

BAN HỌC TẬP KHOA CÔNG NGHỆ PHẦN MỀM

CHUỖI TRAINING CUỐI HỌC KÌ I NĂM HỌC 2021 - 2022



Sharing is learning



Ban học tập


Khoa Công Nghệ Phần Mềm
Trường ĐH Công Nghệ Thông Tin
ĐHQG Hồ Chí Minh



Email / Group

bht.cnpm.uit@gmail.com
fb.com/groups/bht.cnpm.uit
<https://www.facebook.com/bhtcnpm>

Hệ điều hành

 Thời gian training: 19h30 ngày 21 – 22/12/2021

 Trainer: Chu Kim Chí – KHMT2020

Nguyễn Minh Hiếu – KTPM2020

Đoàn Ngọc Như Quỳnh – KHCL2020.1

Nội dung Training



Sharing is learning

I. Đồng bộ

II. Deadlocks

III. Quản lý bộ nhớ

IV. Bộ nhớ ảo



Sharing is learning

1. Đồng bộ (Synchronization)

Đồng bộ



Sharing is learning

- 1. Tổng quan về đồng bộ**
- 2. Các giải pháp Busy Waiting**
- 3. Các giải pháp Sleep & Wakeup**

Tổng quan về đồng bộ



1.1. Giới thiệu về điều kiện tranh chấp (race condition)

- Một số process thực thi đồng thời và chia sẻ dữ liệu (qua shared memory, file).
- Không kiểm soát khi các tiến trình đồng thời truy cập dữ liệu chia sẻ
=> **Không nhất quán dữ liệu (data inconsistency).**
- Cần có cơ chế để đảm bảo thực thi có trật tự của các process đồng thời.

Tổng quan về đồng bộ



Sharing is learning

Bài toán ví dụ:

count = 0

- Cả 2 process thực thi đồng thời.
- Increase không thể thực thi khi **count** = MAX_SIZE.
- Decrease không thể thực thi khi **count** = 0.

INCREASE

```
while(1)
{
    while (count == MAX_SIZE);
    count++;
}
```

DECREASE

```
while(1)
{
    while (count == 0);
    count--;
}
```

Tổng quan về đồng bộ



Sharing is learning

Mã máy tương ứng:

count++;

- register1 = count
- register1 = register1 + 1
- count = register1

count--;

- register2 = count
- register2 = register2 - 1
- count = register2

register là các thanh ghi của CPU.

Tổng quan về đồng bộ



Sharing is learning

Giả sử $\text{count} = 3$, quantum time = 2.

1: INCREASE	$\text{register1} := \text{count}$	$\text{register1} = 3, \text{count} = 3$
2: INCREASE	$\text{register1} := \text{register1} + 1$	$\text{register1} = 4, \text{count} = 3$
3: DECREASE	$\text{register2} := \text{count}$	$\text{register2} = 3, \text{count} = 3$
4: DECREASE	$\text{register2} := \text{register2} - 1$	$\text{register2} = 2, \text{count} = 3$
5: INCREASE	$\text{count} := \text{register1}$	$\text{count} = 4$
6: DECREASE	$\text{count} := \text{register2}$	$\text{count} = 2$

Kết quả biến count phụ thuộc vào thứ tự dòng lệnh.

Ta mong đợi sau khi INCREASE, $\text{count} = 4$, sau khi DECREASE, **$\text{count} = 3$**

Tổng quan về đồng bộ



Sharing is learning

Giả sử $\text{count} = 3$, quantum time = 3.

1: INCREASE	$\text{register1} := \text{count}$	$\text{register1} = 3, \text{count} = 3$
2: INCREASE	$\text{register1} := \text{register1} + 1$	$\text{register1} = 4, \text{count} = 3$
3: INCREASE	$\text{count} := \text{register1}$	$\text{count} = 4$
4: DECREASE	$\text{register2} := \text{count}$	$\text{register2} = 4, \text{count} = 4$
5: DECREASE	$\text{register2} := \text{register2} - 1$	$\text{register2} = 3, \text{count} = 4$
6: DECREASE	$\text{count} := \text{register2}$	$\text{count} = 3$

- Cần phải có giải pháp để các lệnh $\text{count}++$, $\text{count}--$ phải là **đơn nguyên (atomic)**, nghĩa là thực hiện **như một lệnh đơn, không bị ngắt nửa chừng**.

1.2. Vùng tranh chấp (Critical Section – CS, miền găng)

- CS là **đoạn mã** chứa các **thao tác lên dữ liệu chia sẻ** trong mỗi tiến trình.
- Ở ví dụ trước biến **count** chính là một Critical Section.
- Cấu trúc của mỗi process truy cập vào CS có đoạn code như sau:

```
Do {  
    entry section          /* vào critical section */  
        critical section          /* truy xuất dữ liệu chia sẻ */  
    exit section           /* rời critical section */  
        remainder section        /* làm những việc khác */  
} While (1)
```

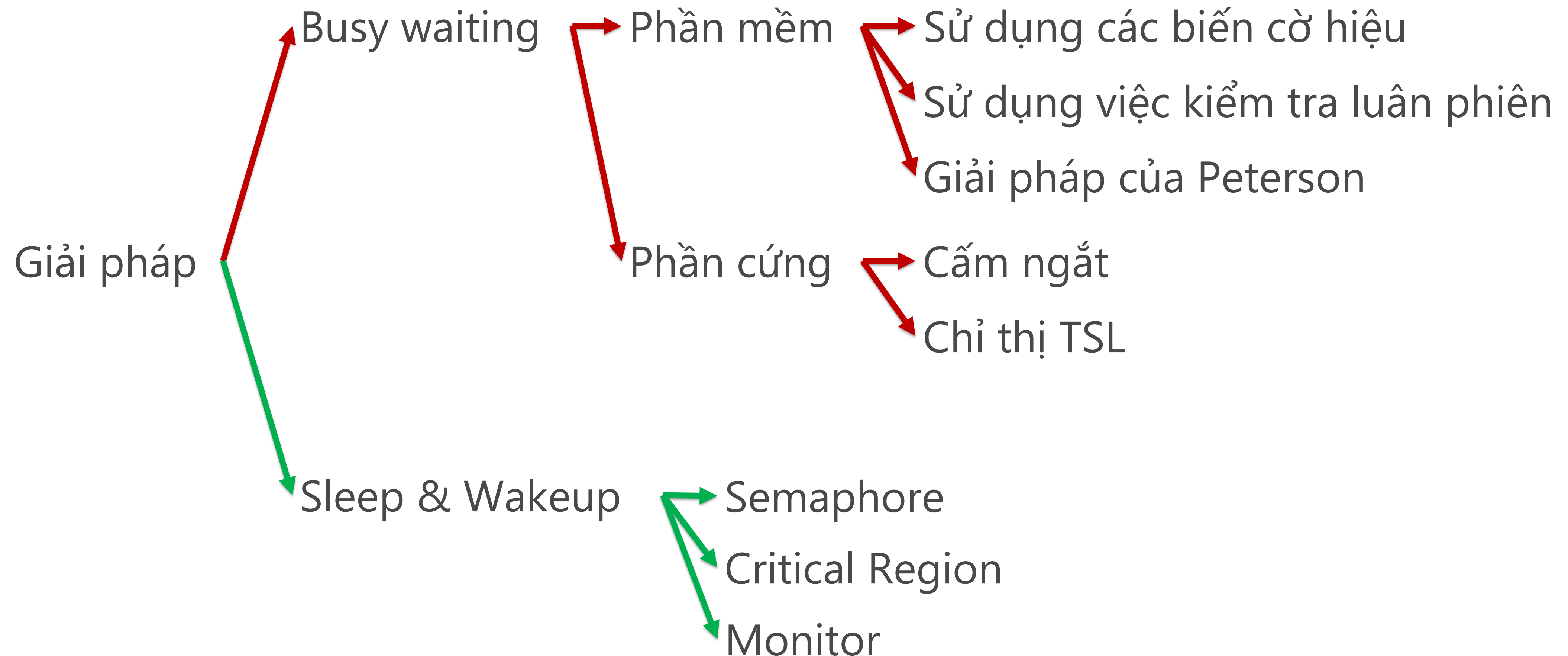
- Cần lời giải cho vấn đề CS

1.3. Yêu cầu của lời giải cho CS Problem

Lời giải phải thỏa ba tính chất:

1. Loại trừ tương hỗ (Mutual exclusion - Mutex): 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).

1.4. Phân nhóm các giải pháp



1.4.1. Các giải pháp Busy Waiting

- Tiếp tục tiêu thụ CPU trong khi chờ đợi vào vùng tranh chấp.
- Không đòi hỏi sự trợ giúp của hệ điều hành.

while (chưa có quyền) do_nothing();

CS;

Từ bỏ quyền sử dụng CS;

Tổng quan về đồng bộ



Sharing is learning

1.4.2. Các giải pháp “Sleep & Wake up”

- Tắt CPU khi chưa được vào vùng tranh chấp.
- Cần hệ điều hành hỗ trợ.

if (chưa có quyền) sleep();

CS;

Wakeup(somebody);

Tổng quan về đồng bộ



Keywords

- Đồng bộ: synchronization.
- Dữ liệu không nhất quán: data inconsistency.
- Đơn nguyên: atomic.
- Vùng tranh chấp: critical section.
- Loại trừ tương hỗ: mutual exclusion.
- Chờ đợi giới hạn: bounded waiting.

Các giải pháp Busy Waiting



Sharing is learning

2.1 Giải thuật kiểm tra luân phiên (Strict Alternation)

Đặc điểm:

Biến chia sẻ: `int turn; /* khởi đầu turn = 0 */`

Nếu `turn = i` thì P_i được phép vào CS, với $i = 0$ hay 1

Các giải pháp Busy Waiting



Sharing is learning

2.1 Giải thuật kiểm tra luân phiên

P0

```
do
{
    while (turn != 0);
        CS;
    turn = 1;
        RS;
} while(1);
```

P1

```
do
{
    while (turn != 1);
        CS;
    turn = 0;
        RS;
} while(1);
```

- **Thỏa Mutex** vì tại 1 thời điểm, biến turn chỉ có thể mang giá trị 0 hoặc 1.
- **Không thỏa Progress** và **Bounded waiting** vì tính chất "strict alternation" của giải thuật.

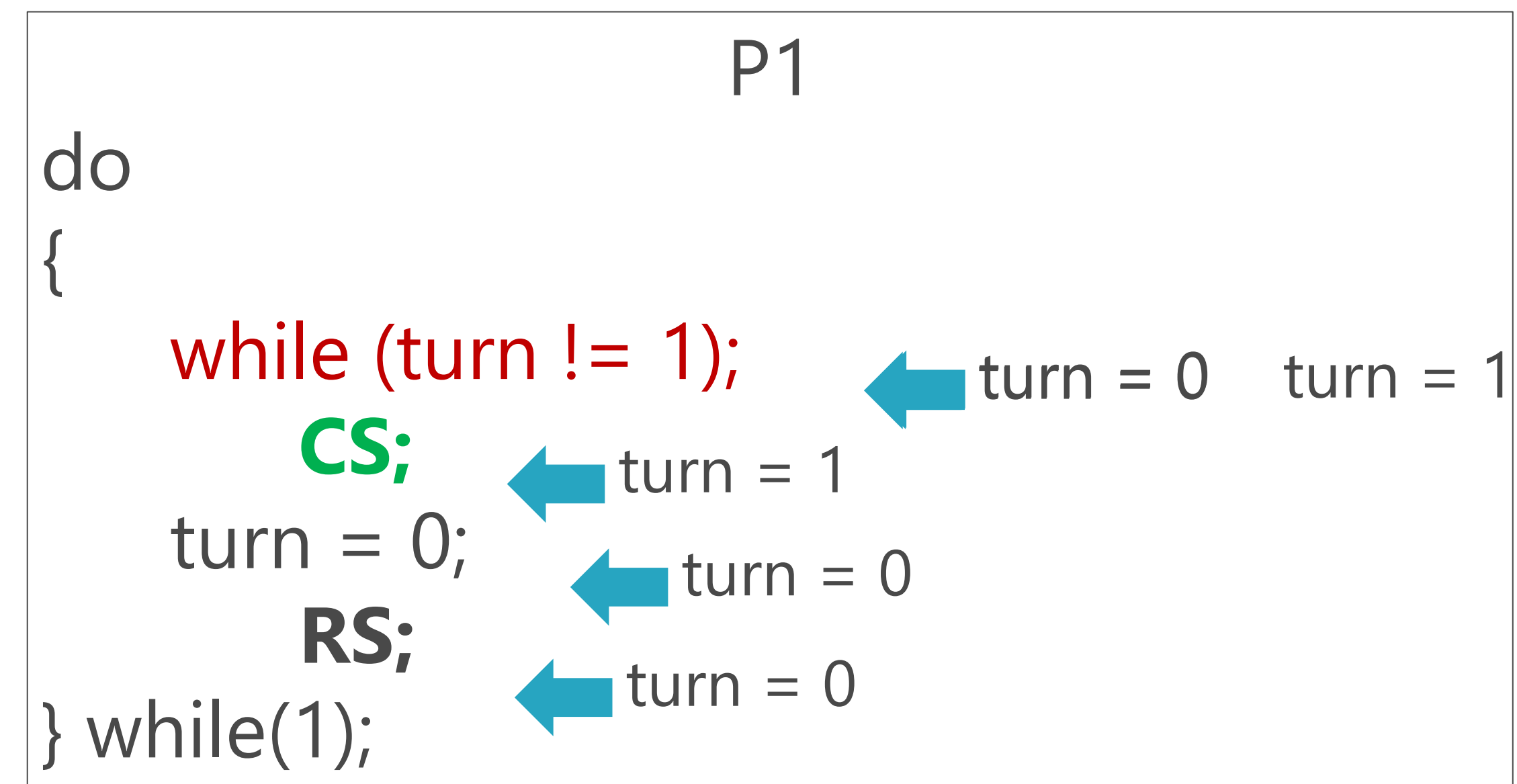
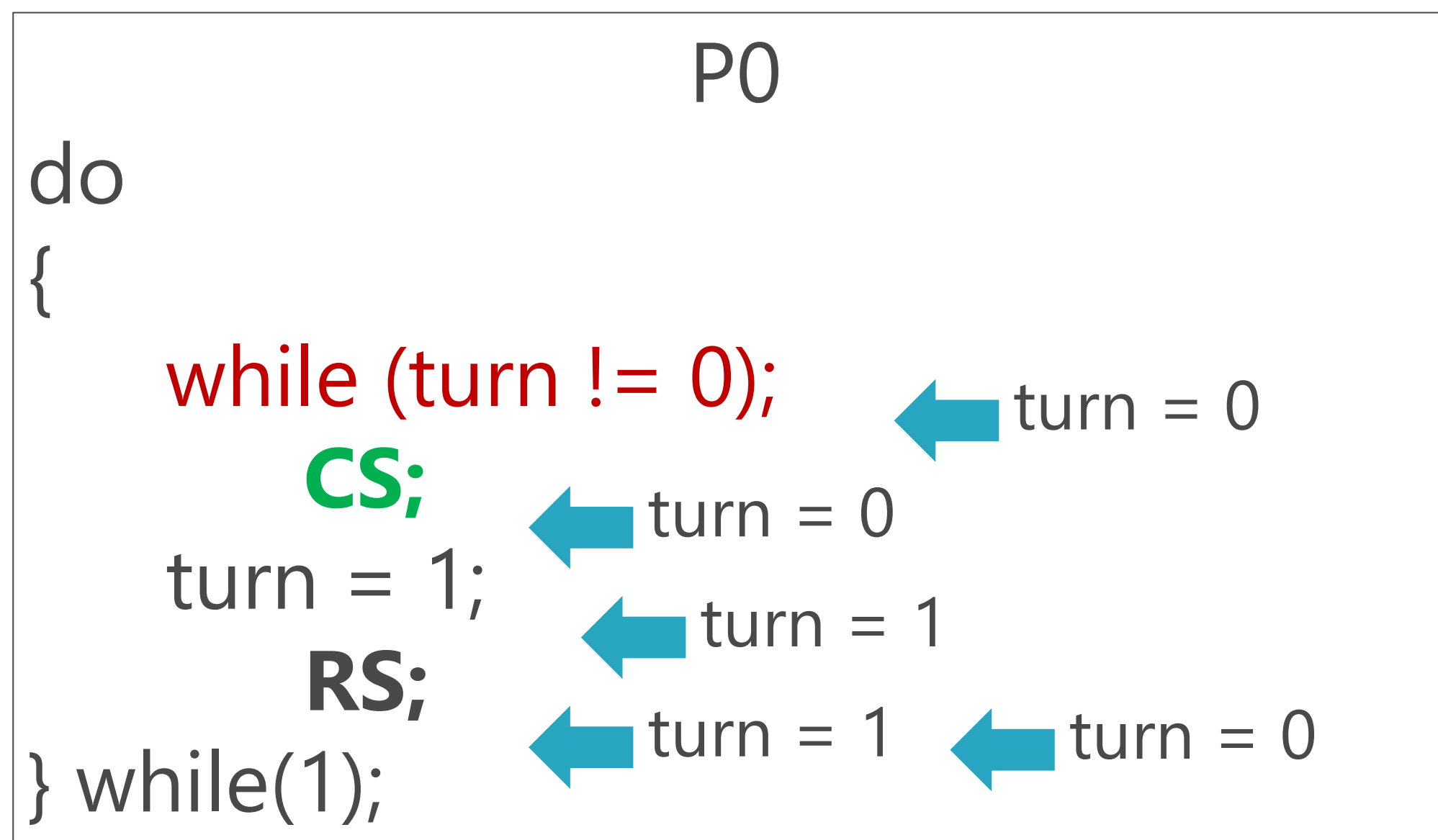
Các giải pháp Busy Waiting



