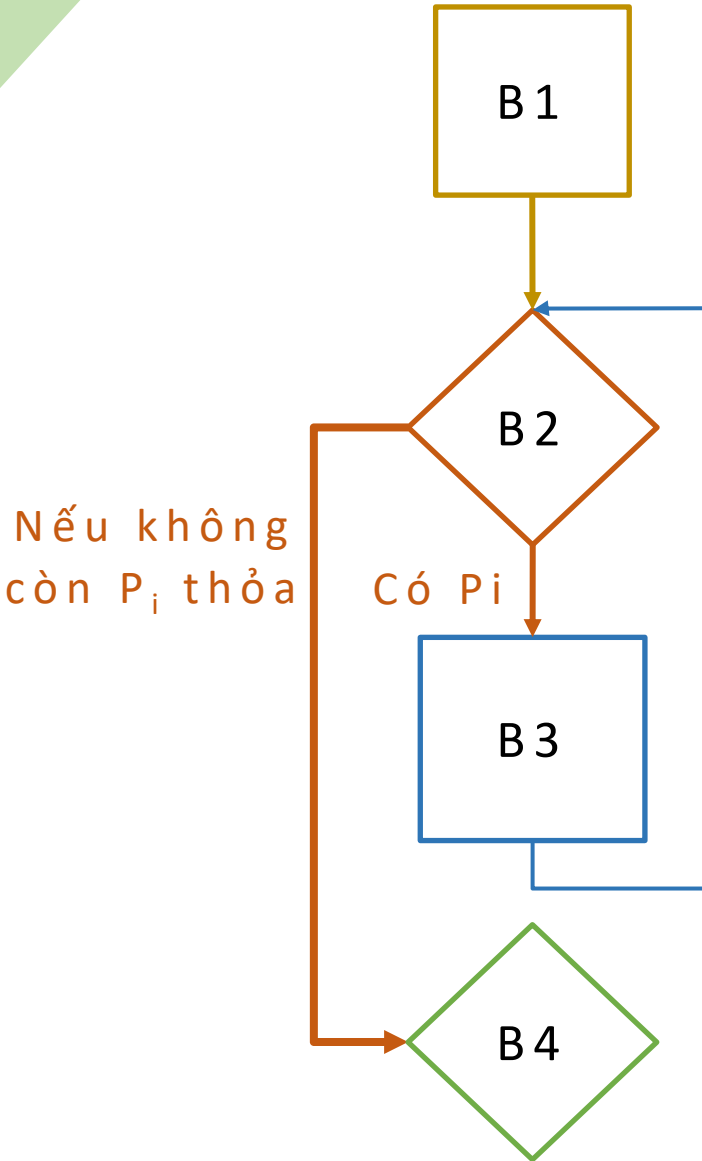
- + P_i chưa hoàn thành thực thi
- + $Need_i \leq Available$

$Available = Available + Allocation_i$

Thêm P_i vào chuỗi

Nếu chuỗi có tồn tại đủ hết các P

=> hệ thống tồn tại chuỗi an toàn & ngược lại





Sử dụng giải thuật Banker, cho biết hệ thống có an toàn hay không?

Tiến trình	Allocation				Max			
	R1	R2	R3	R4	R1	R2	R3	R4
P1	3	1	1	2	5	3	4	3
P2	1	1	2	1	3	4	6	1
P3	2	1	4	5	3	5	5	7
P4	3	5	2	2	4	6	4	5
P5	1	3	4	1	1	5	7	2

Available			
R1	R2	R3	R4
4	3	3	5

Tiến trình	Allocation				Need				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	3	1	1	2	2	2	3	1	4	3	3	5
P2	1	1	2	1	2	3	4	0	7	4	4	7
P3	2	1	4	5	1	4	1	2	8	5	6	8
P4	3	5	2	2	1	1	2	3	10	6	10	13
P5	1	3	4	1	0	2	3	1	13	11	12	15
Chuỗi an toàn <P1, P2, P3, P4, P5>									14	14	16	16

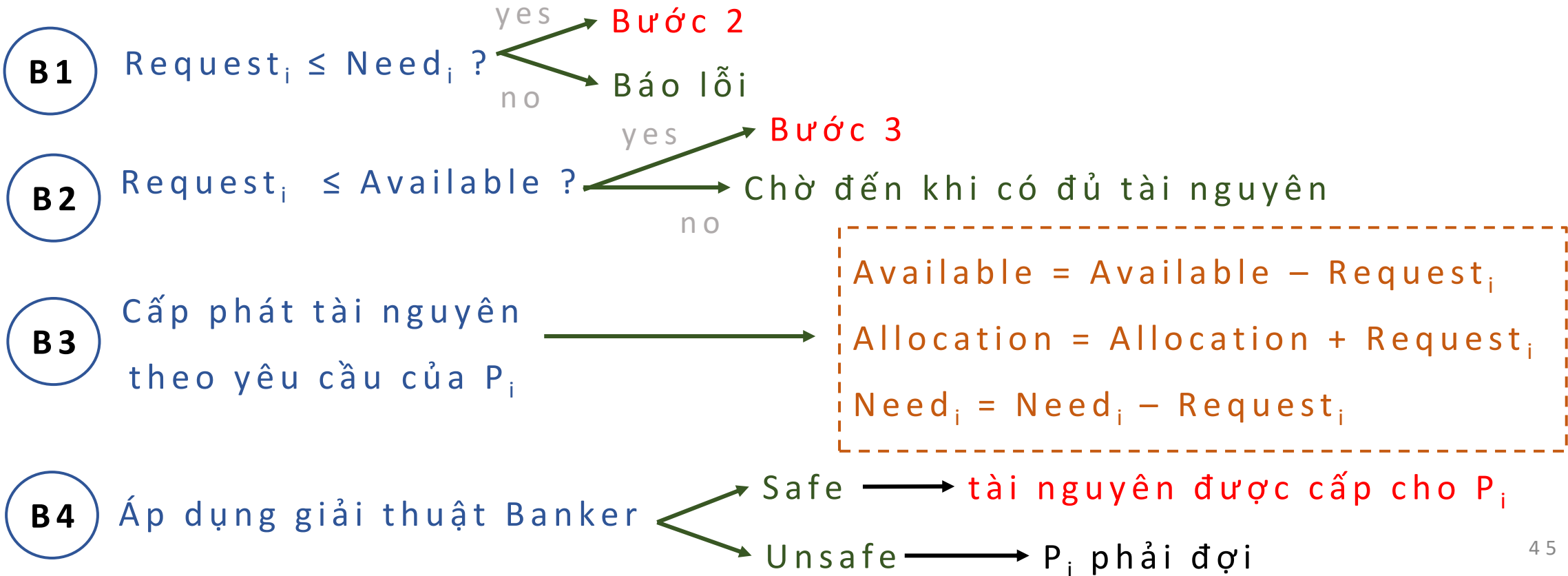


Nếu P3 yêu cầu thêm tài nguyên (1, 3, 1, 2) thì hệ thống có đáp ứng không?

Tiến trình	Allocation				Max			
	R1	R2	R3	R4	R1	R2	R3	R4
P1	3	1	1	2	5	3	4	3
P2	1	1	2	1	3	4	6	1
P3	2	1	4	5	3	5	5	7
P4	3	5	2	2	4	6	4	5
P5	1	3	4	1	1	5	7	2

Available			
R1	R2	R3	R4
4	3	3	5

Phương pháp giải:





Các giải thuật tránh deadlocks

Giải thuật Banker

Nếu P3 yêu cầu thêm tài nguyên (1, 3, 1, 2) thì hệ thống có đáp ứng không?

Tiến trình	Allocation				Max			
	R1	R2	R3	R4	R1	R2	R3	R4
P1	3	1	1	2	5	3	4	3
P2	1	1	2	1	3	4	6	1
P3	2	1	4	5	3	5	5	7
P4	3	5	2	2	4	6	4	5
P5	1	3	4	1	1	5	7	2

Available			
R1	R2	R3	R4
4	3	3	5

Tiến trình	Allocation				Need				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P3	2	1	4	5	1	4	1	2	8	5	6	8

Do $\text{Request}(3) \leq \text{Need}(3)$ và $\text{Request}(3) \leq \text{Available}$

$\text{Request}(3) = (1, 3, 1, 2)$

Tiến trình	Allocation				Need				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	3	1	1	2	2	2	3	1	3	0	2	3
P2	1	1	2	1	2	3	4	0				
P3	3	4	5	7	0	1	0	0	6	4	7	10
P4	3	5	2	2	1	1	2	3				
P5	1	3	4	1	0	2	3	1				

Trạng thái mới là safe (chuỗi an toàn <P3, P4, P5, P1, P2> vậy có thể cấp phát tài nguyên cho P3)

BT2: tại thời điểm t_0

	Allocation				Max			
Tiến trình	R1	R2	R3	R4	R1	R2	R3	R4
P1	1	2	2	3	2	3	4	3
P2	3	1	3	1	3	8	6	1
P3	2	1	4	5	7	7	5	7
P4	3	1	5	2	5	4	6	7
P5	1	4	4	2	1	6	7	3

Available			
R1	R2	R3	R4
3	4	4	3

Chọn phát biểu **SAI**:

- A. Tại thời điểm t_1 , nếu P4 yêu cầu thêm tài nguyên (2, 3, 1, 3) thì hệ thống sẽ đáp ứng.
- B. Trạng thái hiện tại của hệ thống là an toàn.
- C. Tại thời điểm t_1 , nếu P1 yêu cầu thêm tài nguyên (1, 2, 1, 2) thì hệ thống không đáp ứng
- D. Chuỗi (1,3,2,5,4) là một chuỗi an toàn của hệ thống**

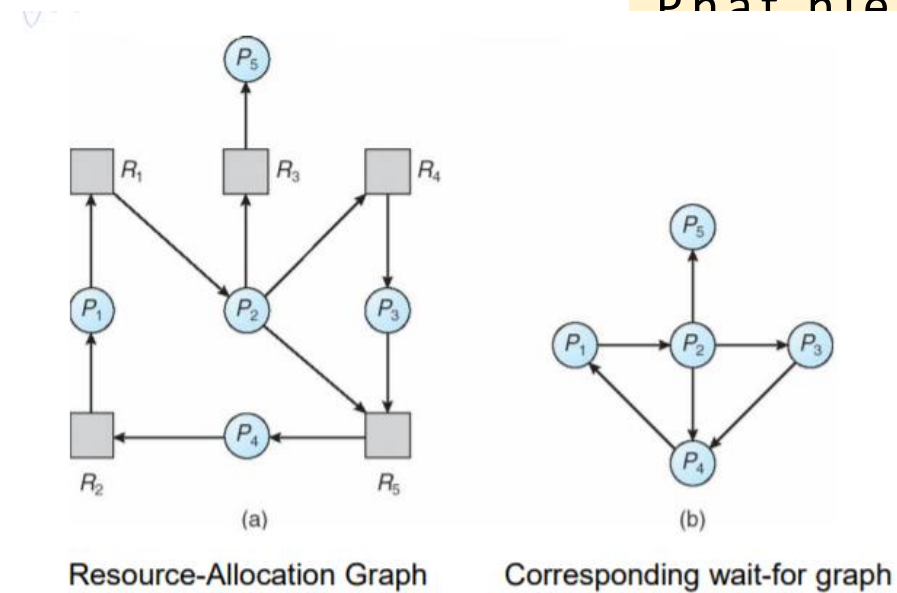
Tiến trình	Allocation				Need				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	1	2	2	3	1	1	2	0	3	4	4	3
P2	3	1	3	1	0	7	3	0	4	6	6	6
P3	2	1	4	5	5	6	1	2	7	7	9	7
P4	3	1	5	2	2	3	1	5	9	8	13	12
P5	1	4	4	2	0	2	3	1	12	9	18	14
Chuỗi an toàn <P1, P2, P3, P4, P5>									13	13	22	16