Sharing is learning

2.1 Giải thuật kiểm tra luân phiên

- Giả sử ban đầu $turn = 0$ và P0 có RS lớn hơn nhiều so với P1.



P0 đang ở ngoài CS (trong RS) nhưng vẫn ngăn cản P1 vào CS, và nếu P0 vẫn còn trong RS thì P1 sẽ không thể vào CS được.

Các giải pháp Busy Waiting



Sharing is learning

2.2 Sử dụng biến cờ hiệu

Đặc điểm:

- boolean flag[2]; /* khởi đầu flag[0] = flag[1] = false */
- Nếu flag[i] = true thì P_i "sẵn sàng" vào CS.

```

                                Pi
do {
    flag[ i ] = true;        /* Pi "sẵn sàng" vào CS */
    while (flag[ j ]); /* Pi "nhường" Pj */
    CS;
    flag[i] = false;
    RS;
} while(1);
```


Các giải pháp Busy Waiting



Sharing is learning

2.2 Sử dụng biến cờ hiệu

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);
```

Các giải pháp Busy Waiting



Sharing is learning

2.2 Sử dụng biến cờ hiệu

Thoả Mutex. Vì khi P0 đang ở trong CS ($\text{flag}[0] = \text{true}$) thì P1 không thể vào CS được và nếu P1 muốn vào CS thì $\text{flag}[0]$ phải = false.
=> Điều này không thể xảy ra.

Không thoả Progress, Bounded waiting. Vì sao?

Xét TH cả 2 process cùng đang thực hiện việc gán biến cờ hiệu tương ứng của chúng = true: $\text{flag}[0] = \text{flag}[1] = \text{true}$.

2 tiến trình sẽ mắc trong 1 vòng lặp vô hạn => Cả 2 tiến trình nằm ngoài vùng tranh chấp nhưng chúng lại cản trở lẫn nhau.

Các giải pháp Busy Waiting



Sharing is learning

2.3 Giải thuật Peterson

- Kết hợp cả giải thuật kiểm tra luân phiên và sử dụng biến cờ hiệu.
- Biến chia sẻ: turn và flag[2]

P0

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

P1

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

Các giải pháp Busy Waiting



Sharing is learning

2.3 Giải thuật Peterson (cho 2 tiến trình)

- Trước khi vào CS, có thể $\text{flag}[0] = \text{flag}[1] = \text{true}$. Nhưng tại 1 thời điểm turn chỉ có thể là 0 hoặc 1. Do đó **thỏa Mutex**.
- **Khi P0 nằm ngoài CS, thì P0 có chặn P1 vào CS không?**
 - TH1:** Trước khi P0 vào CS:
P0 bị chặn bởi while(). Vì lúc này $\text{flag}[1] = \text{true}$ và $\text{turn} = 1 \Rightarrow$ P1 vào được CS.
 - TH2:** P0 vừa ra khỏi CS:
 $\text{flag}[0] = \text{false} \Rightarrow$ P1 vào được CS. **Thỏa Progress**.
- **Giả sử P0 đang chờ để vào CS. Thời gian P0 chờ tối đa?**
P0 kẹt ở while(). P0 sẽ chờ đến khi 1 trong 2 điều kiện:
 - + **ĐK1:** $\text{flag}[1] = \text{false}$:
Sau khi P1 ra khỏi CS \Rightarrow P1 gán $\text{flag}[1] = \text{false}$.
 - + **ĐK2:** $\text{turn} = 0$:
Khi P1 chuẩn bị vào CS lần nữa.Vậy P0 chờ lâu nhất là sau khi P1 vào CS được 1 lần. **Thỏa Bounded waiting**.

Các giải pháp Busy Waiting



Sharing is learning

2.4 Giải thuật Bakery

Đặc điểm:

- 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:
Tiến trình nào nhận trước thì vào trước.
- 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
VD: 1, 2, 3, 3, 3, 3, 4, 5,...
- Kí hiệu:
 $(a,b) < (c,d)$ nếu $a < c$ hoặc nếu $a = c$ và $b < d$
 $\max(a_0, \dots, a_k)$ là con số b sao cho $b \geq a_i$ với mọi $i = 0, \dots, k$

Các giải pháp Busy Waiting



Sharing is learning

2.4 Giải thuật Bakery (tham khảo thêm)

```
boolean    choosing[ n ]; /* Khởi tạo, choosing[ l ] = false */
int        num[ n ]; /* Khởi tạo, num[ l ] = 0 */

do {
    choosing[ l ] = true;
    num[ i ] = max(num[0], num[1], ..., num[n - 1]) + 1; /* Pi càng vào sau thì num[i] càng lớn */
    choosing[ i ] = false;
    for (j = 0; j < n; j++)
    {
        while (choosing[ j ]); /* Pj chưa gán num[ j ] thì chờ Pj gán xong */
        while ((num[ j ] != 0) && (num[ j ], j) < (num[ i ], i));
    }
    critical section
    num[ i ] = 0;
    remainder section
} while (1);
```

Các giải pháp Busy Waiting



Khuyết điểm của các giải pháp phần mềm:

- Tiêu tốn CPU trong khi process đợi để vào CS (liên tục kiểm tra điều kiện).
- Nếu 1 process thực thi lâu trong CS thì 1 giải pháp hiệu quả nên có cơ chế block các process cần đợi.

Tiếp theo ta sẽ tìm hiểu về các giải pháp phần cứng.

Cấm ngắt (disable interrupts)

Dùng các lệnh đặc biệt

Các giải pháp Busy Waiting



Sharing is learning

2.4 Cấm ngắt

P0

```
do
{
    disable_interrupts();
    CS;
    enable_interrupts();
    RS;
} while(1);
```

P1

```
do
{
    disable_interrupts();
    CS;
    enable_interrupts();
    RS;
} while(1);
```

2.4 Cấm ngắt (Busy waiting)

Trong hệ thống uniprocessor:

- Mutex được đảm bảo.
- Gây ảnh hưởng đến các thành phần khác của hệ thống có sử dụng ngắt như system clock (xem thêm tại <https://tinyurl.com/stclockInterrupt>).
- Cần phải liên tục tạm dừng và phục hồi ngắt dẫn đến hệ thống tốn chi phí quản lý và kiểm soát.

Trong hệ thống multiprocessor:

- Mutex không được đảm bảo.
- Chỉ cấm ngắt tại CPU thực thi lệnh `disable_interrupts`.
- Các CPU khác vẫn có thể truy cập bộ nhớ chia sẻ.

Các giải pháp Busy Waiting



Sharing is learning

2.5 Chỉ thị TSL (Test and Set Lock)

Đặc điểm:

- Đọc và ghi một biến trong 1 thao tác **atomic** (không chia cắt được)
- Biến **Lock** được chia sẻ và khởi tạo **Lock = false**;

```
boolean TestAndSet(boolean *lock) {  
    boolean returnValue = *lock;  
    *lock = true;  
    return returnValue;  
}
```

```
do  
{  
    while(TestAndSet(&Lock))  
        CS;  
    Lock = false;  
    RS;  
} while(1);
```

Các giải pháp Busy Waiting



2.5 Chỉ thị TSL

- Khi **Lock** = false bất kỳ tiến trình nào đều có thể vào CS;
1 tiến trình vào được CS sẽ set **Lock** = true nên các tiến trình khác không vào được. **Thỏa Mutex**.
- Khi P_i ra khỏi CS thì các tiến trình khác được chọn tùy ý để vào CS. **Không thỏa Bounded waiting** => Một số tiến trình có thể bị đói.
- Các processor (Pentium) cung cấp lệnh đơn Swap(a, b) có tác dụng hoán đổi giá trị a và b và có ưu nhược điểm tương tự như TestAndSet.

Các giải pháp Busy Waiting



2.5 Chỉ thị TSL thỏa mãn 3 yêu cầu (tham khảo thêm)

- Cấu trúc dữ liệu dùng chung:
 bool waiting[n];
 bool lock;
- Tiến trình chờ đợi lâu nhất sau 1 lần tất cả các tiến trình khác vào CS.

Các giải pháp Busy Waiting



Chọn phát biểu SAI trong các phát biểu dưới đây?

- A. Giải thuật Peterson và giải thuật Bakery là các giải pháp đồng bộ busy waiting sử dụng phần mềm.
- ☒ B. Cấm ngắt là giải pháp đồng bộ busy waiting luôn đảm bảo tính chất loại trừ tương hỗ.
- C. Trong giải thuật Bakery, trước khi vào vùng tranh chấp, mỗi tiến trình sẽ được nhận một con số.
- D. Trong giải thuật Peterson, tính chất chờ đợi giới hạn luôn được đảm bảo.

Giải thích: Cấm ngắt không đảm bảo được tính chất loại trừ tương hỗ trên các hệ thống đa bộ xử lý.

Các giải pháp Busy Waiting



Sharing is learning

Kiến thức cần nhớ

- **Giải thuật kiểm tra luân phiên:**

Thoả Mutex. Không thoả Progress, Bounded waiting.

- **Sử dụng biến cờ hiệu:**

Thoả Mutex. Không thoả Progress.

- **Giải thuật Peterson:**

Kết hợp cả 2 giải thuật trên.

Áp dụng cho 2 tiến trình.

Thoả cả 3 yêu cầu.

Thời gian 1 tiến trình phải chờ lâu nhất để đi vào CS là sau 1 lần tiến trình kia vào CS.

Các giải pháp Busy Waiting



Sharing is learning

Kiến thức cần nhớ

– Giải thuật Bakery:

Là giải thuật Peterson áp dụng cho n tiến trình.

Trước khi vào CS, mỗi tiến trình sẽ được nhận một con số.

Tiến trình giữ con số nhỏ nhất sẽ được vào CS.

Thoả cả 3 yêu cầu.

– Cấm ngắt:

Không đảm bảo Mutex trên các hệ thống đa bộ xử lý.

Gây ảnh hưởng đến các hoạt động khác có sử dụng ngắt.

Giảm hiệu quả hoạt động của các tiến trình do việc cấm ngắt và phục hồi ngắt thực hiện liên tục.

Các giải pháp Busy Waiting



Kiến thức cần nhớ

- **Chỉ thị TSL:**

Đọc và ghi 1 biến trong 1 thao tác không chia cắt được.

Thoả Mutex, không thoả Bounded waiting (đối với TSL đơn thuần).

Có thể gây ra tình trạng starvation.

- **Với chỉ thị TSL thoả mãn 3 yêu cầu:**

Tiến trình chờ đợi lâu nhất sau 1 lần tất cả các tiến trình khác vào CS.

- **Nhược điểm của các giải pháp đồng bộ Busy waiting sử dụng phần mềm:**

Tốn nhiều thời gian xử lý của CPU khi kiểm tra điều kiện để cho phép tiến trình vào CS.

Các giải pháp Sleep & Wakeup



Sharing is learning

Ý tưởng

int **busy**; // = 1 nếu CS đang bị chiếm.

int **blocked** //thể hiện 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);
```

3.1 Semaphore

Đặc điểm:

- Là công cụ đồng bộ cung cấp bởi OS mà không đòi hỏi busy waiting.
- Là một biến số nguyên.
- Chỉ có thể được truy xuất qua hai tác vụ có tính đơn nguyên (atomic) và loại trừ (mutual exclusive).
 - wait(S) ~ P(S): giành tài nguyên và $S.value--$. Nếu $S.value < 0$ thì process bị blocked.
 - signal(S) ~ V(S): trả tài nguyên và $S.value++$. Nếu $S.value \leq 0$ thì 1 process đang blocked bởi 1 lệnh wait() sẽ được wakeup() và thực thi – **FIFO** (ở đầu hàng đợi).
- Nếu P(S) được thực hiện trên biến đếm ≤ 0 , tiến trình phải đợi V(S) (hay chờ đợi sự giải phóng tài nguyên)

Các loại semaphore:

Counting semaphore: một số nguyên có giá trị không hạn chế.

Binary semaphore: có trị là 0 hay 1.

Có thể hiện thực counting semaphore bằng binary semaphore.

Các giải pháp Sleep & Wakeup



Sharing is learning

3.1 Semaphore

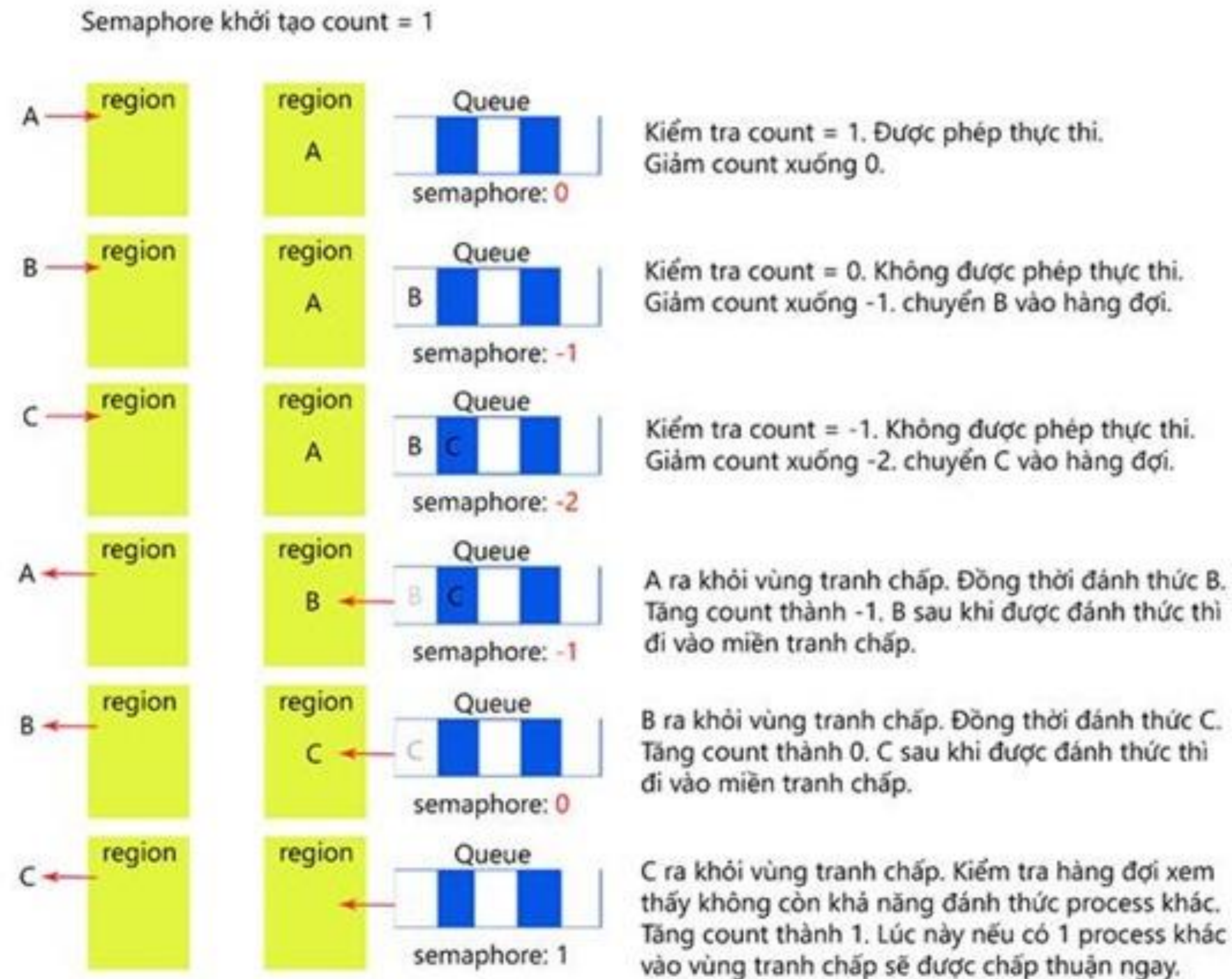


Figure: Counting Semaphore (non priority)

Các giải pháp Sleep & Wakeup



Sharing is learning

3.1 Semaphore

Hiện thực Semaphore: Định nghĩa semaphore là 1 record.

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

- Biến value: thể hiện số tiến trình có thể vào CS (hoặc tài nguyên) một lúc.
- *L là DSLK các PCB (Process Control Block) (Xem thêm về PCB).

Giả sử OS cung cấp 2 tác vụ (system call):

block(): tạm treo process nào thực thi lệnh này (running -> waiting).

wakeup(P): hồi phục quá trình thực thi của process P đang blocked (waiting -> ready).

Các giải pháp Sleep & Wakeup



Sharing is learning

3.1 Semaphore

Các tác vụ semaphore được hiện thực như sau:

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

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

- Nhiều process thực thi đồng thời => Hai đoạn mã trên là CS.

Các giải pháp Sleep & Wakeup



Sharing is learning

3.1 Semaphore

VD sử dụng Semaphore 1:

Yêu cầu:

Dùng cho n process

Chỉ duy nhất 1 process được vào CS (mutual exclusion)

Khởi tạo S.Value = 1

Để cho phép k process vào CS,

khởi tạo S.value = k

Shared data:
semaphore mutex;
/* initially mutex.value = 1 */

Process P_i:
do {
 wait(mutex);
 CS;
 signal(mutex);
 RS;
} while (1);

Các giải pháp Sleep & Wakeup



Sharing is learning

3.1 Semaphore

VD sử dụng semaphore 2:

- Có 2 process P1 và P2.
- Yêu cầu: lệnh S1 trong P1 cần thực thi trước lệnh S2 trong P2.
- Khởi tạo semaphore synch:
 $\text{synch.value} = 0.$

P1

S1;
signal(synch);

P2

wait(synch);
S2;

Các giải pháp Sleep & Wakeup



Sharing is learning

Bài tập

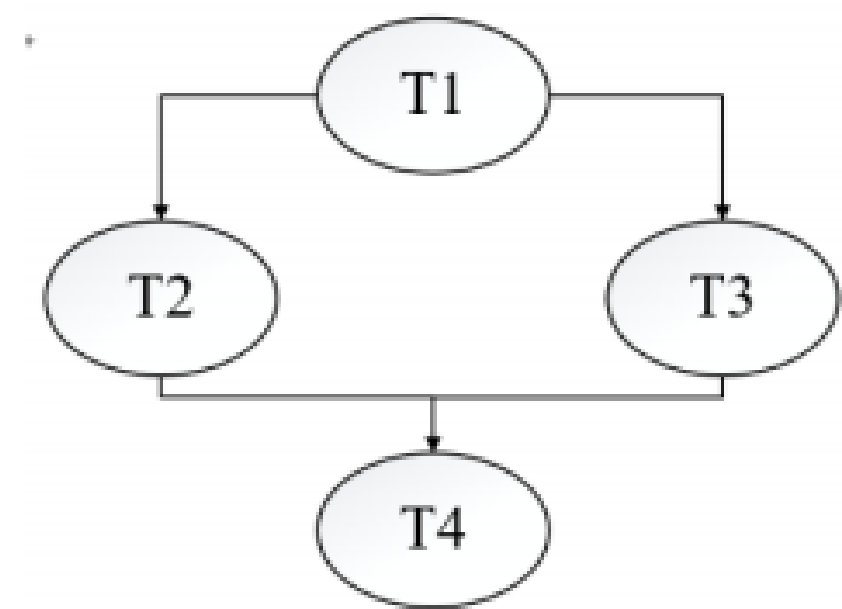
Hệ thống có 4 tiểu trình T1, T2, T3, T4. Quan hệ giữa các tiểu trình này được biểu diễn như sơ đồ, mũi tên từ Tx sang Ty nghĩa là Tx phải kết thúc trước khi Ty bắt đầu thực thi. Giả sử tất cả các tiểu trình đã được khởi tạo và sẵn sàng để thực thi. Dùng semaphore để đồng bộ hoạt động của các tiểu trình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

Điều kiện:

T1 thực thi trước T2, T3

T2, T3 thực thi trước T4

Có 2 điều kiện => dùng 2 semaphore



Khai báo và khởi tạo các semaphore:

init(sem1,0); //khởi tạo semaphore sem1 có giá trị bằng 0

init(sem2,0); //khởi tạo semaphore sem2 có giá trị bằng 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
}
```


Các giải pháp Sleep & Wakeup



3.1 Semaphore

Nhận xét:

$S.value \geq 0$: số process có thể thực thi $wait(S)$ mà không bị blocked = $S.value$

$S.value < 0$: số process đang đợi trên S là $|S.value|$

Vấn đề: Nếu tại cùng 1 thời điểm, lệnh $wait(S)$ được gọi bởi 2 tiến trình cùng 1 lúc?

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.

⇒ Đoạn mã định nghĩa các lệnh $wait(S)$ và $signal(S)$ cũng chính là CS.

Các giải pháp Sleep & Wakeup



Sharing is learning

3.1 Semaphore

Deadlock và starvation

Deadlock: hai hay nhiều process đang chờ đợi vô hạn định một sự kiện không bao giờ xảy 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.

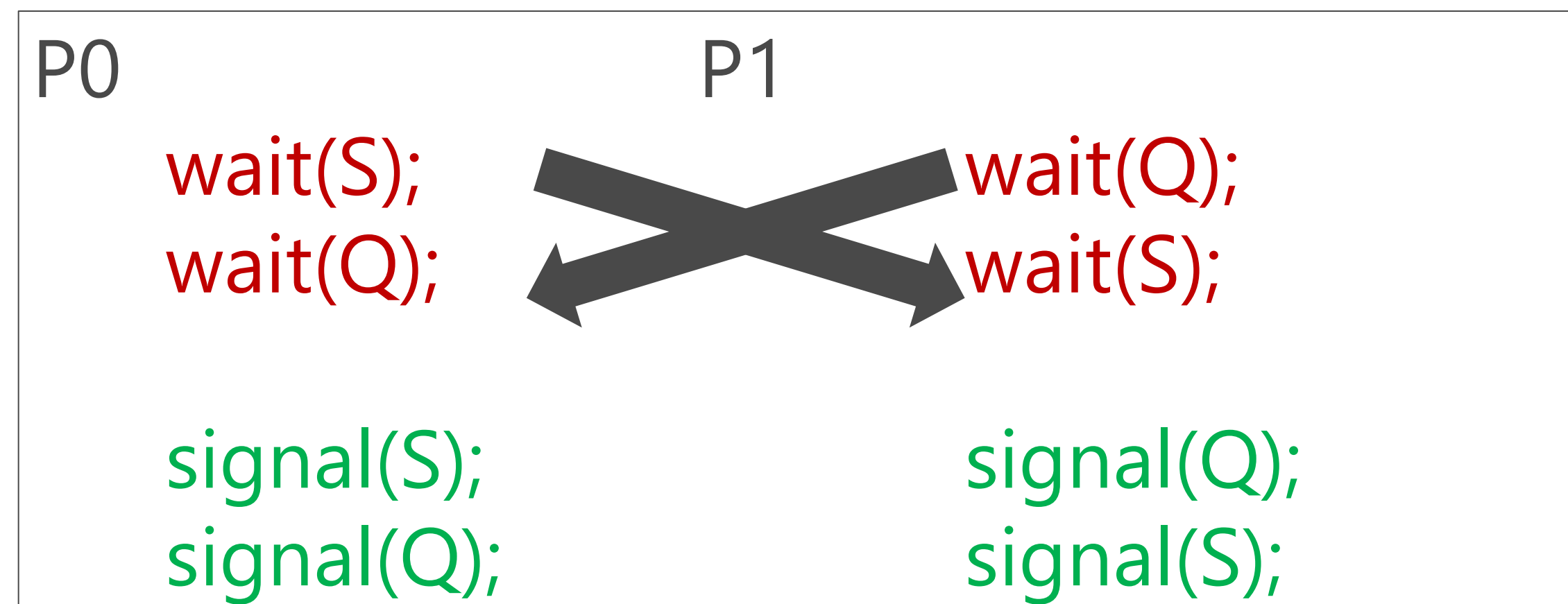
Khi số lượng Semaphore nhiều, người dùng khó có thể nắm rõ kịch bản xảy ra.

Nếu không sử dụng đúng => có thể xảy ra deadlock hoặc starvation.

– Semaphore có thể gây ra hiệu ứng đảo ngược ưu tiên.

VD: Khởi tạo 2 Semaphore S và Q với S.Value = Q.Value = 1.

P0 thực thi wait(S), rồi P1 thực thi wait(Q), rồi P0 thực thi wait(Q) bị blocked, P1 thực thi wait(S) bị blocked. => Deadlock



3.2 Critical Region

- 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.

Các giải pháp Sleep & Wakeup



Sharing is learning

3.2 CR và bài toán Bounded Buffer

Dữ liệu chia sẻ:

struct buffer

{

int pool[n];

int count, in, out;

}

Producer

```
region buffer when (count < n){  
    pool[in] = nextp;  
    in = (in + 1) % n;  
    count ++;  
}
```

Consumer