Phát hiện & phục hồi deadlock

Phát hiện deadlock

Mỗi loại tài nguyên
có **1 thực thể**

Dùng sơ đồ wait-for



Mỗi loại tài nguyên
có **nhiều thực thể**

1. Thực hiện tương tự giải thuật **Banker**
 2. Thay cột Need = cột Request
 3. Liệt kê chuỗi an toàn
- => Tiến trình không có trong chuỗi thì deadlock xảy ra tại tiến trình đó



- ✓ Báo người vận hành
- ✓ Chấm dứt một hay nhiều tiến trình
- ✓ Lấy lại tài nguyên từ một hay nhiều tiến trình

Chấm dứt lần lượt từng tiến trình cho đến khi không còn deadlock

Chấm dứt dựa trên:


1. Độ ưu tiên của tiến trình
2. Thời gian đã thực thi của tiến trình và thời gian còn lại
3. Loại tài nguyên tiến trình đã sử dụng
4. Tài nguyên mà tiến trình cần thêm để hoàn tất công việc
5. Số lượng tiến trình cần được chấm dứt
6. Tiến trình là interactive hay batch




Phục hồi deadlock

✓ Lấy lại tài nguyên từ một hay nhiều tiến trình

- Lấy lại từ 1 tiến trình, cấp cho tiến trình khác cho đến khi không còn deadlock
- Chọn “nạn nhân” để tối thiểu chi phí
- Trở lại trạng thái trước deadlock (Rollback)
- Đói tài nguyên (Starvation)



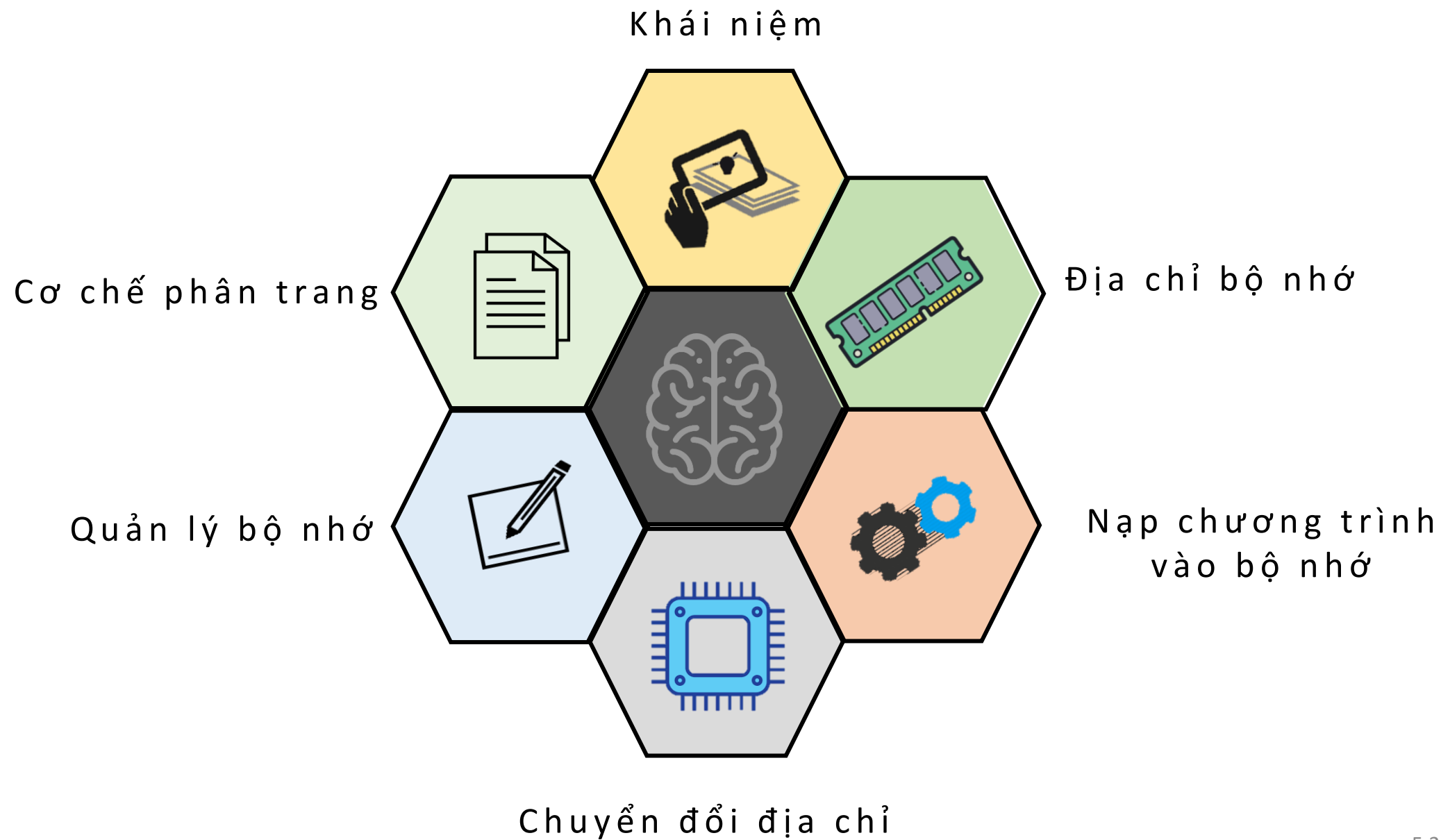
Bảo đảm không có tiến trình nào luôn luôn bị lấy lại tài nguyên mỗi khi có deadlock



Tiến trình bị lấy lại tài nguyên trở về trạng thái safe, tiếp tục tiến trình từ trạng thái đó. Hệ thống cần lưu giữ một số thông tin về trạng thái các tiến trình đang thực thi.

Chương 7

Quản lý bộ nhớ





Khái niệm

- Chương trình => mang vào trong bộ nhớ => đặt vào tiến trình => xử lý
- **Input Queue** – tập hợp những tiến trình trên đĩa, đang chờ để được mang vào trong bộ nhớ để thực thi.
- Hệ điều hành quản lý bộ nhớ với sự hỗ trợ của **phần cứng**
↓
phân phối, sắp xếp các process trong bộ nhớ
- Mục tiêu nạp **càng nhiều process** vào bộ nhớ càng tốt
- Trong hầu hết các hệ thống, **kernel sẽ chiếm một phần cố định** của bộ nhớ; phần còn lại phân phối cho các process.

Yêu cầu đối với
quản lý bộ nhớ

Cấp phát bộ nhớ cho các process

Tái định vị (relocation): khi swapping,...

Bảo vệ: phải kiểm tra truy xuất bộ nhớ có hợp lệ không

Chia sẻ: các process chia sẻ vùng nhớ chung

Gán địa chỉ nhớ luận lý của user vào địa chỉ thực



Địa chỉ bộ nhớ

Địa chỉ vật lý

(Physical/ địa chỉ thực)



1 vị trí thực trong bộ nhớ chính

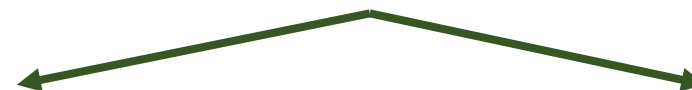
Địa chỉ luận lý

(Logical/ địa chỉ ảo)