```
region buffer when (count > 0){  
    nextc = pool[out];  
    out = (out + 1) % n;  
    count--;  
}
```

Các giải pháp Sleep & Wakeup



Sharing is learning

3.3 Monitor

- Cũng là một cấu trúc ngôn ngữ cấp cao tương tự CR, có chức năng như Semaphore nhưng dễ điều khiển hơn.
- Xuất hiện trong nhiều ngôn ngữ lập trình đồng thời như: Concurrent Pascal, Modula-3, Java,...
- Có thể hiện thực bằng Semaphore.
- Là một module phần mềm, bao gồm:
 - Một hoặc nhiều thủ tục (procedure).
 - Một đoạn code khởi tạo (initialization code).
 - Các biến dữ liệu cục bộ (local data variable).

Các giải pháp Sleep & Wakeup



Sharing is learning

Đặc tính của Monitor

- Local variable chỉ có thể truy xuất bởi các thủ tục của Monitor.
- Process “vào monitor” bằng cách gọi một trong các thủ tục đó.
- Chỉ có một process có thể vào monitor tại một thời điểm

Mutex được bảo đảm.

Các giải pháp Sleep & Wakeup



Sharing is learning

Condition variable

- Nhằm cho phép một process đợi “trong monitor”, phải khai báo biến điều kiện (condition variable)
condition a, b;
- **Các biến điều kiện đều cục bộ và chỉ được truy cập bên trong monitor.**
- **Chỉ có thể thao tác lên biến điều kiện bằng hai thủ tục:**
 - **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.

Các giải pháp Sleep & Wakeup



Chọn phát biểu SAI trong các phát biểu dưới đây?

- 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.
- 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.

Các giải pháp Sleep & Wakeup



Chọn phát biểu ĐÚNG trong các phát biểu dưới đây?

- A. Lệnh wait(S) sẽ làm tăng giá trị của semaphore S thêm 1 đơn vị.
- B. Lệnh signal(S) sẽ làm giảm giá trị của semaphore S đi 1 đơn vị.
- ☒ C. Đoạn mã định nghĩa các lệnh wait(S) và signal(S) cũng là các vùng tranh chấp.
- D. Có thể hiện thực binary semaphore bằng counting semaphore.



Sharing is learning

2. Deadlocks

Deadlocks



Sharing is learning

- I. Định nghĩa**
- II. Mô hình hóa hệ thống**
- III. Các tính chất của deadlocks**
- IV. Phương pháp giải quyết deadlocks**
- V. Bài tập ôn**

1. Định nghĩa

Tình huống: Một tập các tiến trình bị block, mỗi tiến trình giữ tài nguyên và đang chờ tài nguyên mà tiến trình khác trong tập đang giữ

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

1. Định nghĩa

- Một tiến trình gọi là **deadlocks** nếu nó đang đợi một sự kiện mà sẽ không bao giờ xảy ra. Thông thường, có nhiều hơn một tiến trình bị liên quan trong deadlocks.
- Một tiến trình gọi là trì hoãn vô hạn định nếu nó bị trì hoãn một khoảng thời gian dài lặp đi lặp lại trong khi hệ thống đáp ứng cho những tiến trình khác.
- **starvation.**
 - + Ví dụ: Một tiến trình sẵn sàng để xử lý nhưng nó không bao giờ nhận được CPU.

DEADLOCKS VÀ STARVATION ???

2. Mô hình hóa hệ thống

- Các loại tài nguyên, kí hiệu R_1, R_2, \dots, R_m , bao gồm:
 - + CPU cycle, không gian bộ nhớ, thiết bị I/O, file, semaphore,...
 - + Mỗi loại tài nguyên R_i có W_i thực thể.
- Giả sử tài nguyên tái sử dụng theo chu kỳ:
 - + **Yêu cầu**: tiến trình phải chờ nếu yêu cầu không được đáp ứng ngay.
 - + **Sử dụng**: tiến trình sử dụng tài nguyên.
 - + **Hoàn trả**: tiến trình hoàn trả tài nguyên.
- Các tác vụ yêu cầu và hoàn trả đều là system call. Ví dụ:
 - + Request/ release device
 - + Open / close file
 - + Allocate/ free memory
 - + Wait/ signal

3. CÁC TÍNH CHẤT CỦA DEADLOCKS

3.1. Đồ thị cấp phát tài nguyên (RAG)

- Là đồ thị **có hướng**, với tập **đỉnh V** và tập **cạnh E**.
- Tập **đỉnh V** gồm 2 loại:
 - $P = \{P1, P2, \dots, Pn\}$ (All process).
 - $R = \{R1, R2, \dots, Rn\}$ (All resource).
- Tập **cạnh E** gồm 2 loại:
 - **Cạnh yêu cầu:** $P_i \rightarrow R_j$.
 - **Cạnh cấp phát:** $R_j \rightarrow P_i$.

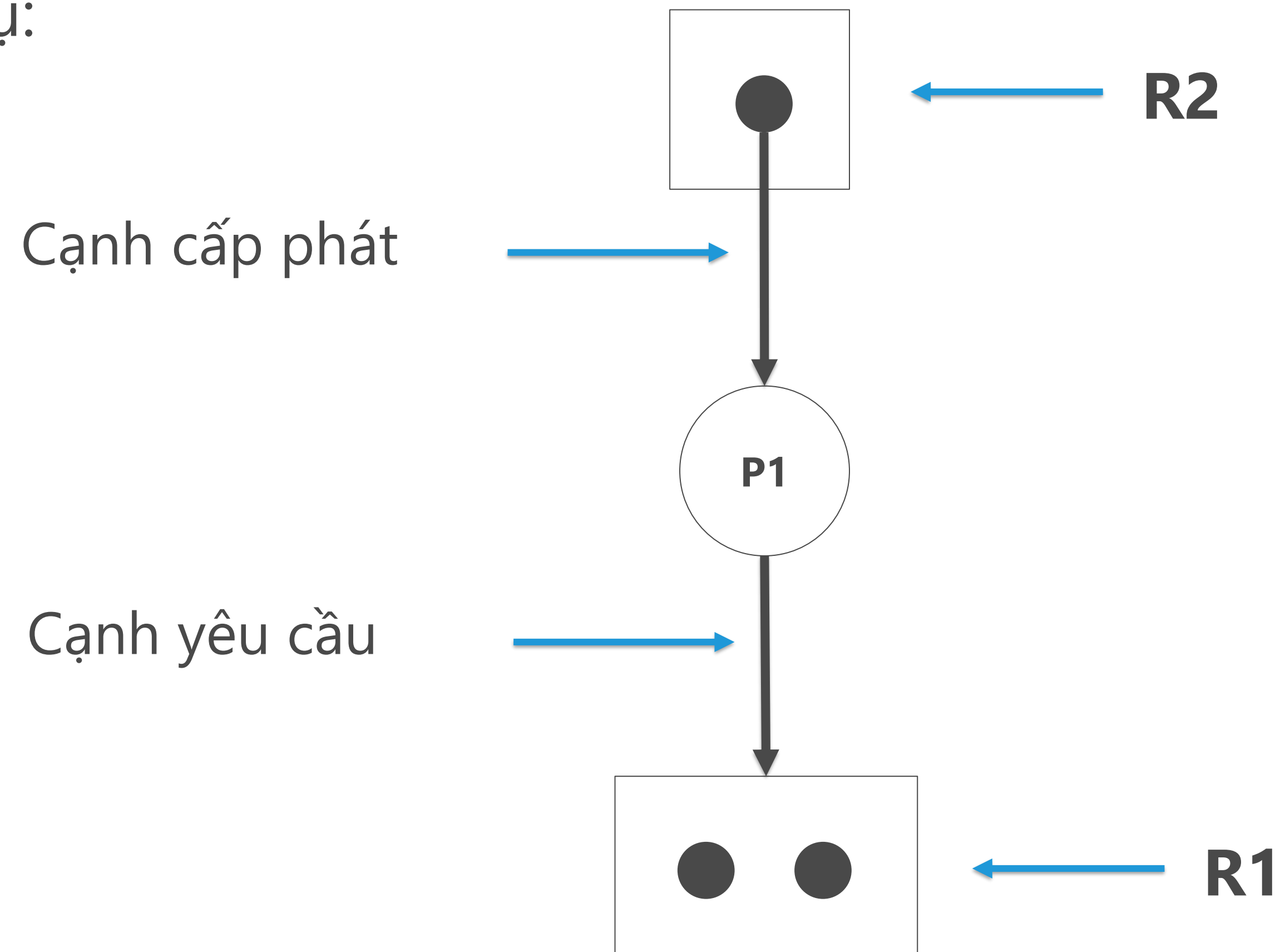
Deadlocks



Sharing is learning

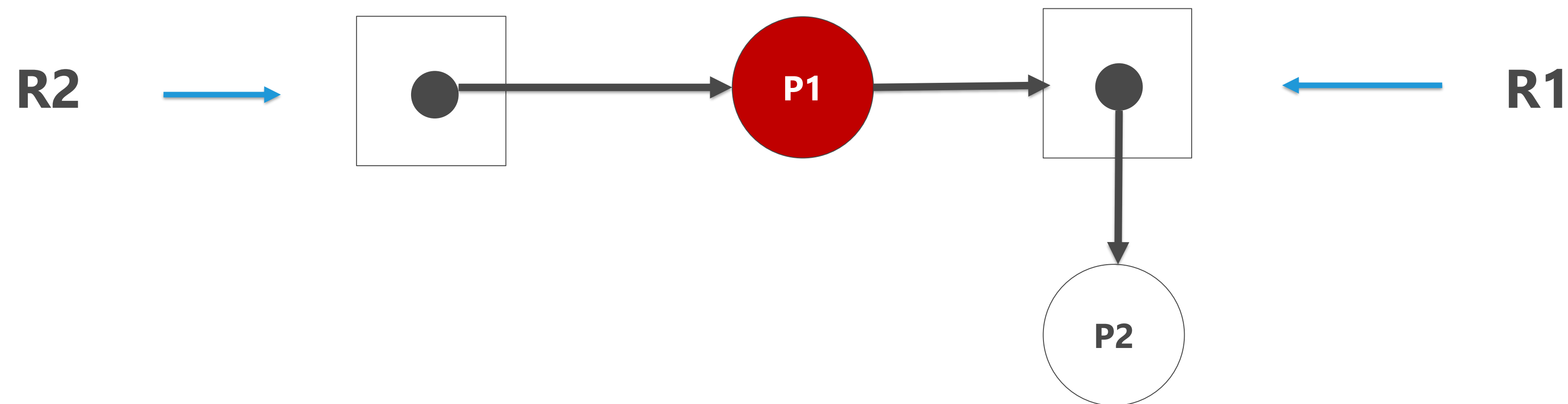
3.1. Đồ thị cấp phát tài nguyên (RAG)

– Ví dụ:



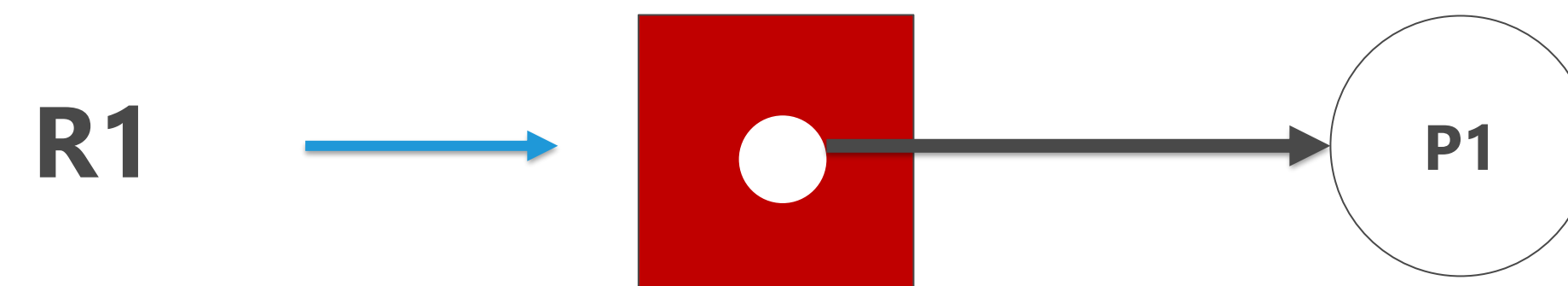
3.2. Điều kiện xảy ra deadlocks (4 điều kiện)

- **Mutex:** 1 tài nguyên chỉ có thể được giữ bởi 1 process tại 1 thời điểm.
- **Hold & Wait:** Một tiến trình đang giữ ít nhất một tài nguyên và đợi thêm tài nguyên do quá trình khác giữ.

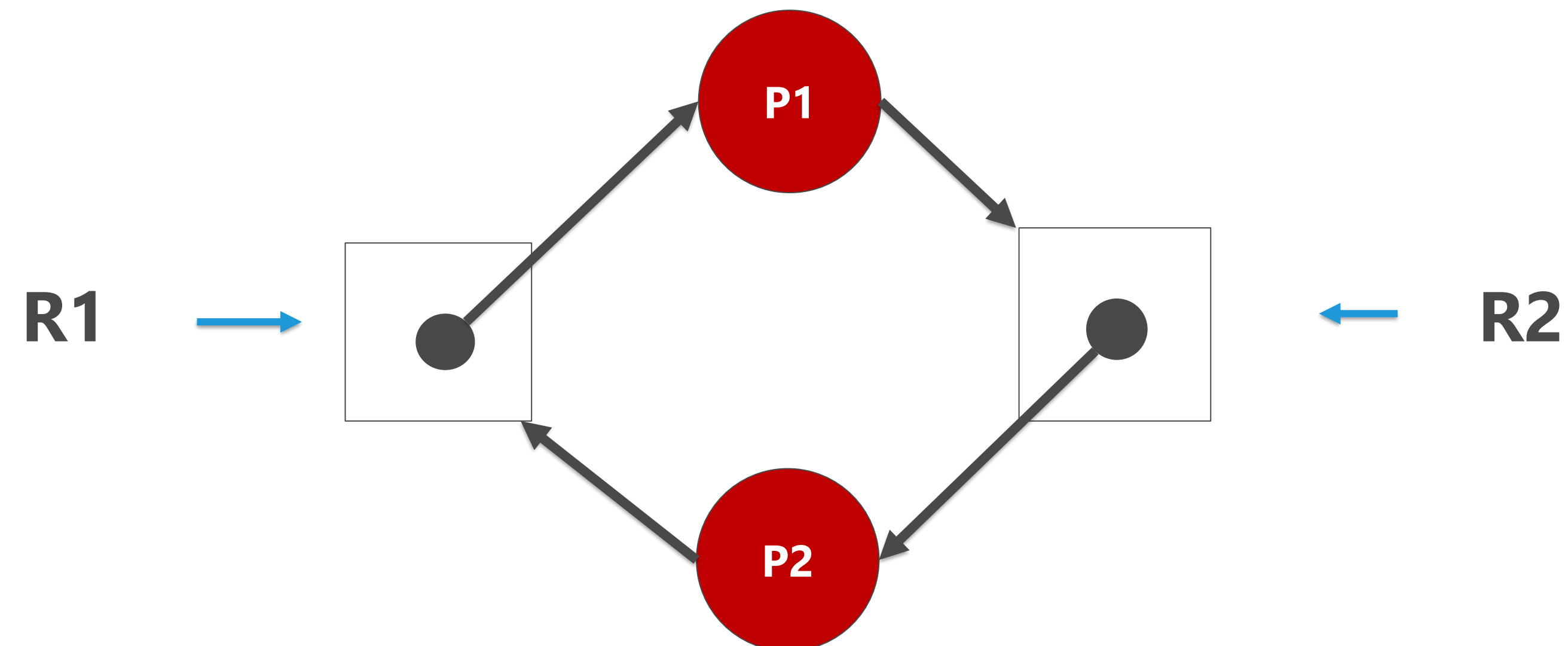


3.2. Điều kiện xảy ra deadlock

- **No Preemption:** Tài nguyên không thể lấy lại từ 1 quy trình trừ khi process đó giải phóng tài nguyên.



- **Circular Wait:** Một tập hợp các process đang chờ đợi lẫn nhau ở dạng vòng tròn (chu trình).



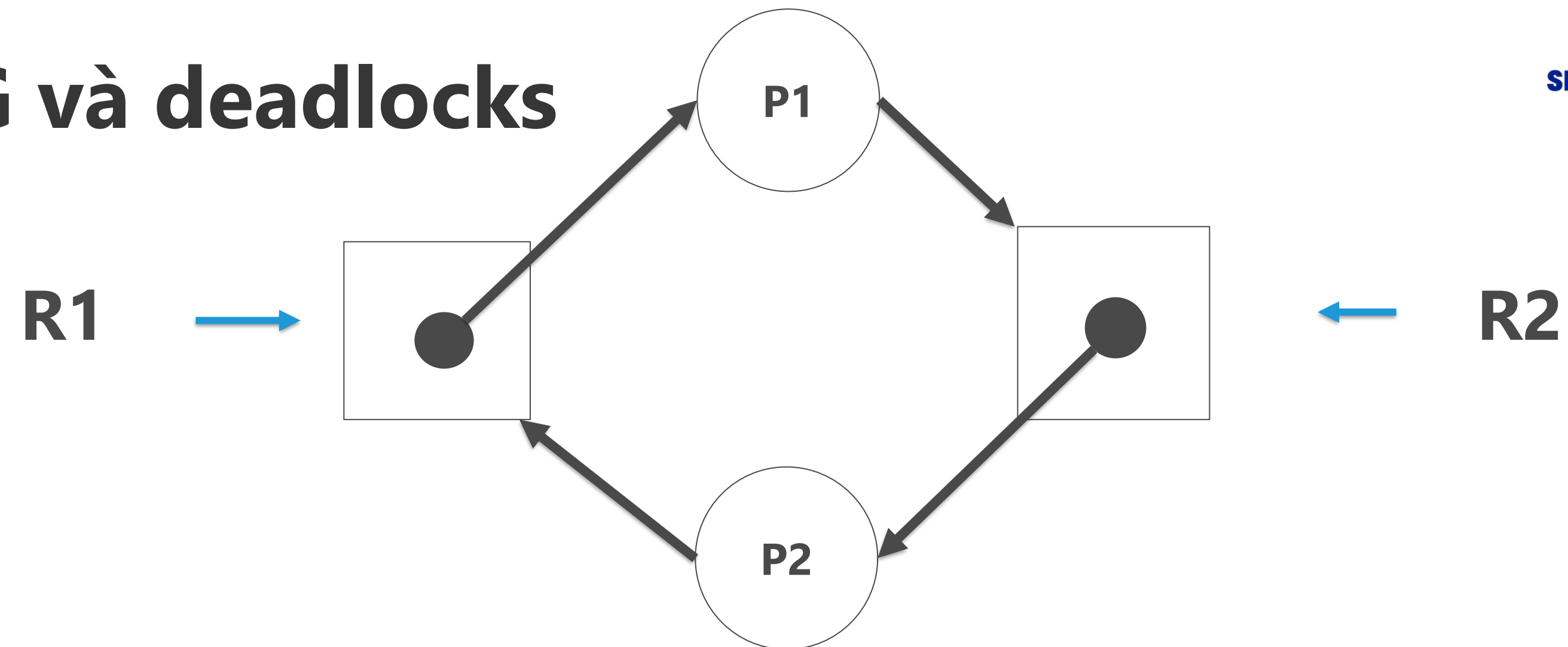
Deadlocks



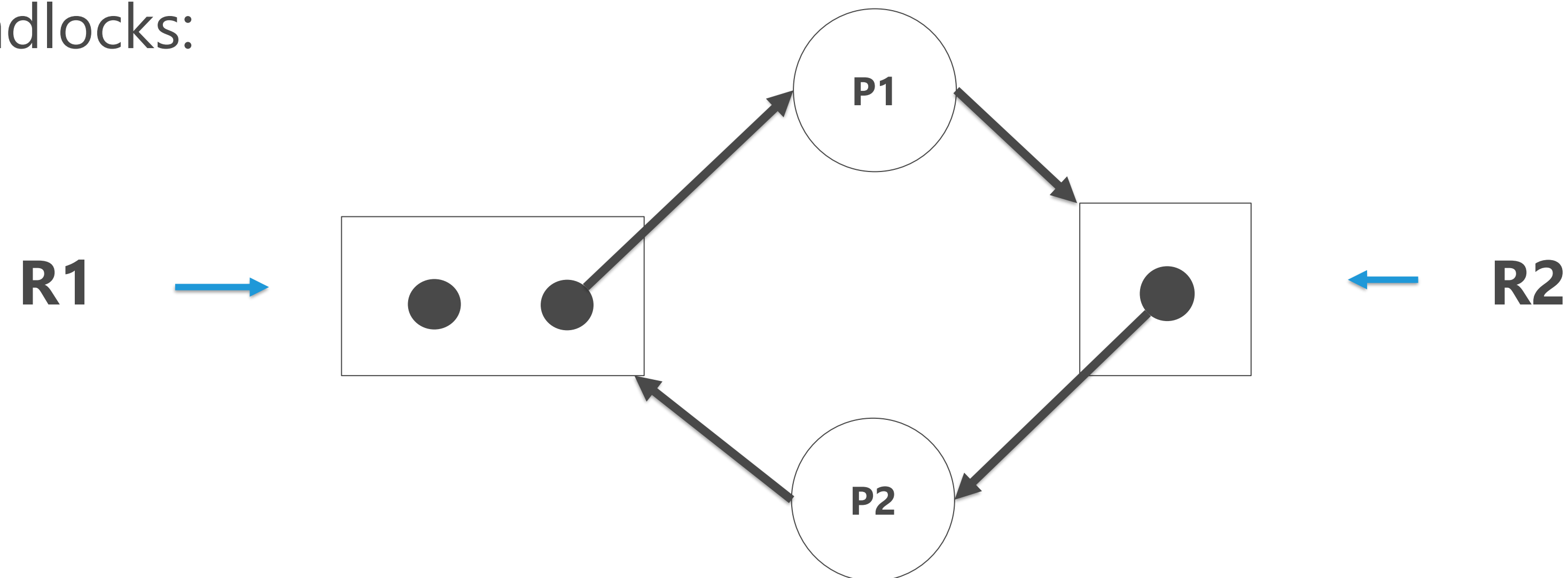
Sharing is learning

Mối liên hệ giữa RAG và deadlocks

- RAG **có** deadlocks:



- RAG **không có** deadlocks:



Mối liên hệ giữa RAG và Deadlocks

- RAG **không** chứa chu trình · **không có deadlocks**.
- RAG **chứa một (hay nhiều)** chu trình:
 - Nếu mỗi loại tài nguyên **chỉ có một thực thể** · **deadlocks**.
 - Nếu mỗi loại tài nguyên **có nhiều thực thể** · **có thể xảy ra deadlocks**.

4. Phương pháp giải quyết deadlocks

- **Ngăn chặn** deadlocks (Deadlock prevention).
- **Tránh** deadlocks (Deadlock avoidance).
- Cho phép hệ thống vào trạng thái deadlocks, nhưng sau đó **phát hiện deadlocks và phục hồi hệ thống** (Deadlocks detection and recovery).

Ngoài ra:

Bỏ qua mọi vấn đề, **xem như deadlocks không bao giờ xảy ra** trong hệ thống (Deadlocks Ignorance).

4. Phương pháp giải quyết deadlocks

- Khác biệt giữa **ngăn** deadlocks và **tránh** deadlocks:
 - **Ngăn** deadlocks: không cho phép (ít nhất) một trong 4 điều kiện cần cho deadlock.
 - **Tránh** deadlocks: các quá trình cần cung cấp thông tin về tài nguyên nó cần để hệ thống cấp phát tài nguyên một cách thích hợp.
- Bỏ qua deadlock được sử dụng nhiều nhất trong các cơ chế trên (Window và Linux), nếu xảy ra deadlocks cần khởi động lại máy.

4.1 Ngăn deadlocks

Ngăn deadlocks bằng cách ngăn chặn 1 trong 4 điều kiện cần của deadlocks:

- Ngăn **Mutex**:
 - Đối với tài nguyên không chia sẻ (printer): không làm được.
 - Đối với tài nguyên chia sẻ (read-only file): không cần thiết.

4.1 Ngăn deadlocks

- Ngăn **Hold & Wait**: **!(hold and wait) = !hold or !wait**
 - + Không giữ: khi yêu cầu tài nguyên thì process không được giữ tài nguyên nào, nếu có thì phải trả lại.
 - + Không chờ: process yêu cầu toàn bộ tài nguyên, nếu đủ HDH sẽ cấp phát, nếu không sẽ bị block.
- Không thể áp dụng trong thực tế vì process không thể xác định được tài nguyên cần thiết trước khi yêu cầu, và process có thể giữ tài nguyên trong một thời gian dài, chưa thể trả ngay.

4.1 Ngăn deadlocks

- Ngăn **No Preemption**: *khi một process đang chạy thì những tài nguyên nó nắm giữ sẽ không trưng dụng.* · Thu hồi tài nguyên.
 - + Vấn đề xảy ra là nhất quán dữ liệu · Không thực tế.
 - + Ví dụ: một máy in đang được sử dụng bởi tiến trình A, được thu hồi và cấp phát cho tiến trình B sử dụng.

4.1 Ngăn deadlocks

- Ngăn **Circular Wait**: *gán một thứ tự cho tất cả các tài nguyên trong hệ thống. Một process không thể yêu cầu một tài nguyên ít ưu tiên hơn.*
 - + Ví dụ, nếu process P1 được cấp phát tài nguyên R5, không được cấp phát tài nguyên R3. Nếu muốn được cấp phát R3 cần giải phóng R5, R4 (nếu có).