1 vị trí nhớ trong một chương trình

*trình biên dịch dịch mã => mọi tham chiếu
bộ nhớ đều là địa chỉ logic*



Địa chỉ tương đối

địa chỉ được biểu diễn
tương đối so với một vị trí
nào đó trong chương trình

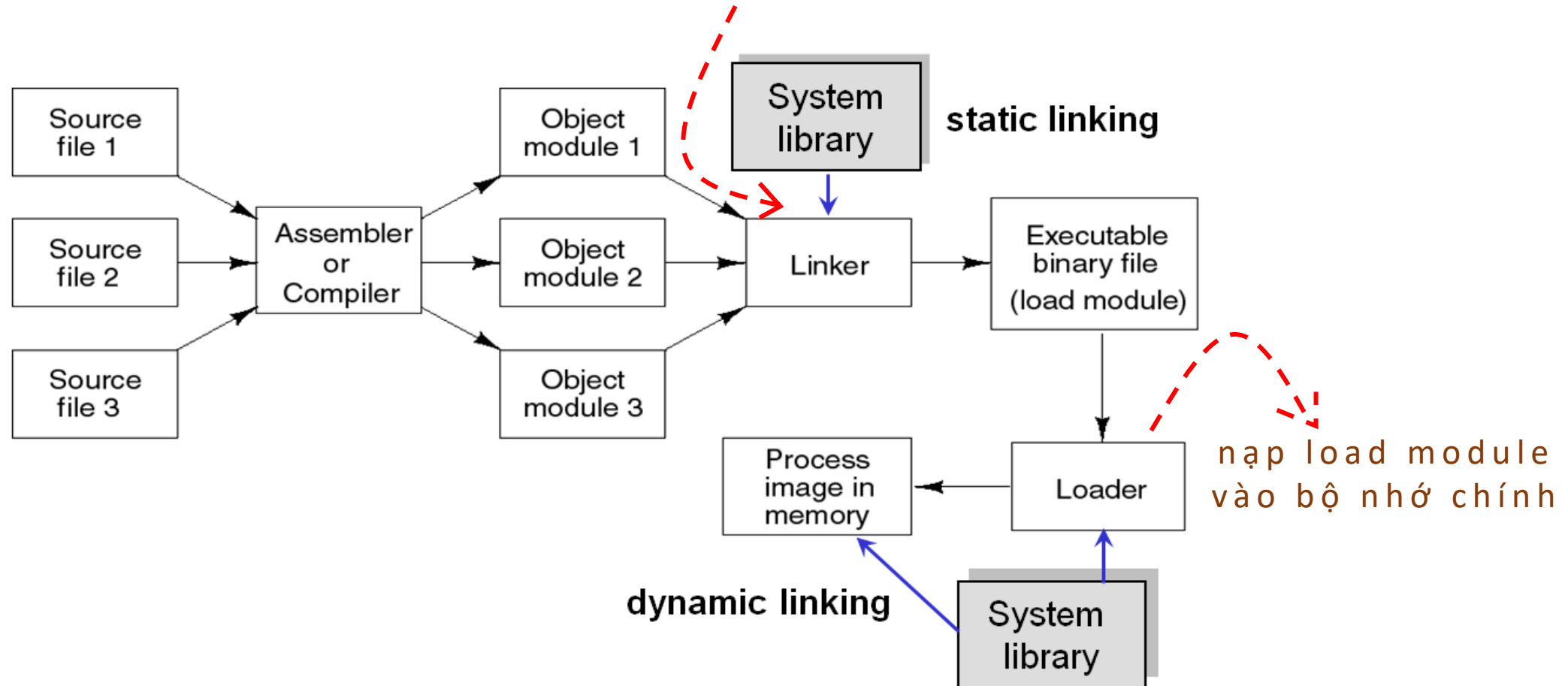
Địa chỉ tuyệt đối

= địa chỉ thực



Nạp chương trình vào bộ nhớ

Tổng hợp object module => file nhị phân load module

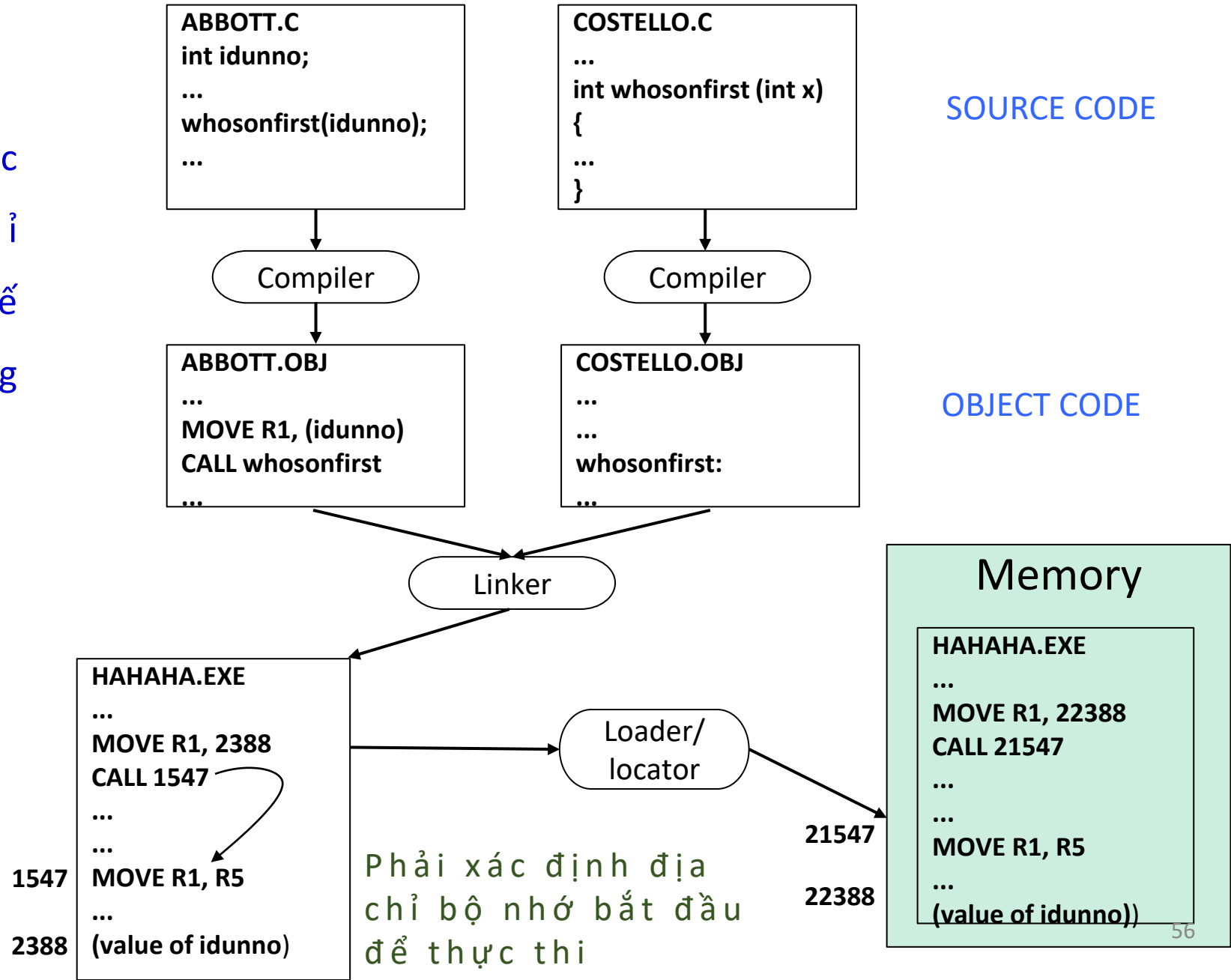




Nạp chương trình vào bộ nhớ

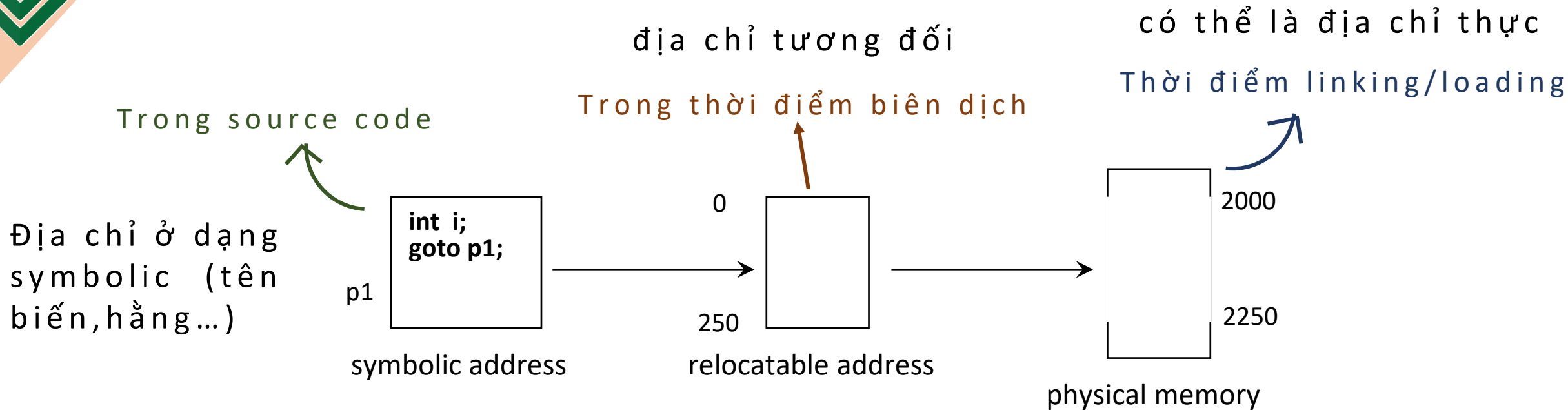
Khi mỗi file được biên dịch, các địa chỉ là chưa biết, vì thế trình biên dịch dùng các cờ để đánh dấu

Trình linker kết nối các files => nó có thể thay thế các chỗ đánh dấu với địa chỉ thật





Chuyển đổi địa chỉ

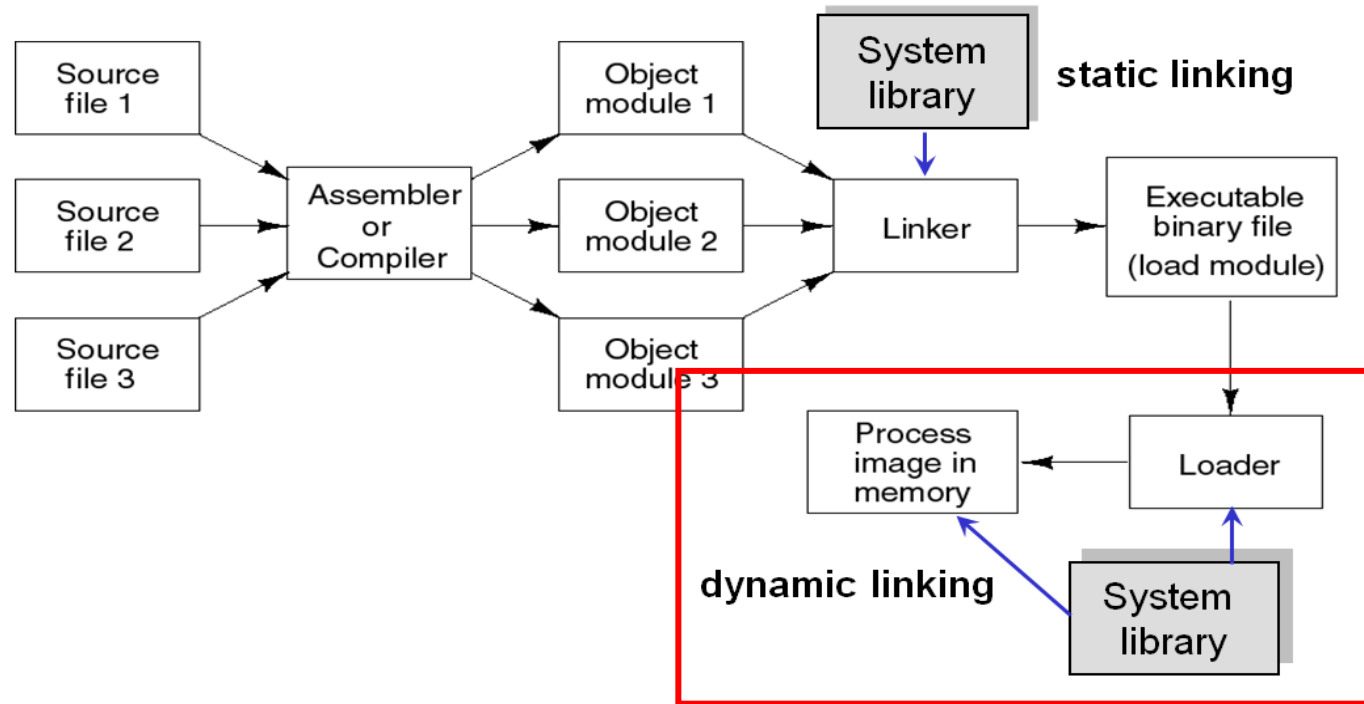


Chuyển địa chỉ thành
địa chỉ thực tại

Compile time: biết trước địa chỉ bộ nhớ của chương trình => gán địa chỉ tuyệt đối khi compile

Load time: loader phải chuyển đổi địa chỉ tương đối thành địa chỉ thực dựa trên một địa chỉ nền

Thời điểm **thực thi chương trình**



Là quá trình **kết nối** đến một **external module** sau khi đã tạo xong **load module**

- ✓ Giúp chương trình thực thi có thể dùng các phiên bản khác nhau của external module mà không cần sửa đổi, biên dịch lại.
- ✓ code sharing: một external module chỉ cần nạp vào bộ nhớ một lần. Các process cần dùng external module này thì cùng chia sẻ đoạn mã ⇒ tiết kiệm không gian nhớ



- ✓ Chỉ nạp thủ tục vào bộ nhớ khi được gọi đến
=> tăng hiệu suất bộ nhớ, tránh lãng phí
- ✓ Rất hiệu quả trong trường hợp tồn tại khối lượng lớn mã chương trình có tần suất sử dụng thấp
- ✓ User chịu trách nhiệm thiết kế và hiện thực các chương trình có dynamic loading
- ✓ Hệ điều hành cung cấp một số thủ tục thư viện hỗ trợ, tạo điều kiện cho lập trình viên



Phân mảnh

Phân mảnh ngoại

không gian nhớ còn trống đủ để thỏa mãn một yêu cầu nhưng không gian nhớ này **không liên tục**

⇒ dùng cơ chế kết khối để **gom lại thành vùng nhớ liên tục**.