 - + Khó khăn ở việc xác định tương đối thứ tự ưu tiên của tài nguyên.
- Các phương pháp ngăn deadlocks sử dụng tài nguyên không hiệu quả, lãng phí hiệu suất thấp.

4.2 Tránh deadlocks

4.2.1. Khái quát

- Tránh deadlocks vẫn đảm bảo hiệu suất sử dụng tài nguyên tối đa đến mức có thể.
- Yêu cầu mỗi tiến trình khai báo số lượng tài nguyên tối đa cần để thực hiện công việc.
- Giải thuật tránh deadlocks sẽ kiểm tra trạng thái cấp phát tài nguyên để đảm bảo hệ thống không rơi vào deadlocks.
- Trạng thái cấp phát tài nguyên được định nghĩa dựa trên số tài nguyên còn lại, đã được cấp phát và yêu cầu tối đa của các tiến trình.

4.2 Tránh deadlocks

4.2.2. Trạng thái safe và unsafe

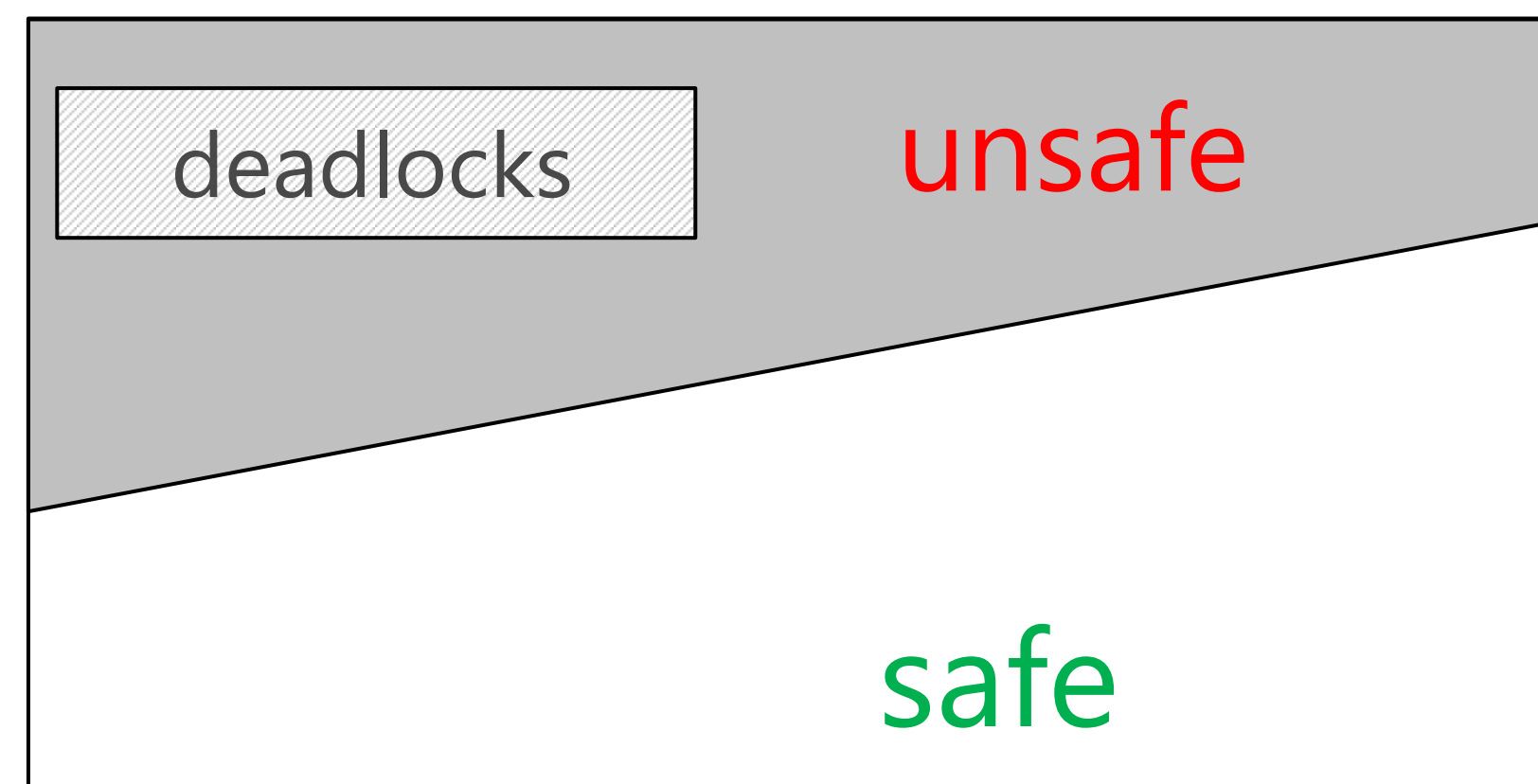
Một chuỗi quá trình $\langle P_1, P_2, \dots, P_n \rangle$ là một chuỗi an toàn nếu:

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.

4.2 Tránh deadlocks

4.2.2. Trạng thái safe và unsafe

- Một trạng thái của hệ thống được gọi là an toàn (safe) nếu tồn tại một chuỗi thứ tự an toàn. → **Không** xảy ra deadlocks.
- Một trạng thái của hệ thống được gọi là không an toàn (unsafe) nếu không tồn tại một chuỗi an toàn. → **Có thể** xảy ra deadlocks.
- Tránh deadlock bằng cách bảo đảm hệ thống không đi đến trạng thái unsafe



4.2.2. Trạng thái safe và unsafe

- Ví dụ: hệ thống có 12 tape 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 tape drive sẵn sàng.
- Chuỗi $\langle P1, P0, P2 \rangle$ là chuỗi an toàn \rightarrow hệ thống là an toàn.

4.2.2. Trạng thái safe và unsafe

- Tại T1, P2 yêu cầu và được cấp thêm 1 tape drive.

	Cần tối đa	Đang giữ	Cần thêm
P0	10	5	5
P1	4	2	2
P2	9	3	6

- Còn 2 tape drive sẵn sàng.
- Hệ thống còn an toàn không?

4.2.3. Các giải thuật tránh deadlocks

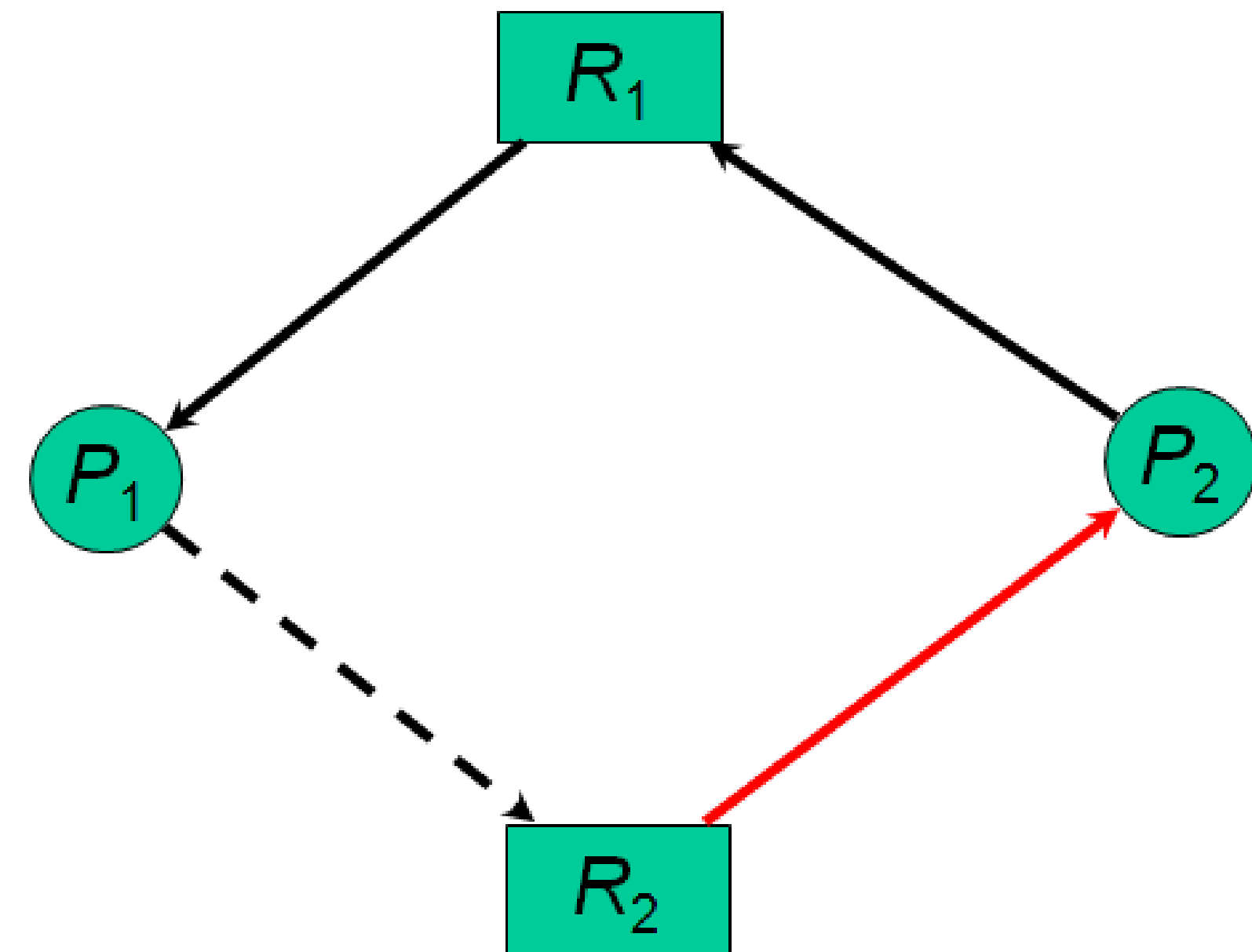
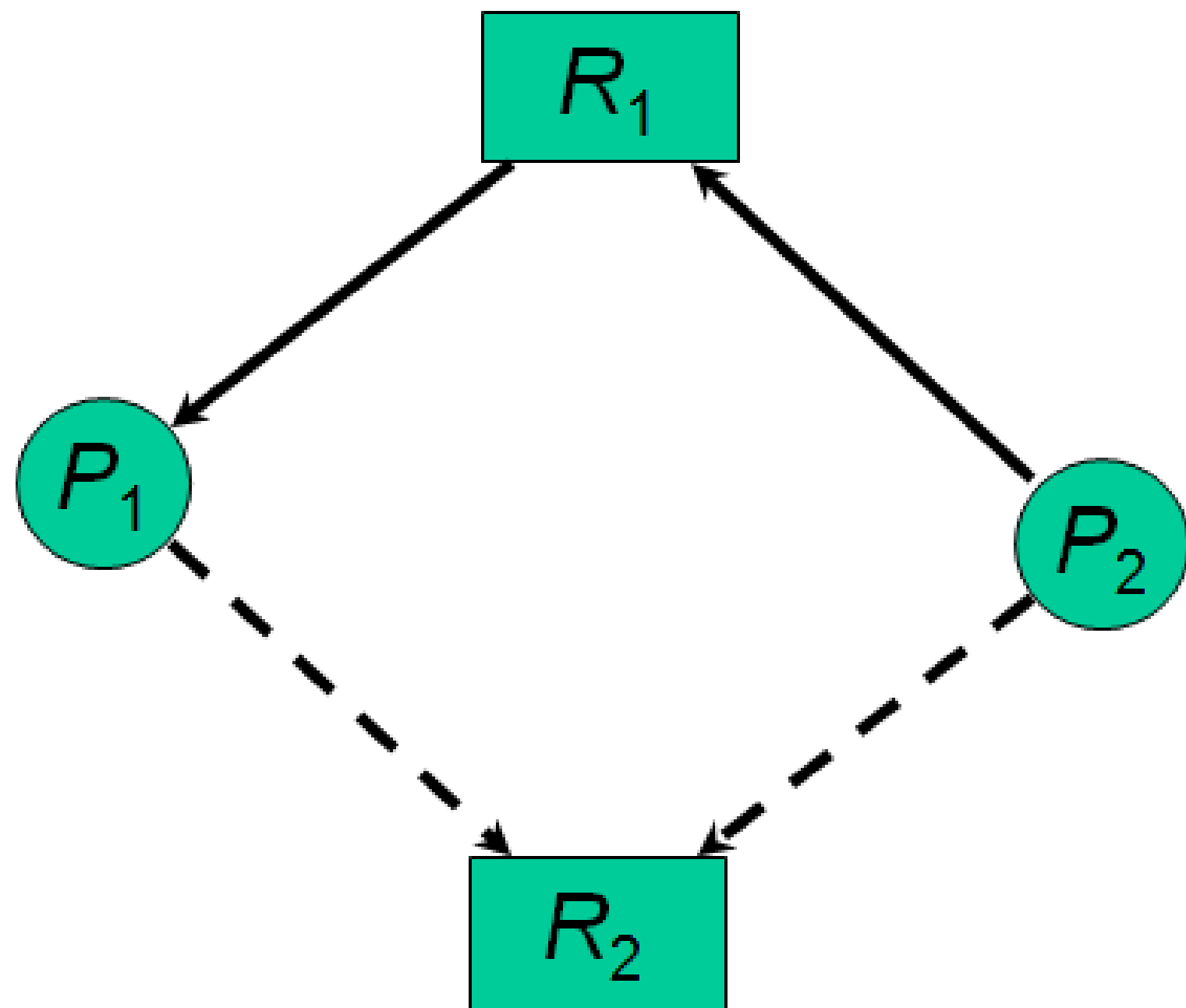
- Mỗi tài nguyên chỉ có một thực thể
- Giải thuật đồ thị cấp phát tài nguyên.
- Mỗi tài nguyên có nhiều thực thể
- Giải thuật Banker

Deadlocks



Sharing is learning

4.2.3.1. Đồ thị cấp phát tài nguyên



4.2.3.2. Giải thuật Banker

- Dùng để kiểm tra trạng thái an toàn của hệ thống tại một thời điểm.
- B1: tìm kiếm tiến trình i thỏa $\text{Need} \leq \text{Available}$. Nếu không có • **Unsafe**.
- B2: $\text{Available} += \text{Allocation}$. $\text{Finish}[i] = \text{true}$

Quay về B1.

	Max	Allocation	Need	Available
P0	10	5	5	3
P1	4	2	2	5
P2	9	2	7	10
				12

- Chuỗi $\langle P1, P0, P2 \rangle$ là chuỗi an toàn • hệ thống là an toàn.

4.2.3.2. Giải thuật Banker

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

Available

A	B	C
3	3	2
5	3	2
7	4	3
7	4	5
10	4	7
10	5	7

- Chuỗi $\langle P1, P3, P4, P2, P0 \rangle$ là chuỗi an toàn · hệ thống là an toàn.

4.2.3.2. Giải thuật Banker yêu cầu tài nguyên cho 1 tiến trình

(Request_i[j] = k ⇔ P_i cần k instance của tài nguyên R_j)

B1. Nếu Request_i ≤ Need_i thì đến B2. Nếu không, báo lỗi vì tiến trình đã vượt yêu cầu tối đa.

B2. Nếu Request_i ≤ Available thì qua B3. Nếu không, P_i phải chờ vì tài nguyên không còn đủ để cấp phát.

B3. Giả định cấp phát tài nguyên đáp ứng yêu cầu của P_i bằng cách cập nhật trạng thái hệ thống như sau:

$$\text{Available} = \text{Available} - \text{Request}_i$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i$$

4.2.3.2. Giải thuật Banker yêu cầu tài nguyên cho 1 tiến trình

- P1 yêu cầu (1, 0, 2) · Request1 \leq Available (3, 3, 2).

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	2	4	1			
P1	2	0	0	1	2	2			
P2	2	1	1	2	1	1			