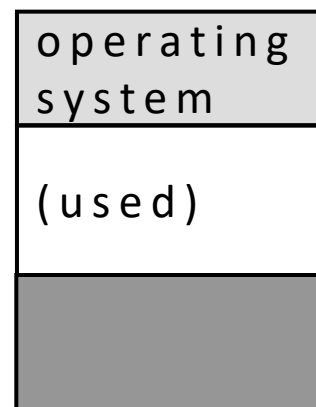
Phân mảnh nội

Xảy ra khi bộ nhớ thực được **chia thành các khối kích thước cố định** và các process được cấp phát theo đơn vị khối.

vùng nhớ được cấp phát có thể **nhiều hơn** vùng nhớ được yêu cầu

Yêu cầu: 18000 byte

Hiện có: 18002 byte



Dư ra 2 byte không dùng => cấp luôn 18002 byte

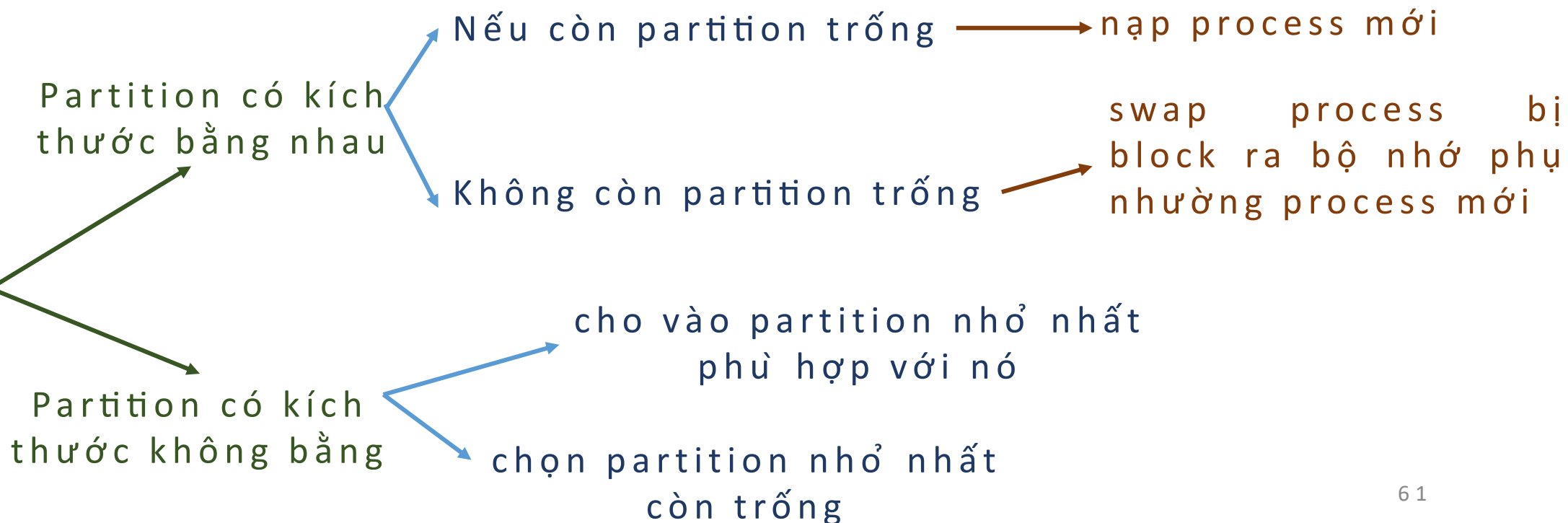


Chiến lược placement

Đặt vấn đề

- Khi khởi động, bộ nhớ chính được chia thành nhiều phần (partition)
- Process có kích thước \leq partition \rightarrow được nạp vào partition
- Chương trình có kích thước $>$ partition \rightarrow dùng cơ chế overlay
- Do bị phân mảnh nội \rightarrow một chương trình nhỏ cũng được cấp phát trọn 1 partition

Giải pháp

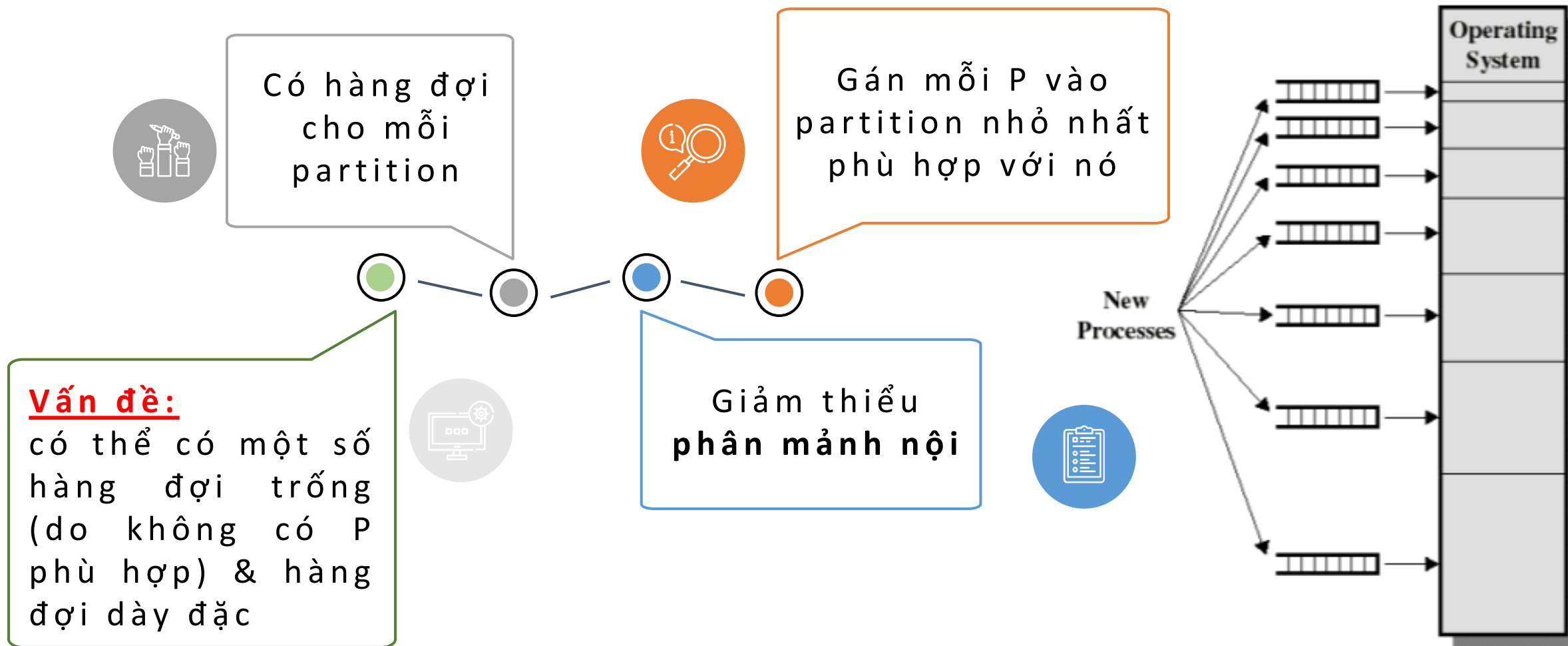




Chiến lược placement

Partition có kích thước không bằng nhau

Giải pháp 1

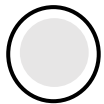




Partition có kích thước không bằng nhau

Giải pháp 2

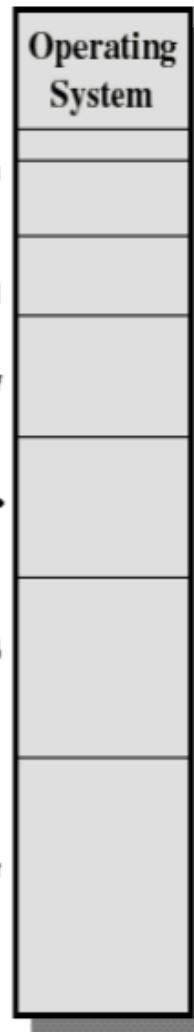
1 hàng đợi chung
cho mọi partition



Nạp P vào bộ nhớ chính

chọn partition nhỏ
nhất còn trống

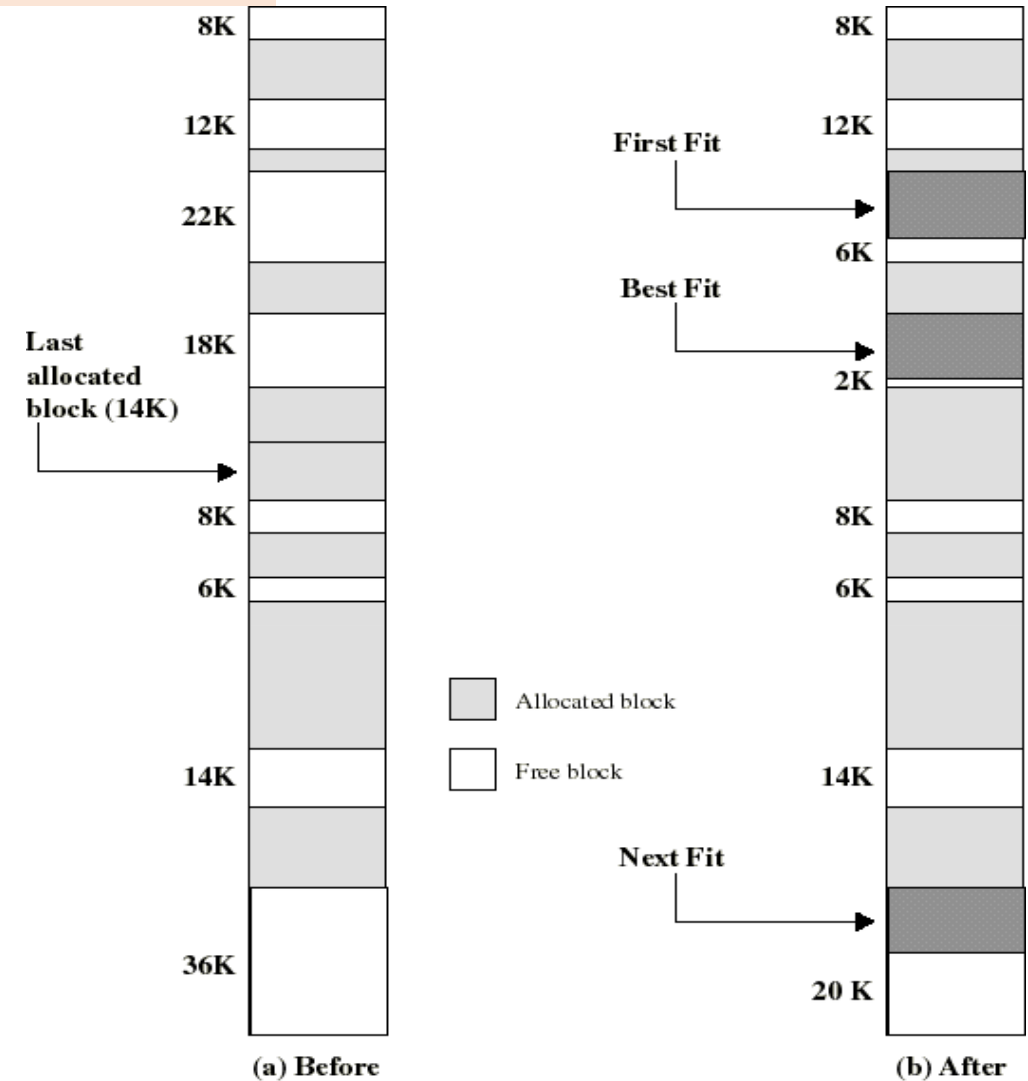
New
Processes





Chiến lược placement

- ✓ Best-fit: chọn khối nhớ trống nhỏ nhất
- ✓ First-fit: chọn khối nhớ trống phù hợp đầu tiên kể từ đầu bộ nhớ
- ✓ Next-fit: chọn khối nhớ trống phù hợp đầu tiên kể từ vị trí cấp phát cuối cùng
- ✓ Worst-fit: chọn khối nhớ trống lớn nhất



Example Memory Configuration Before and After Allocation of 16 Kbyte Block



Bộ nhớ vật lý: là **khung trang** (frame), có kích thước lũy thừa của 2

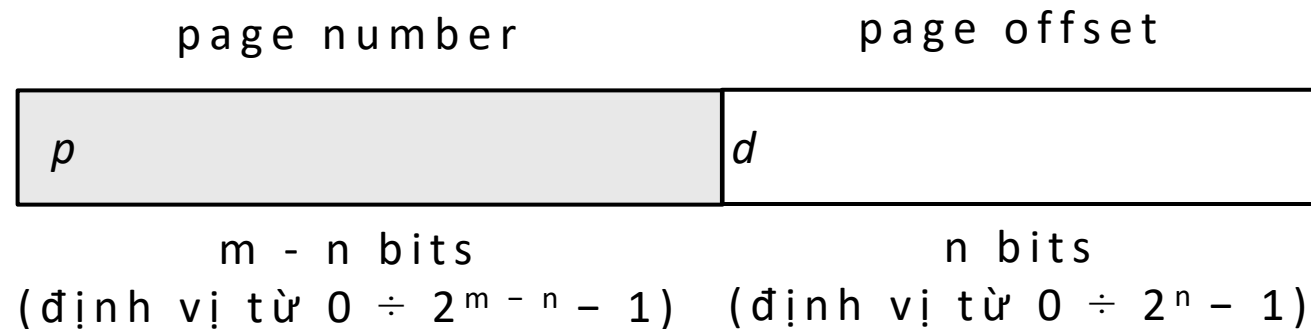
Bộ nhớ luận lý: tập hợp địa chỉ logic mà 1 chương trình có thể sinh ra → **page**

Bảng phân trang: chuyển địa chỉ luận lý → địa chỉ thực

Địa chỉ luận lý gồm:

- Số trang (Page number) **p**
- Địa chỉ tương đối trong trang (Page offset) **d**

Kích thước của không gian địa chỉ ảo là 2^m , và kích thước của trang là 2^n thì



Bảng trang sẽ có tổng cộng $2^m / 2^n = 2^{m-n}$ mục (entry)



Cơ chế phân trang

Cơ chế

0	0
1	1
2	2
3	3

Process A
page table

0	—
1	—
2	—

Process B
page table

0	7
1	8
2	9
3	10

Process C
page table

0	4
1	5
2	6
3	11
4	12

Process D
page table

13
14

Free frame
list

page
number

0	
1	
2	
3	

logical memory

frame
number

0	1
1	4
2	3
3	5

page table

0

1

2

3

4

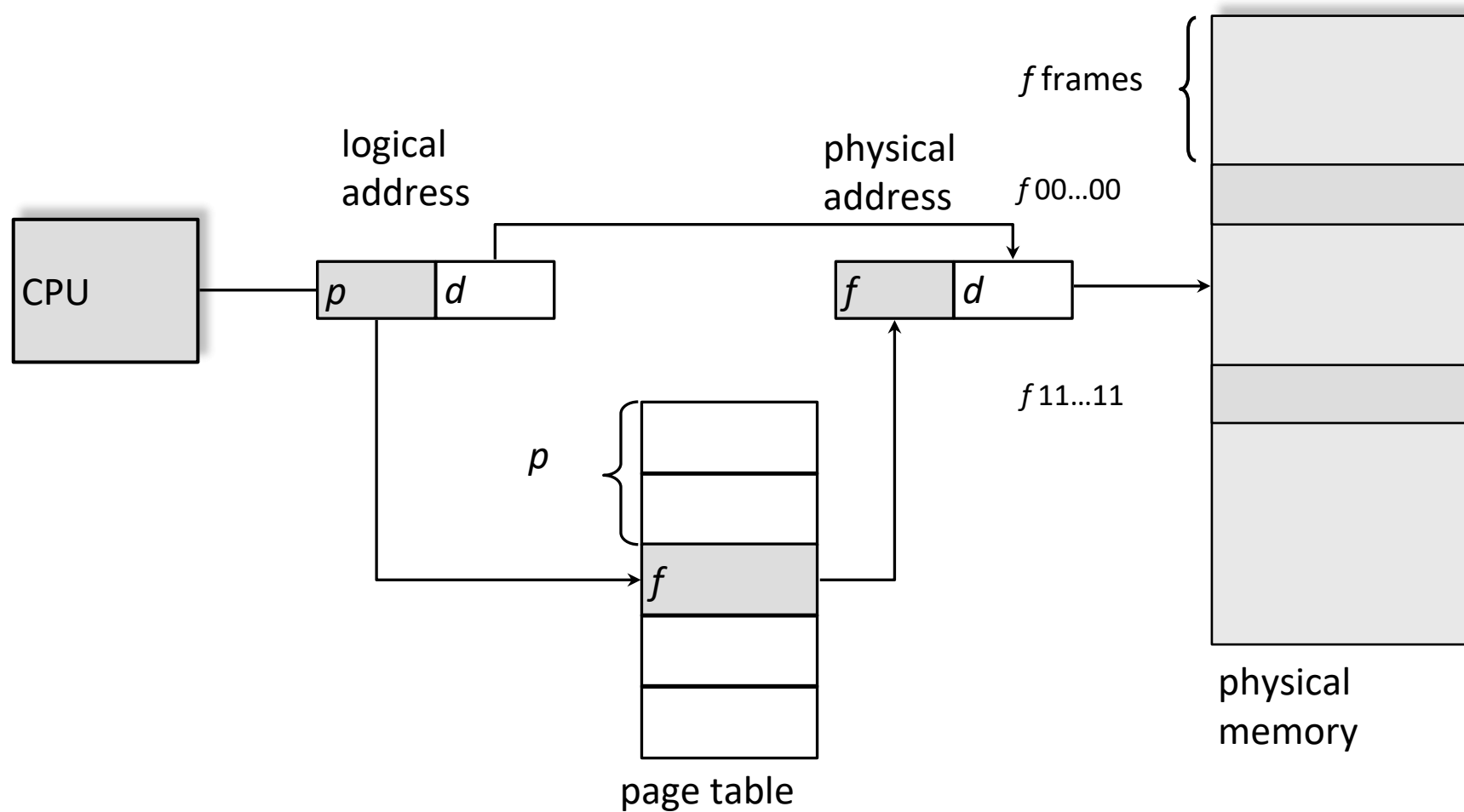
page 0
page 2
page 1

physical memory



Cơ chế phân trang

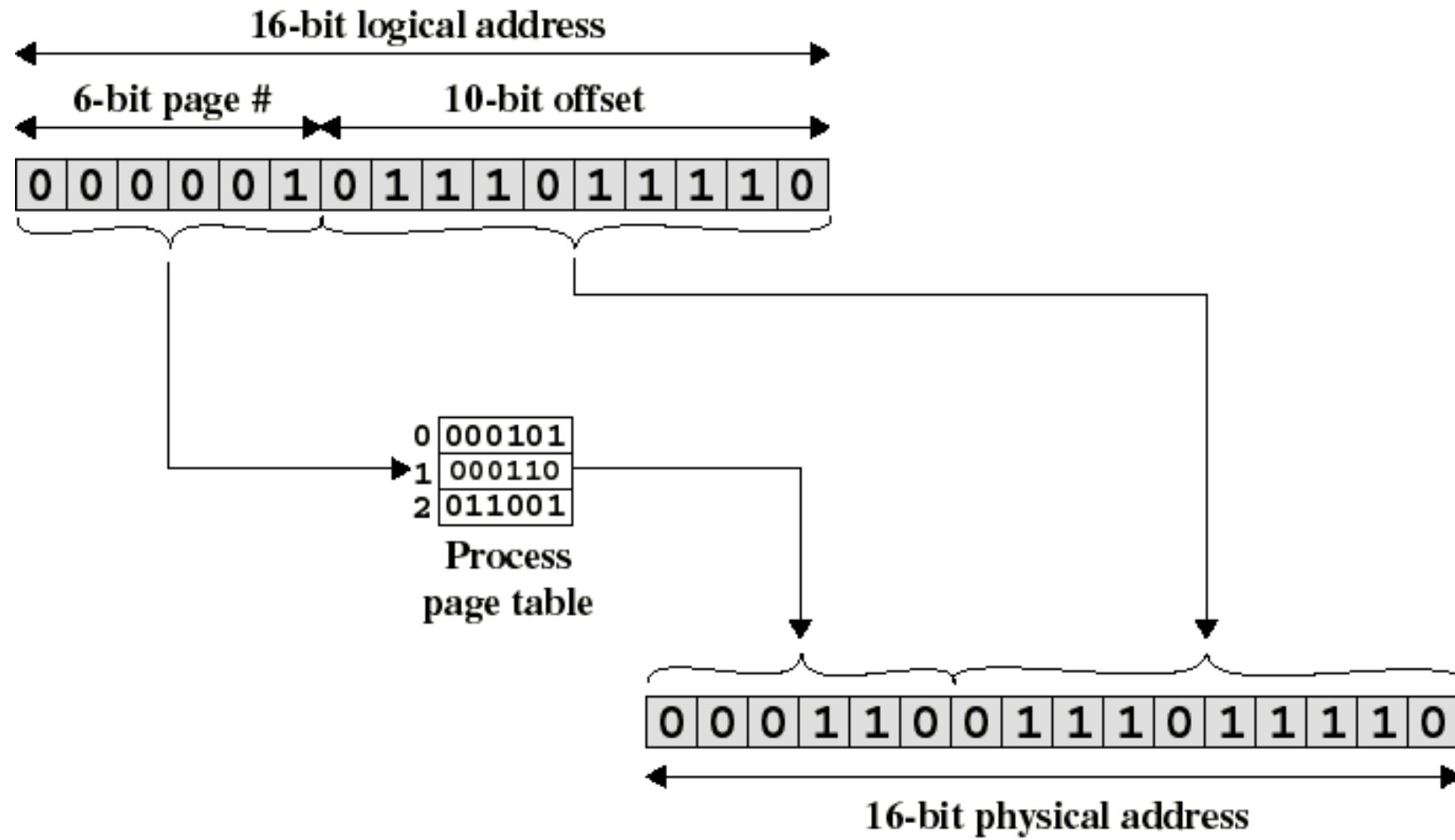
Chuyển đổi địa chỉ trong phân trang





Cơ chế phân trang

Chuyển đổi địa chỉ trong phân trang





BT1: Hệ thống có bộ nhớ được cấp phát theo cơ chế phân trang với kích thước trang và khung trang là 1024 byte. Biết trang 0 và trang 1 của bộ nhớ ảo lần lượt được nạp vào khung trang 4, 2 của bộ nhớ vật lý. Hỏi địa chỉ ảo 684 được ánh xạ thành địa chỉ vật lý bao nhiêu?