- Chuỗi <P1, P2, P0> là chuỗi an toàn · Có thể đáp ứng.

4.2.3.2. Giải thuật Banker yêu cầu tài nguyên cho 1 tiến trình

- P1 yêu cầu (1, 0, 2) · Request1 \leq Available (3, 3, 2).

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	2	4	1	2	3	0
P1	3	0	2	0	2	0	5	3	2
P2	2	1	1	2	1	1	7	4	3
							7	5	7

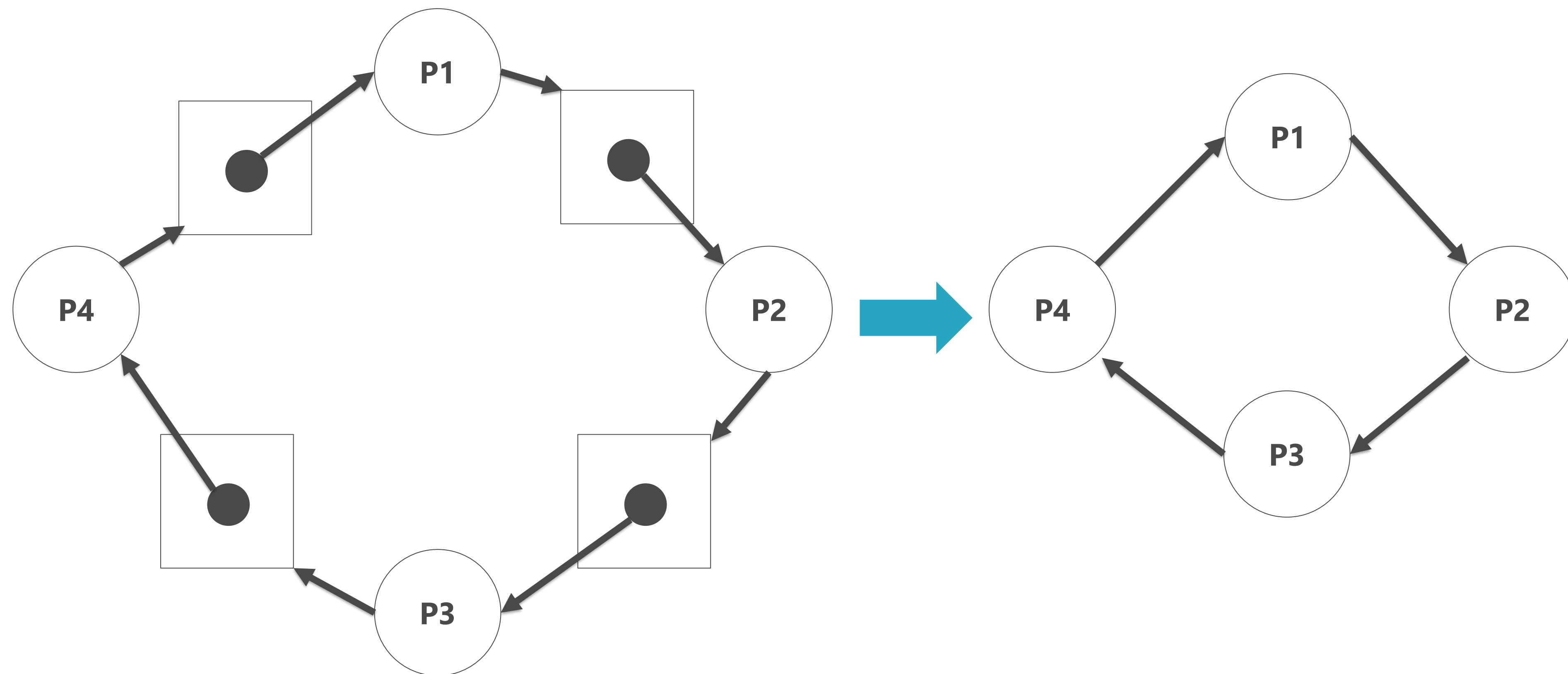
- Chuỗi <P1, P2, P0> là chuỗi an toàn · Có thể đáp ứng.

4.2.3.3. Phát hiện deadlocks

- Chấp nhận xảy ra deadlocks trong hệ thống
- Giải thuật phát hiện deadlocks
- Cơ chế phục hồi

4.2.3.3 Phát hiện deadlocks

- Đối với tài nguyên chỉ có 1 thực thể
- Sử dụng wait-for graph.



4.2.3.3 Phát hiện deadlocks

Đối với tài nguyên nhiều thực thể: tương tự như giải thuật Banker nhưng lúc này sẽ không xét Need mà xét cho nhiều Request:

- Nếu như process i có $\text{Finish}[i] = \text{false}$ sau khi đã kết thúc thuật toán thì hệ thống đang ở trạng thái deadlocks (vì không còn tài nguyên để đáp ứng cho process i).

4.2.3.3 Phát hiện deadlocks

	Allocation			Request		
	A	B	C	A	B	C
P0	0	1	0	7	4	3
P1	2	0	0	1	2	2
P2	3	0	2	6	0	0
P3	2	1	1	0	1	1
P4	0	0	2	4	3	1

Available

A	B	C
3	3	2
5	3	2
7	4	3
7	4	5
10	4	7
10	5	7

- Chuỗi $\langle P1, P3, P4, P2, P0 \rangle$ sẽ cho $Finish[i] = true$ với $i = 0 \cdot 4$.

Deadlocks



Sharing is learning

4.2.3.3 Phát hiện deadlocks

	Allocation			Request		
	A	B	C	A	B	C
P0	0	1	0	7	4	3
P1	2	0	0	1	2	2
P2	3	0	2	6	0	0
P3	2	1	1	6	1	1
P4	0	0	2	4	3	1

Available

A	B	C
3	3	2
5	3	2
5	3	4

- Xảy ra deadlock tại P0, P2, P3.

4.2.3.3. Phục hồi deadlocks

- Khi deadlocks xảy ra, để phục hồi:
 - + Báo người vận hành
 - + Hệ thống tự động phục hồi bằng cách bẻ gãy chu trình deadlocks:
 - Chấm dứt lần lượt từng tiến trình cho đến khi không còn deadlocks.
 - Lấy lại tài nguyên từ một hay nhiều tiến trình.
 - Sử dụng giải thuật phát hiện deadlocks để xác định còn deadlocks hay không.

4.2.3.3. Phục hồi deadlocks

Dựa trên yếu tố nào để chấm dứt?

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

4.2.3.3. Phục hồi deadlocks

- Lấy tài nguyên từ một tiến trình · tiến trình khác cho đến khi hết deadlocks.
- Chọn nạn nhân để giảm thiểu chi phí.
- Trở lại trạng thái trước deadlocks (Rollback):
 - Rollback 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.
 - (Starvation): để tránh starvation, phải bảo đảm không có tiến trình sẽ luôn luôn bị lấy lại tài nguyên mỗi khi deadlocks xảy ra.

5. BÀI TẬP ÔN

BT1: “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 deadlocks nào?

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

Deadlocks



BT2: 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.

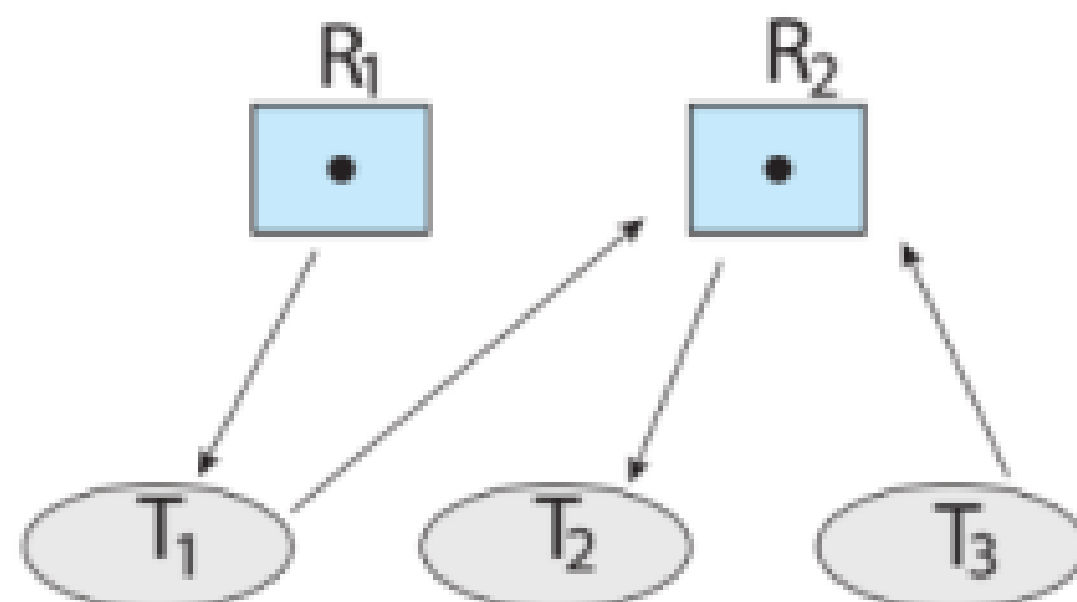
Khi xảy ra deadlocks, các giải pháp nào có thể được sử dụng để phục hồi hệ thống? (G1)

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

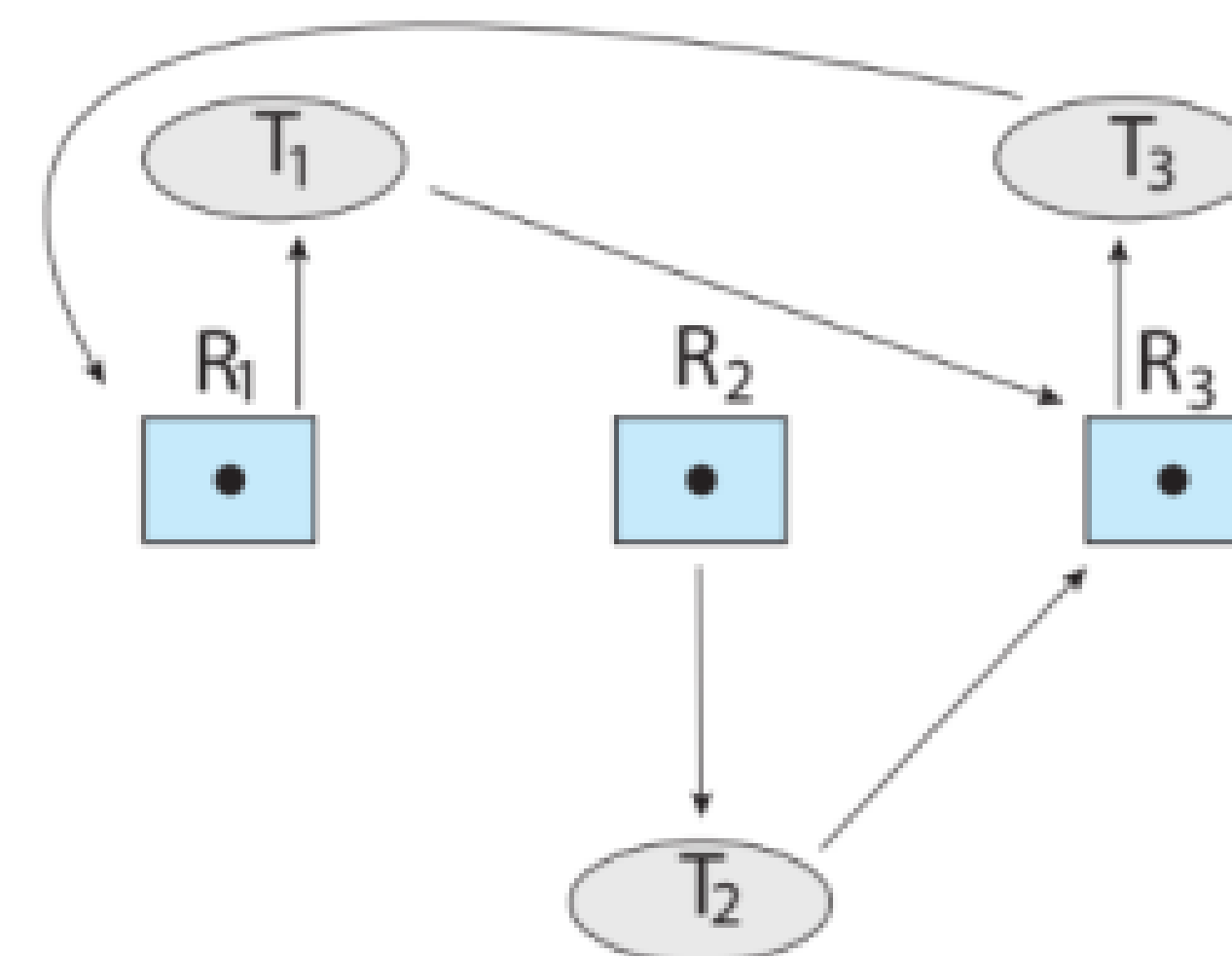
BT3: Chọn phát biểu SAI trong các phát biểu bên dưới? (G2)

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

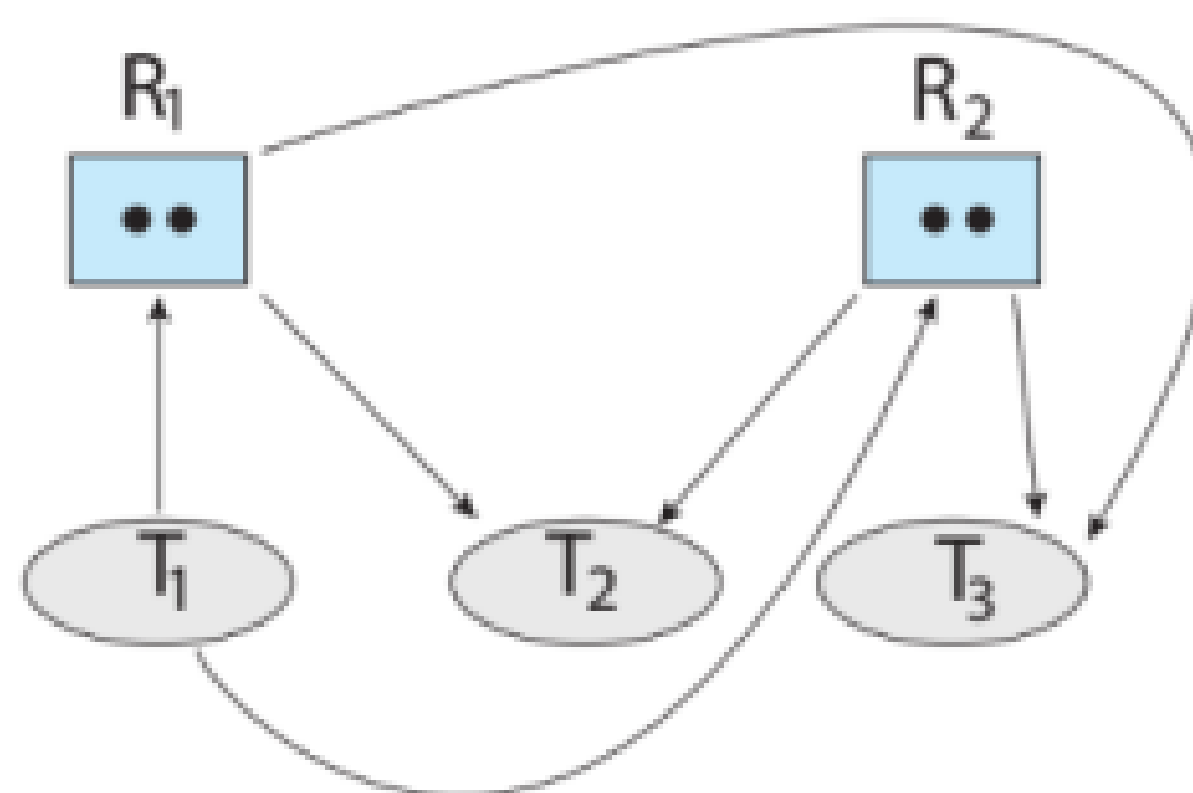
(a)



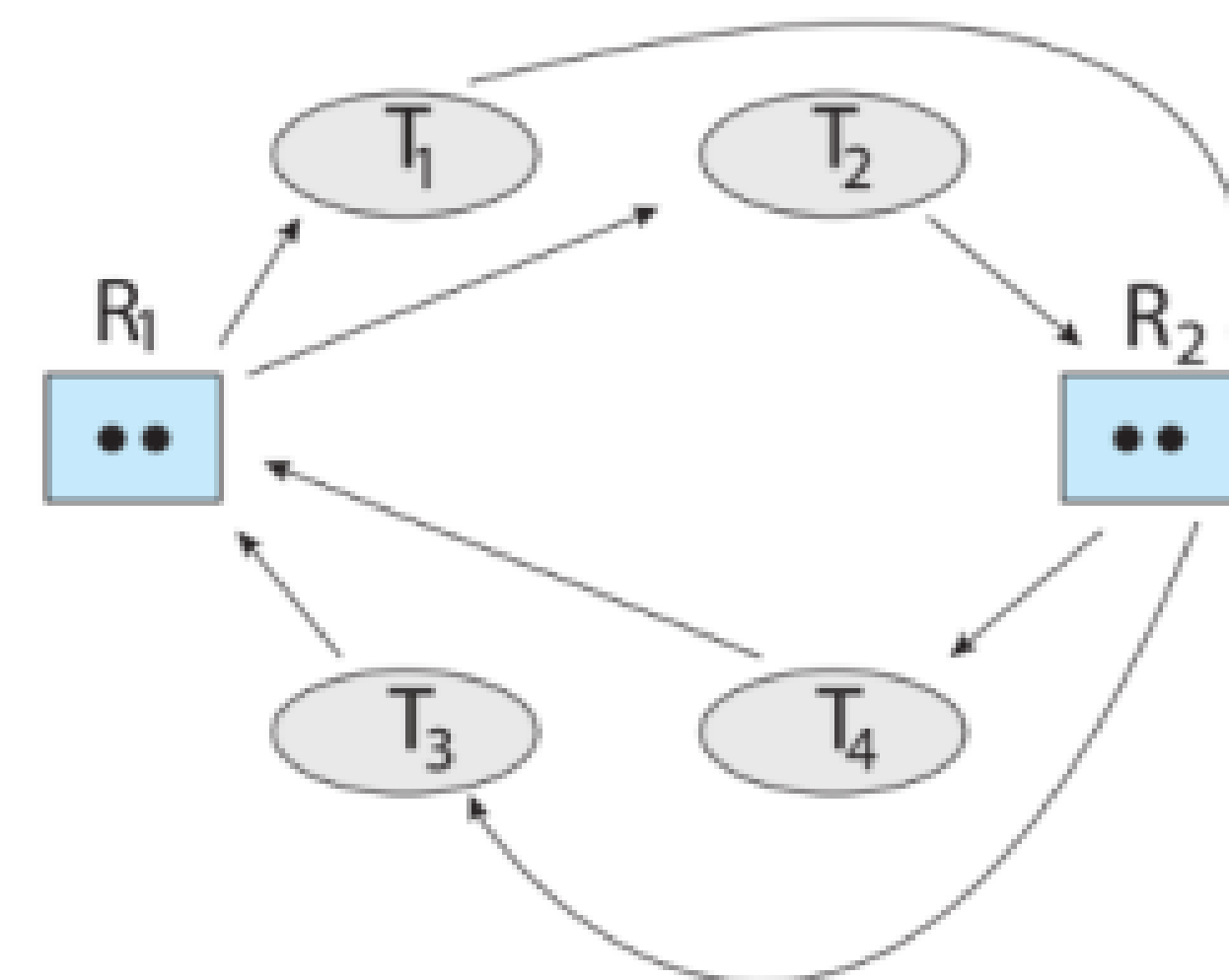
(b)



(c)



(d)



BT4: Chọn đáp án chứa các đồ thị có deadlocks

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



Sharing is learning

3. Quản lý bộ nhớ

Nội dung Training



Sharing is learning

I. Khái niệm

II. Các kiểu địa chỉ nhớ

III. Mô hình quản lý bộ nhớ

IV. Cơ chế phân trang

V. TLB (Translation Look-aside Buffer)

VI. Phân đoạn bộ nhớ

Nhắc lại

- Một chương trình phải được mang vào trong bộ nhớ chính và đặt nó trong một tiến trình để được xử lý
- Trước khi được vào bộ nhớ chính, các tiến trình phải đợi trong một Input Queue.