A. 684

B. 2732

C. 4780

D. 1708

Kích thước 1024 = $2^{10} \Rightarrow n = 10$ bit

Địa chỉ tương đối là 684 $\Rightarrow 684 = 1010101$

Mà $n = 10$ bit \Rightarrow cắt từ 684 n bit: $\dots 0 \mid 10$

m bit
số 0

684 vừa đủ
10 bit

Địa chỉ vật lý = **100** | 1010101100 = 4780



BT2: Bộ vi xử lý MIPS R2000 có không gian địa chỉ ảo 32 bit với kích thước trang (page) là 4096 byte. Mỗi mục (entry) trong bảng trang có kích thước 32 bit. Hỏi kích thước của bảng trang là bao nhiêu?

A. 0.5 MB

B. 1 MB

C. 2 MB

D. 4 MB

Kích thước trang $2^n = 4096$ suy ra $n = 12$

Số mục $= 2^{m-n} = 2^{32-12} = 2^{20}$

Kích thước bảng trang $= 2^{20} \times 32 = 4\text{MB}$

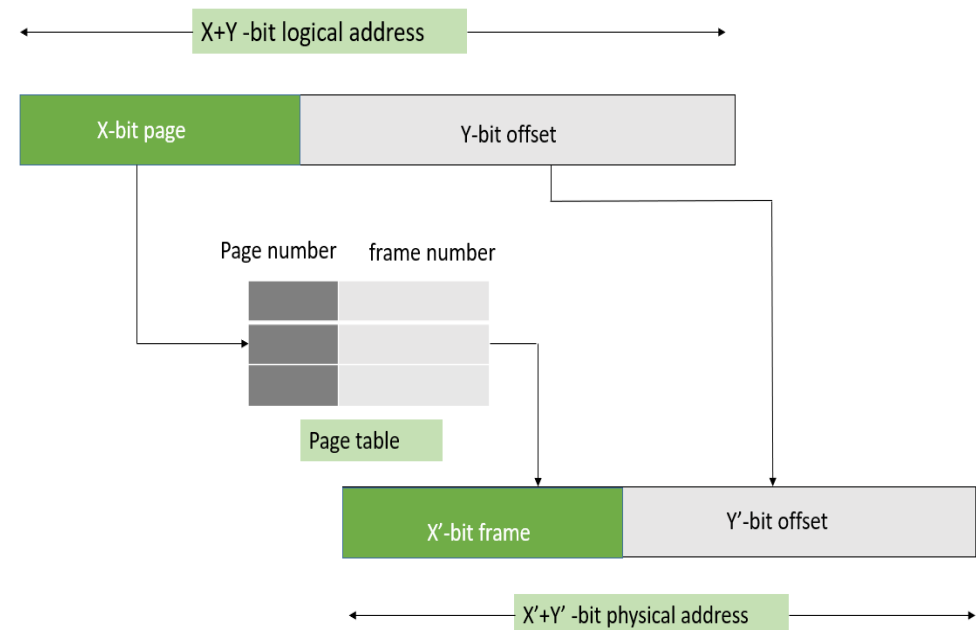


Cơ chế phân trang

Bài tập phân trang

BT3. Bộ nhớ vật lý của một hệ thống có kích thước 4MB được chia thành 256 khung trang. Để quản lý bộ nhớ này, hệ thống sử dụng một bảng trang với 32 trang. Địa chỉ luận lý có tối thiểu bao nhiêu bit để truy xuất bộ nhớ trên?

- A. 19 bit C. 14 bit
- B. 22 bit D. 13 bit



Kích thước bộ nhớ vật lý = 4MB = 2^{22} Byte
Số khung trang = 256 \rightarrow cần tối thiểu 8 bit trang
 \rightarrow Số bit offset = $22 - 8 = 14$ bit
Số trang 32 \rightarrow cần tối thiểu 5bit khung trang
 \rightarrow Địa chỉ luận lý tối thiểu: $5 + 14 = 19$ bit



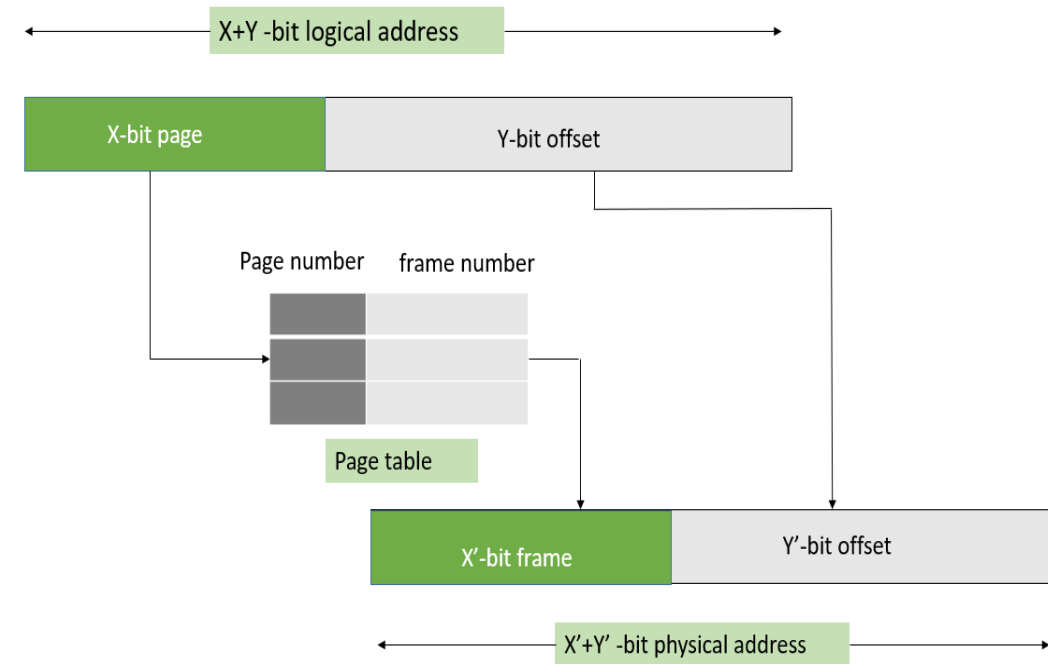
BT4. Bộ nhớ vật lý của một hệ thống có kích thước 8MB được quản lý bởi một bảng trang có 32 trang. Hệ thống sử dụng địa chỉ luận lý 20 bit. Hỏi kích thước của mỗi khung trang của bộ nhớ là bao nhiêu?

A. 32 KB

B. 8 KB

C. 16 KB

D. 1 KB



Có 32 trang → cần tối thiểu 5 bit trang

Địa chỉ luận lý 20 bit

→ Số bit offset = $20 - 5 = 15$ bit

→ **Kích thước của mỗi khung trang = 2^{15} bytes = 32 KB**



BT5. Xét một không gian địa chỉ ảo có 112 trang, mỗi trang có kích thước 2048 bytes được ánh xạ vào bộ nhớ có 64 khung trang. Kích thước bảng phân trang là bao nhiêu nếu mỗi mục(entry) trong nó cần 1 bytes?

A. 58 bytes

B. 128 bytes

C. 112 bytes

D. 64 bytes

Số trang = 112 \rightarrow cần tối thiểu 7 bit (vì $2^7 = 128$)

Mỗi trang có kích thước 2048 bytes = 2^{11}

\rightarrow Kích thước địa chỉ ảo = $2^7 * 2^{11} = 2^{18}$

Số mục = Kích thước địa chỉ ảo / kích thước mỗi trang = 2^7

\rightarrow **Kích thước bảng phân trang = $2^7 * 1 = 2^7$ bytes**



BT6. Một máy tính có không gian địa chỉ ảo 32 bit, quản lý bộ nhớ bằng cách sử dụng kết hợp phân trang và phân đoạn. Trong đó 4 bit đầu tiên là dành cho đoạn, 16 bit kế tiếp dành cho trang, số bit còn lại dành cho offset. Khi tiến trình truy xuất địa chỉ 0xC0DEDBAD thì chỉ số trang là bao nhiêu?

A. 0xC0

B. 0xC0DE

☒ C. 0x0DED

D. 0xBAD

0xC0DEDBAD (hex) =

1100 0000 1101 1110 1101 1011 1010 1101 (bin)

→ 0000 1101 1110 1101 = 0DED.



BT7. Xét một hệ thống sử dụng kỹ thuật phân trang, với bảng trang được lưu trữ trong bộ nhớ chính. Nếu thời gian cho một lần truy xuất bộ nhớ bình thường là 200ns thì mất bao nhiêu thời gian cho một thao tác truy xuất bộ nhớ trong hệ thống này?

A. 400ns

B. 200ns

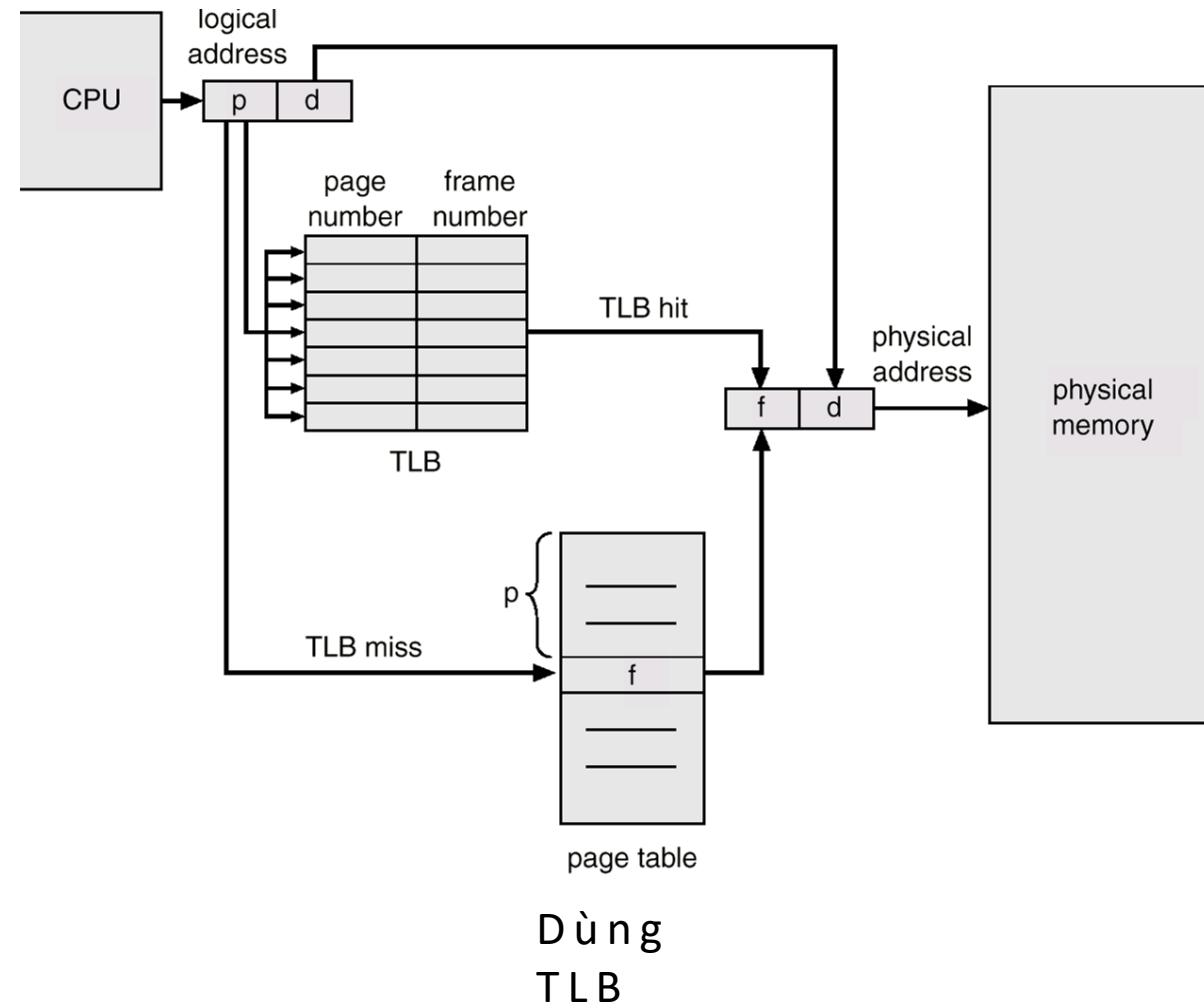
C. 800ns

D. 100ns

$$x + x = 200 + 200 = 400ns$$



- Bảng phân trang lưu trong bộ nhớ chính
- Mỗi process được hệ điều hành cấp một bảng phân trang
- ✓ Thanh ghi PTBR trỏ đến bảng phân trang
- ✓ Thanh ghi PTLR thể hiện kích thước của bảng
- ✓ Thanh ghi TLB: bộ phận cache phần cứng có tốc độ truy xuất và tìm kiếm cao





$$EAT = (\epsilon + x)\alpha + (\epsilon + 2x)(1 - \alpha) = (2 - \alpha)x + \epsilon$$

α (**hit ratio**): tỉ số giữa số lần trang được tìm thấy trong TLB và số lần truy xuất từ CPU

ϵ : Thời gian tìm kiếm trong TLB

x : Thời gian một chu kỳ truy xuất bộ nhớ

Ví dụ

Sử dụng TLBs với hit-ratio (tỉ lệ tìm thấy) là 90% thì thời gian truy xuất bộ nhớ trong hệ thống (effective memory reference time) là 240 ns. Nếu tỉ lệ tìm thấy là 80% thì thời gian truy xuất bộ nhớ trong hệ thống là 260ns. Tính thời gian để tìm trong TLBs?

A. 200

C. 40

B. 20

D. 220

Ta có hệ phương trình:

$$(2 - 0.9)x + \epsilon = 240 \quad (1)$$

$$(2 - 0.8)x + \epsilon = 260 \quad (2)$$

$$\Rightarrow x = 200\text{ns}; \epsilon = 20\text{ns}$$

Chương 8

Bộ nhớ ảo

Tổng quan

Cài đặt

3 giải thuật thay trang

Vấn đề cấp phát

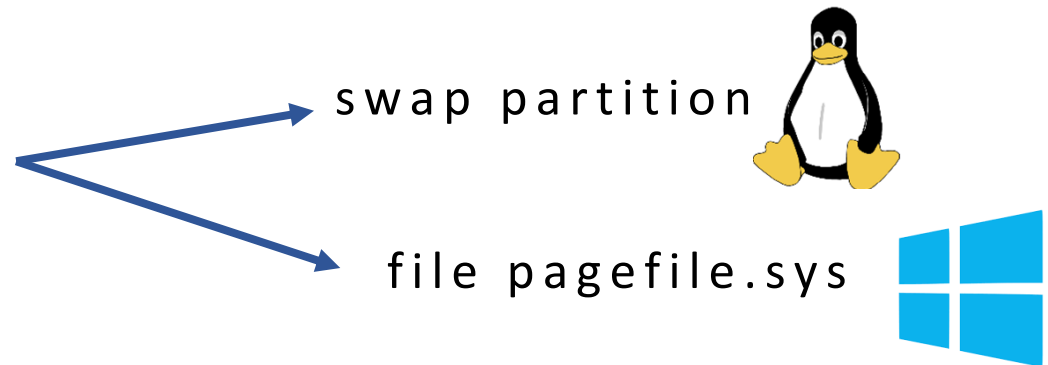
Vấn đề thrashing

Bộ nhớ ảo (virtual memory): 1 tiến trình không được nạp toàn bộ vào bộ nhớ vật lý => xử lý trong bộ nhớ ảo

Ưu điểm

- ✓ Số lượng process trong bộ nhớ nhiều hơn
- ✓ Một process có thể thực thi cả khi kích thước > bộ nhớ thực
- ✓ Giảm nhẹ công việc của lập trình viên

Không gian trao đổi giữa bộ nhớ chính và bộ nhớ phụ (swap space)



2 kỹ thuật:

- Phân trang theo yêu cầu (Demand Paging)

- Phân đoạn theo yêu cầu (Demand Segmentation)

✓ Phần cứng memory management phải hỗ trợ 2 kỹ thuật

✓ OS phải quản lý sự di chuyển của trang/đoạn giữa bộ nhớ chính và bộ nhớ ảo

các trang chỉ được nạp vào bộ nhớ chính khi được yêu cầu

tham chiếu đến 1 trang không có
trong bộ nhớ chính => page-fault

Phần cứng gọi
page-fault trap

Khởi động page-
fault service
routine (PFSR)

Cài đặt bộ nhớ ảo

PFSR

Block process đó

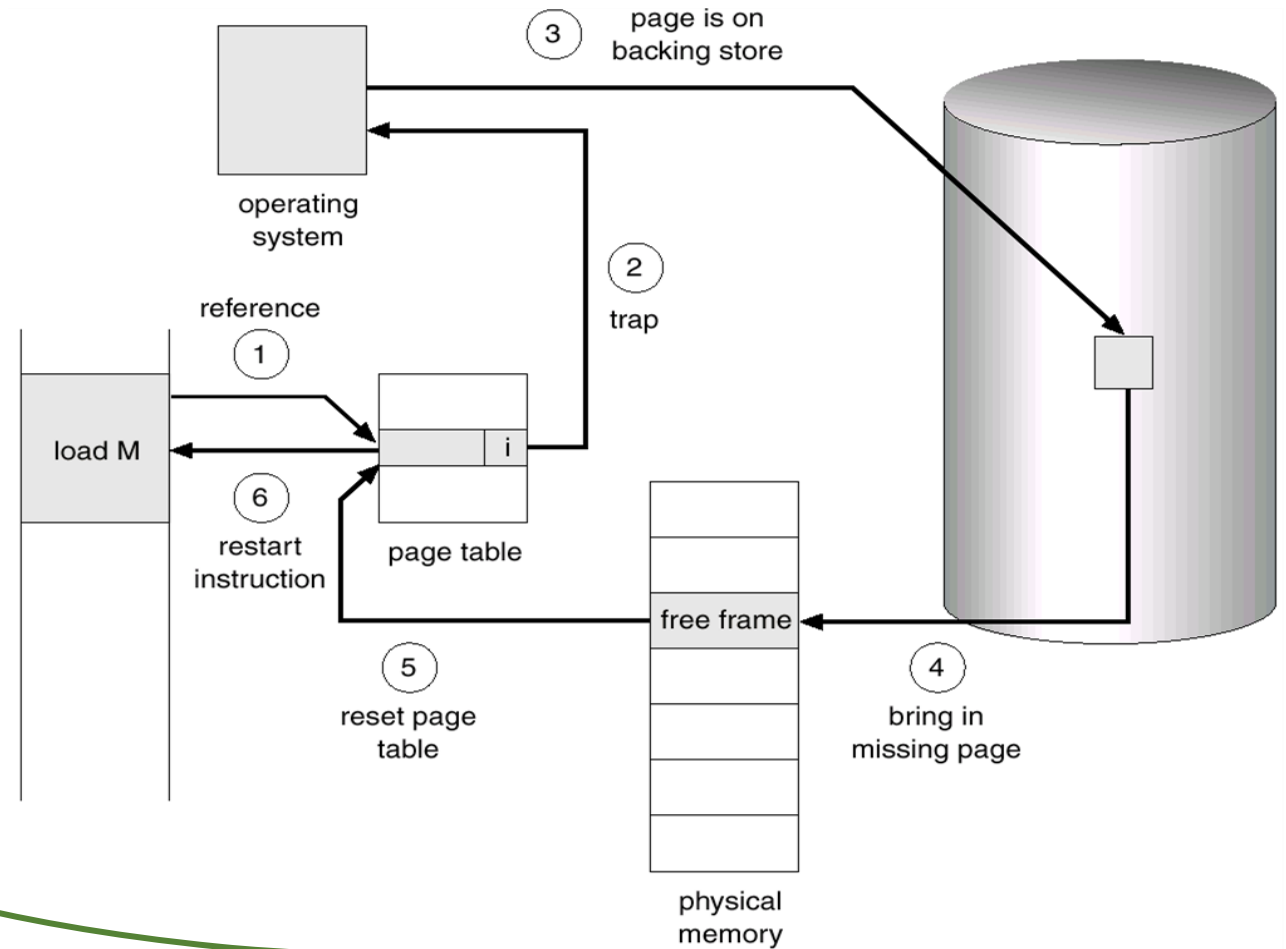
Yêu cầu đọc đĩa để thêm trang mới

Trong lúc đợi I/O, 1 process khác
có thể thực thi trước

I/O ready => đĩa gọi ngắt
=> update page table
=> unblocked process

Không có
frame trống

Dùng **giải thuật thay trang** chọn một trang hy sinh (**victim page**)
Ghi victim page lên đĩa

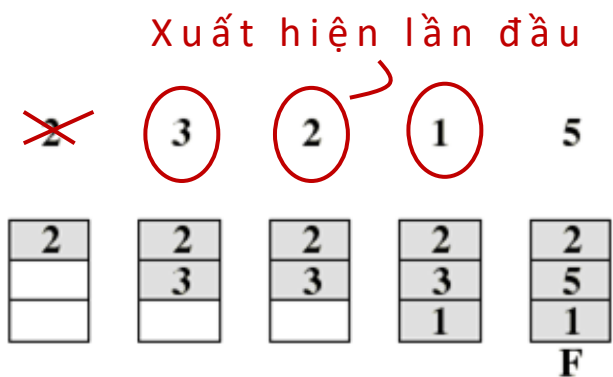


Giải thuật thay trang

Mục tiêu: số lượng page-fault nhỏ nhất

LRU

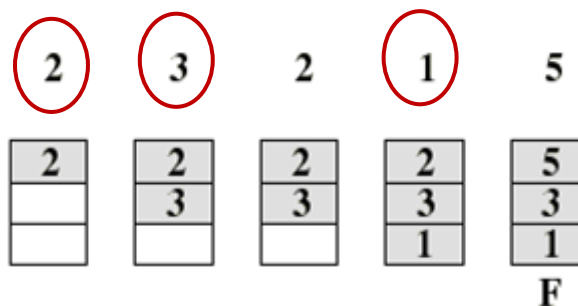
Thay trang có thời điểm tham chiếu (đầu tiên) nhỏ nhất trong quá khứ



tốn chi phí tìm kiếm

FIFO

Vào trước ra trước



OPT

Thay trang nhớ được tham chiếu trễ nhất trong tương lai

2	1	2	0	1	7	0	1
2	2	2	2	2	7	7	7
0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1
	*				*		

Có thể dọn page không cần thiết

1 tiến trình được cấp 4 khung trang trong bộ nhớ vật lý và 7 trang trong bộ nhớ ảo. Tại thời điểm nạp tiến trình vào, 4 khung trang này đang trống. Tiến trình truy xuất 7 trang (1, 2, 3, 4, 5, 6, 7) trong bộ nhớ ảo theo thứ tự như sau:

6 2 4 4 5 6 3 1 4 2 3 7 5 6 7 2 4 3 5 1

Tại thời điểm tiến trình truy xuất trang nhớ số 3 lần đầu tiên, trang nhớ nào sẽ bị thay thế, nếu sử dụng giải thuật FIFO?

A. 6

B. 2

C. 4

D. 5

FIFO	6	2	4	4	5	6	3	1	4	2	3	7	5	6	7	2	4	3	5	1
	6	6	6	6	6	6	3	3	3	3	3	3	5	5	5	5	5	5	5	1
		2	2	2	2	2	2	1	1	1	1	1	1	6	6	6	6	6	6	6
			4	4	4	4	4	4	4	2	2	2	2	2	2	2	4	4	4	4
					5	5	5	5	5	5	5	7	7	7	7	7	7	3	3	3
	*	*	*		*		*	*		*		*	*	*			*	*		*

1 tiến trình được cấp 4 khung trang trong bộ nhớ vật lý và 7 trang trong bộ nhớ ảo. Tại thời điểm nạp tiến trình vào, 4 khung trang này đang trống. Tiến trình truy xuất 7 trang (1, 2, 3, 4, 5, 6, 7) trong bộ nhớ ảo theo thứ tự như sau:

6 2 4 4 5 6 3 1 4 2 3 7 5 6 7 2 4 3 5 1

Tại thời điểm tiến trình truy xuất trang nhớ số 1 lần đầu tiên, trang nhớ nào sẽ bị thay thế, nếu sử dụng giải thuật OPT?

A. 6

B. 2

C. 4

D. 5

OPT	6	2	4	4	5	6	3	1	4	2	3	7	5	6	7	2	4	3	5	1
	6	6	6	6	6	6	3	3	3	3	3	3	5	6	6	6	6	6	6	6
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3
			4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	5	5
					5	5	5	1	1	1	1	7	7	7	7	7	7	7	7	1
	*	*	*		*		*	*				*	*	*				*	*	*

1 tiến trình được cấp 4 khung trang trong bộ nhớ vật lý và 7 trang trong bộ nhớ ảo. Tại thời điểm nạp tiến trình vào, 4 khung trang này đang trống. Tiến trình truy xuất 7 trang (1, 2, 3, 4, 5, 6, 7) trong bộ nhớ ảo theo thứ tự như sau:

6 2 4 4 5 6 3 1 4 2 3 7 5 6 7 2 4 3 5 1

Tại thời điểm tiến trình truy xuất trang nhớ số 7 lần đầu tiên, có tất cả bao nhiêu lỗi trang đã xảy ra (không tính lỗi trang xảy ra khi nạp trang nhớ số 7), nếu sử dụng giải thuật LRU?

A. 6

B. 2

C. 7

D. 8

LRU	6	2	4	4	5	6	3	1	4	2	3	7	5	6	7	2	4	3	5	1
	6	6	6	6	6	6	6	6	6	2	2	2	2	6	6	6	6	3	3	3
		2	2	2	2	2	3	3	3	3	3	3	3	3	3	2	2	2	2	1
			4	4	4	4	4	1	1	1	1	7	7	7	7	7	7	7	5	5
					5	5	5	5	4	4	4	4	5	5	5	5	4	4	4	4
	*	*	*		*		*	*	*	*		*	*	*		*	*	*	*	*



Giải thuật thay trang

Nghịch lý Belady

Sử dụng 3 khung trang, sẽ có 9 lỗi trang phát sinh

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4
*	*	*	*	*	*	*			*	*	

Sử dụng 4 khung trang, sẽ có 10 lỗi trang phát sinh

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3
*	*	*	*			*	*	*	*	*	*

Nghịch lý Belady là số page fault tăng mặc dù quá trình đã được cấp nhiều frame hơn

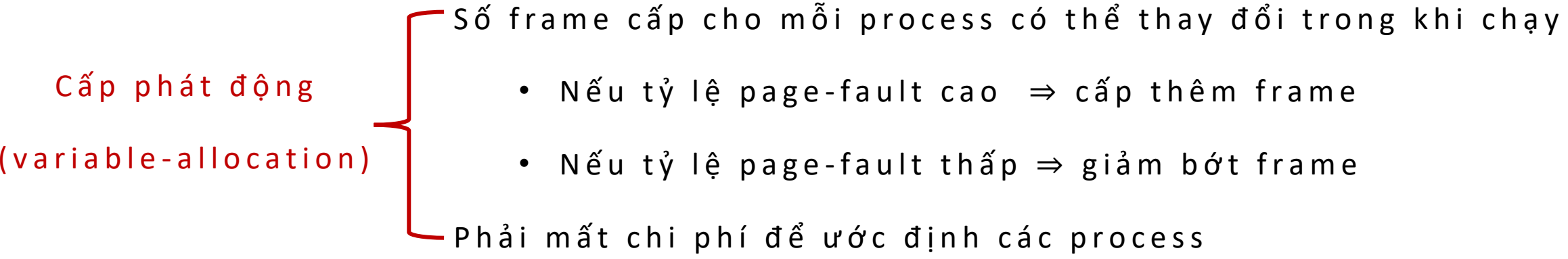
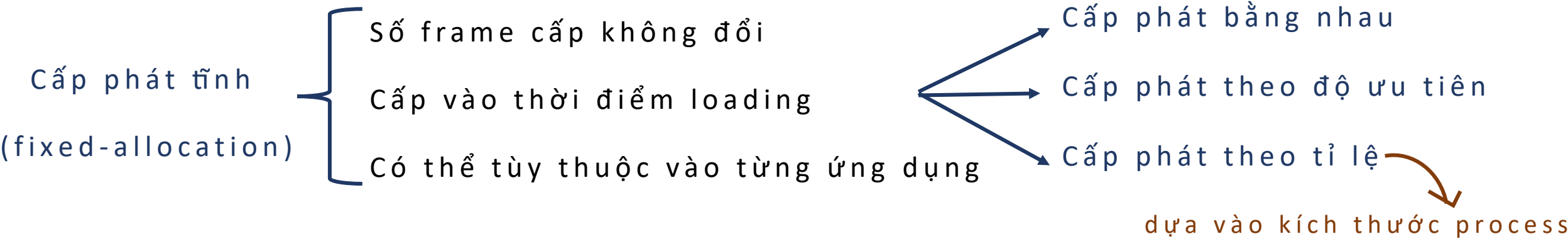
Chỉ xảy ra trong FIFO
Khắc phục được bởi OPT



Vấn đề cấp phát

OS quyết định cấp cho mỗi process bao nhiêu frame.

- Cấp ít frame \Rightarrow nhiều page fault
- Cấp nhiều frame \Rightarrow giảm mức độ multiprogramming





Vấn đề thrashing

Thrashing: hiện tượng các trang nhớ của process bị hoán chuyển vào/ra liên tục

Hạn chế thrashing => hệ điều hành phải cấp cho process càng “đủ” frame càng tốt

Nguyên lý locality (locality principle)

- Locality: tập hợp các trang được tham chiếu gần nhau
- Một process có nhiều locality, trong quá trình thực thi, process sẽ chuyển từ locality này sang locality khác

Hiện tượng thrashing xuất hiện khi:

$$\Sigma \text{ size of locality} > \text{memory size}$$

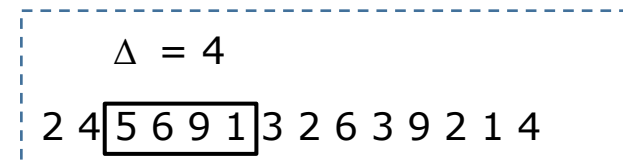


Thiết kế dựa trên nguyên lý locality

Xác định xem process sử dụng bao nhiêu frame

Định nghĩa:

- $WS(t)$ - số lượng các tham chiếu gần nhất cần được quan sát
- Δ - khoảng thời gian tham chiếu



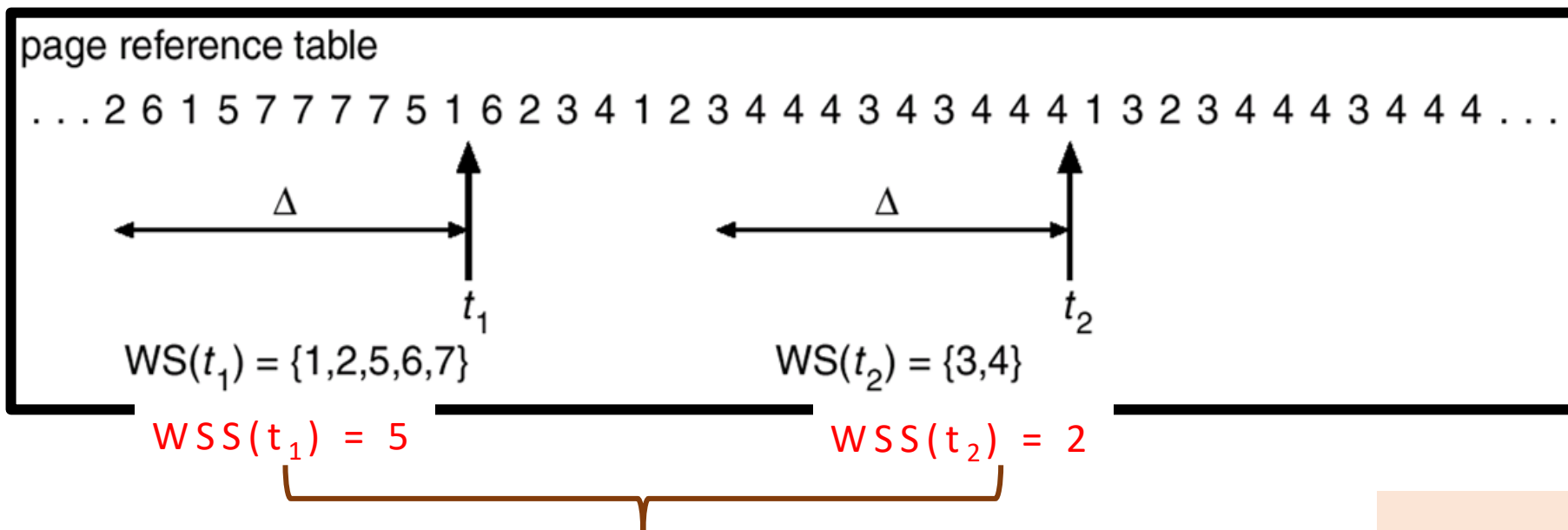
Nhận xét:

- Δ quá nhỏ \Rightarrow không đủ bao phủ toàn bộ locality
- Δ quá lớn \Rightarrow bao phủ nhiều locality khác nhau
- $\Delta = \infty \Rightarrow$ bao gồm tất cả các trang được sử dụng

Vấn đề thrashing

Giải pháp tập làm việc

Ví dụ: $\Delta = 10$



$D = \sum WSS_i =$ tổng các working-set size

$D > \text{số frame của hệ thống} \Rightarrow \text{thrashing}$

Loại trừ được trì trệ, vẫn đảm bảo đa chương



Giải pháp

- ✓ Khi khởi tạo: cung cấp số frame thoả mãn WSS của nó
- ✓ Nếu $D > \text{số frame hệ thống} \Rightarrow$ tạm dừng 1 process
- ✓ Chuyển trang sang đĩa cứng, thu hồi các frame



Wanna Play?



Củng cố kiến thức

THANK
THANK YOU |