Nhiệm vụ của hệ điều hành trong quản lý bộ nhớ

- 5 nhiệm vụ chính bao gồm: Cấp phát bộ nhớ cho process, tái định vị, bảo vệ, chia sẻ, kết gán địa chỉ nhớ luận lý
- **Mục tiêu:** Nạp càng nhiều process vào bộ nhớ càng tốt
- **Lưu ý:** Trong hầu hết hệ thống, kernel chiếm 1 phần cố định của bộ nhớ (low memory), phần còn lại phân phối các process (high memory)

Các kiểu địa chỉ bộ nhớ



Phân loại

- **Địa chỉ vật lý** (physical address – địa chỉ thực): là một vị trí thực trong bộ nhớ chính.
 - Địa chỉ tuyệt đối (absolute address): địa chỉ tương đương với địa chỉ thực
 - Địa chỉ vật lý = địa chỉ frame + offset
- **Địa chỉ luận lý** (logical address – virtual address – địa chỉ ảo): vị trí nhớ được diễn tả trong một chương trình.
 - Địa chỉ tương đối (relative address): địa chỉ được biểu diễn tương đối so với một vị trí xác định nào đó và không phụ thuộc vào vị trí thực của tiến trình trong bộ nhớ.
 - Địa chỉ luận lý = địa chỉ page + offset

Để truy cập bộ nhớ, địa chỉ luận lý cần được biến đổi thành địa chỉ vật lý.

Các kiểu địa chỉ bộ nhớ

Ví dụ:

Địa chỉ vật lý là 4100 sẽ được chuyển thành địa chỉ ảo bao nhiêu? Biết rằng kích thước mỗi frame là 1KB và bảng ánh xạ địa chỉ ảo như bảng.

Frame	Page
0	6
1	4
2	5
3	7
4	1
5	9

Hướng giải:

B1: KB \rightarrow bytes

B2: (*) div 1024: lấy phần nguyên của địa chỉ đã cho với 1024 \rightarrow Địa chỉ ở page nào.

B3: page x 1024 + ((*) % 1024) \rightarrow địa chỉ luận lý

Giải:

B1: 1KB = 1024 bytes

B2: 4100 div 1024 = 4 ta được địa chỉ ở frame 4, so với bảng ánh xạ địa chỉ \rightarrow Địa chỉ ở page 1.

B3: 1 x 1024 + (4100 % 1024) = 1024 + 4 = 1028 (địa chỉ luận lý)

Các kiểu địa chỉ bộ nhớ



Sharing is learning

Bài tập trắc nghiệm:

Cho bảng phân trang sau. Với kích thước mỗi trang là 1KB, hãy chuyển địa chỉ logic 3580 sang địa chỉ vật lý. Hãy chọn đáp án đúng.

Page	Frame
0	3
1	2
2	6
3	4

A. 4500

B. 3950

☒ C. 4604

D. 4230

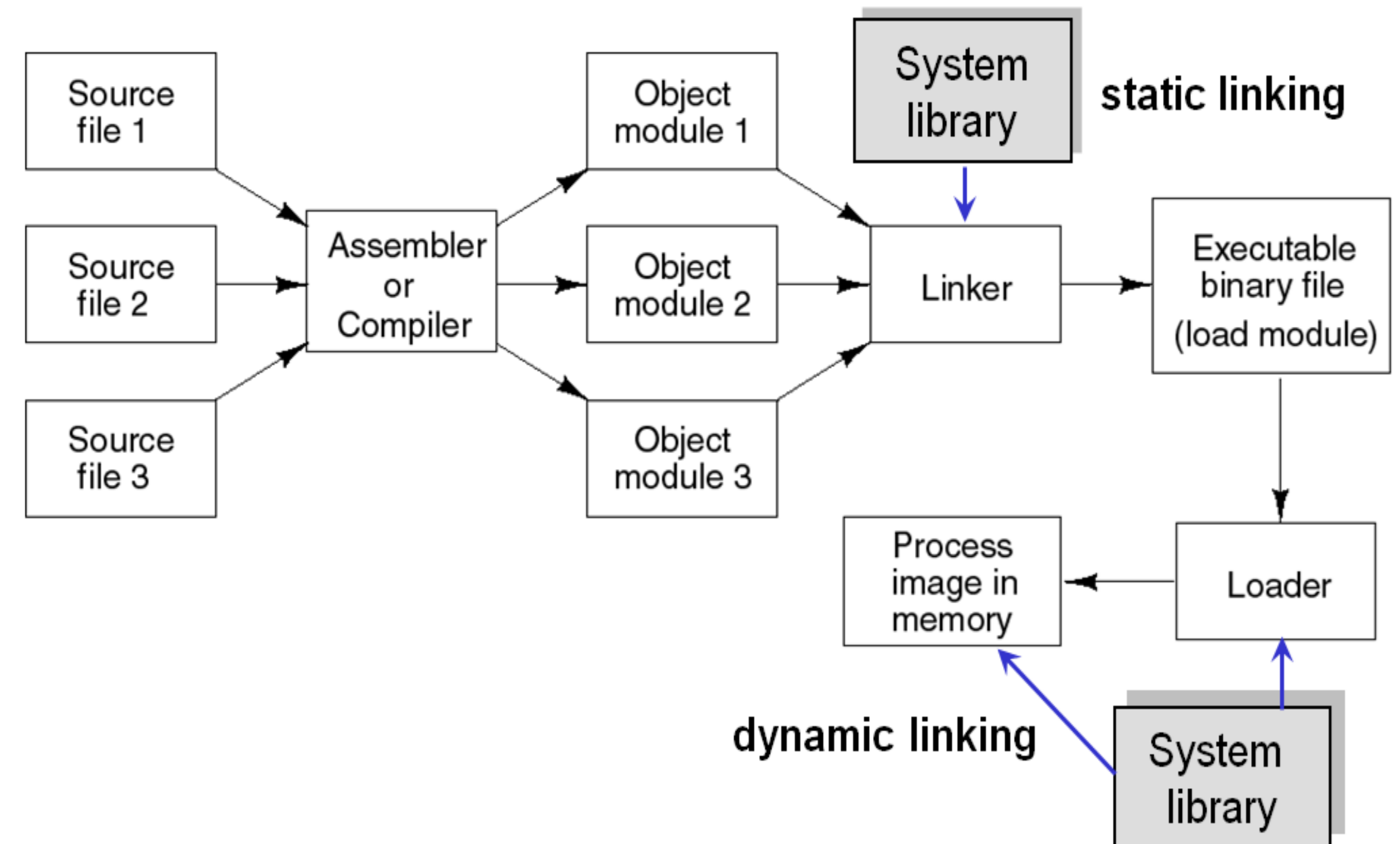
Các kiểu địa chỉ bộ nhớ



Sharing is learning

Nạp chương trình vào bộ nhớ

- **Linker:** kết hợp nhiều object module thành một file nhị phân (file tạo thành gọi là load module).
- **Loader:** nạp module vào bộ nhớ chính.



- **Lưu ý:** Cần phân biệt được sự khác nhau giữa dynamic và static linker/loader

Các kiểu địa chỉ bộ nhớ



Câu hỏi trắc nghiệm

Câu 3: Quá trình chuyển đổi từ địa chỉ luận lý sang địa chỉ thực có thể diễn ra tại thời điểm nào?

- A. Thời điểm biên dịch (compiling time)
- B. Thời điểm nạp chương trình (loading time)
- C. Thời điểm thực thi (execution time)
- ☒ D. Cả a, b, và c đều đúng

Giải thích: Địa chỉ luận lý có thể chuyển thành địa chỉ thực tại 3 thời điểm

- Compile time: nếu biết trước địa chỉ bộ nhớ của chương trình thì có thể gán địa chỉ tuyệt đối lúc biên dịch
- Load time: chuyển đổi địa chỉ luận lý thành địa chỉ thực dựa trên địa chỉ nền
- Excution time: khi process di chuyển qua lại giữa các segment trong bộ nhớ

Các mô hình quản lý bộ nhớ



Sharing is learning

Khái niệm

- Một Process phải được nạp hoàn toàn vào bộ nhớ thì mới được thực thi
- Một số cơ chế quản lý bộ nhớ:
 - Phân chia cố định (fixed partitioning)
 - Phân chia động (dynamic partitioning)
 - Phân trang đơn giản (simple paging)

0	0
1	1
2	2
3	3

Process A
page table

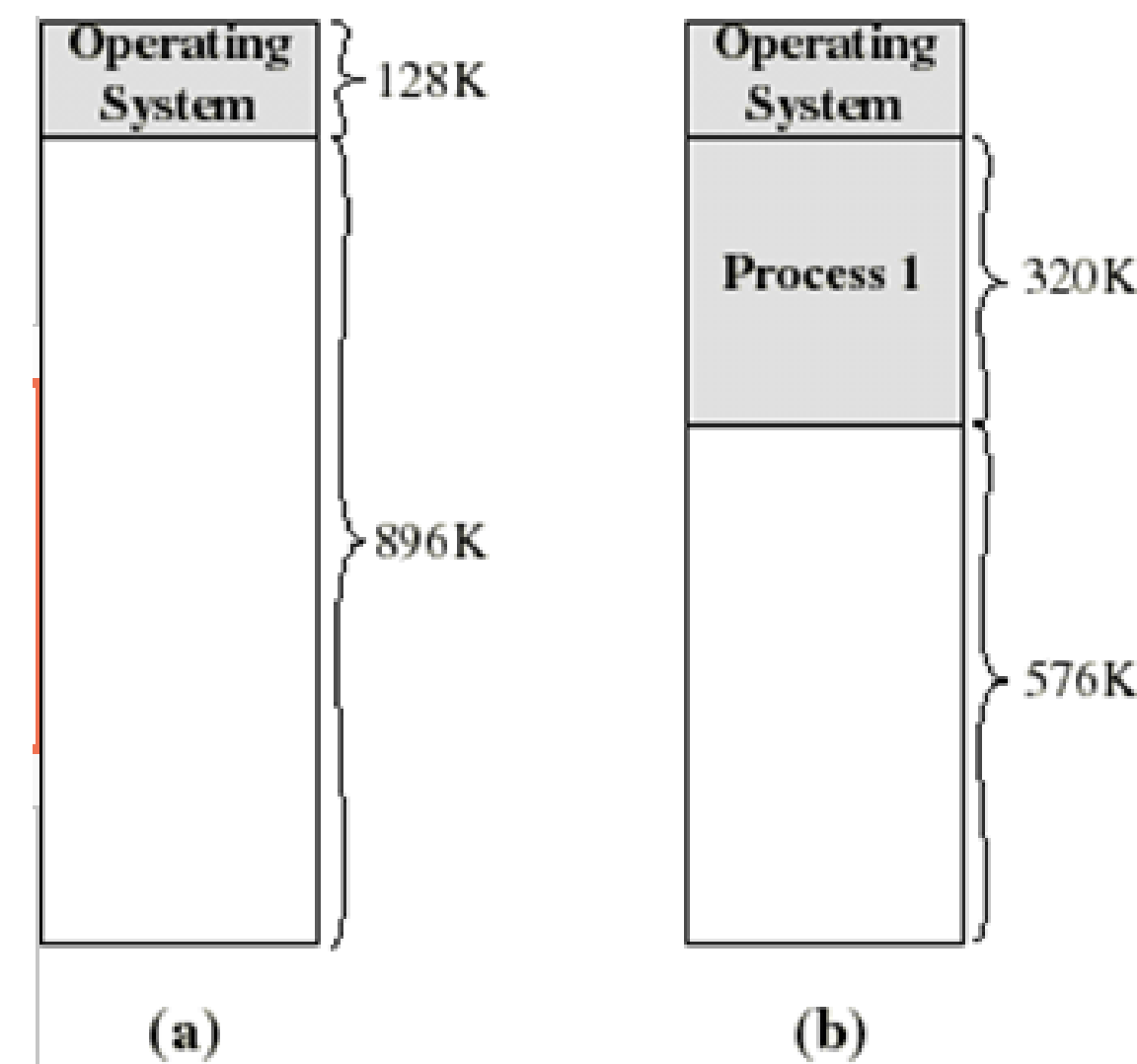
0	—
1	—
2	—

Process B
page table

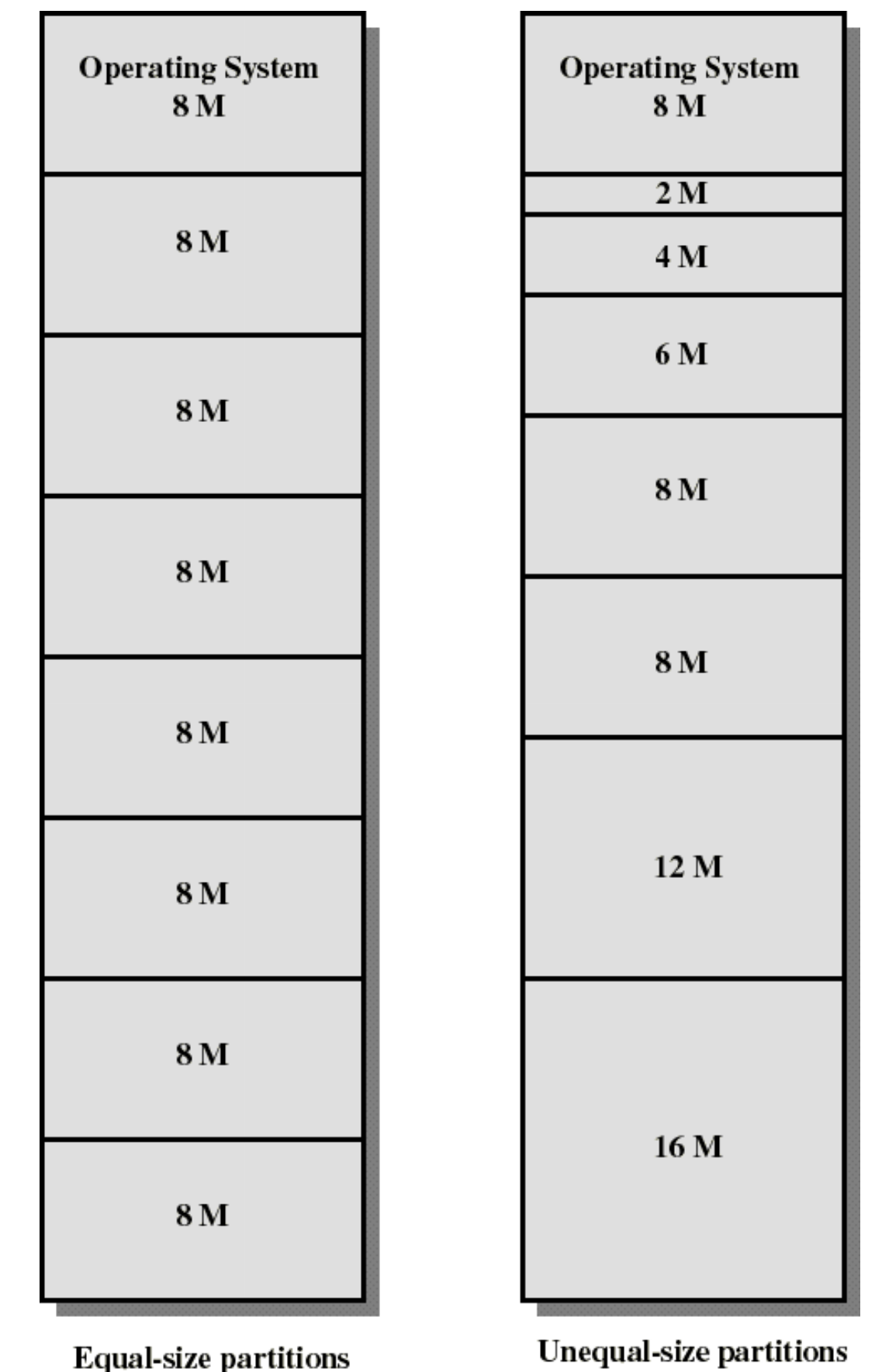
0	7
1	8
2	9
3	10

Process C
page table

Phân trang đơn giản



Phân chia động



Phân chia cố định

Các mô hình quản lý bộ nhớ

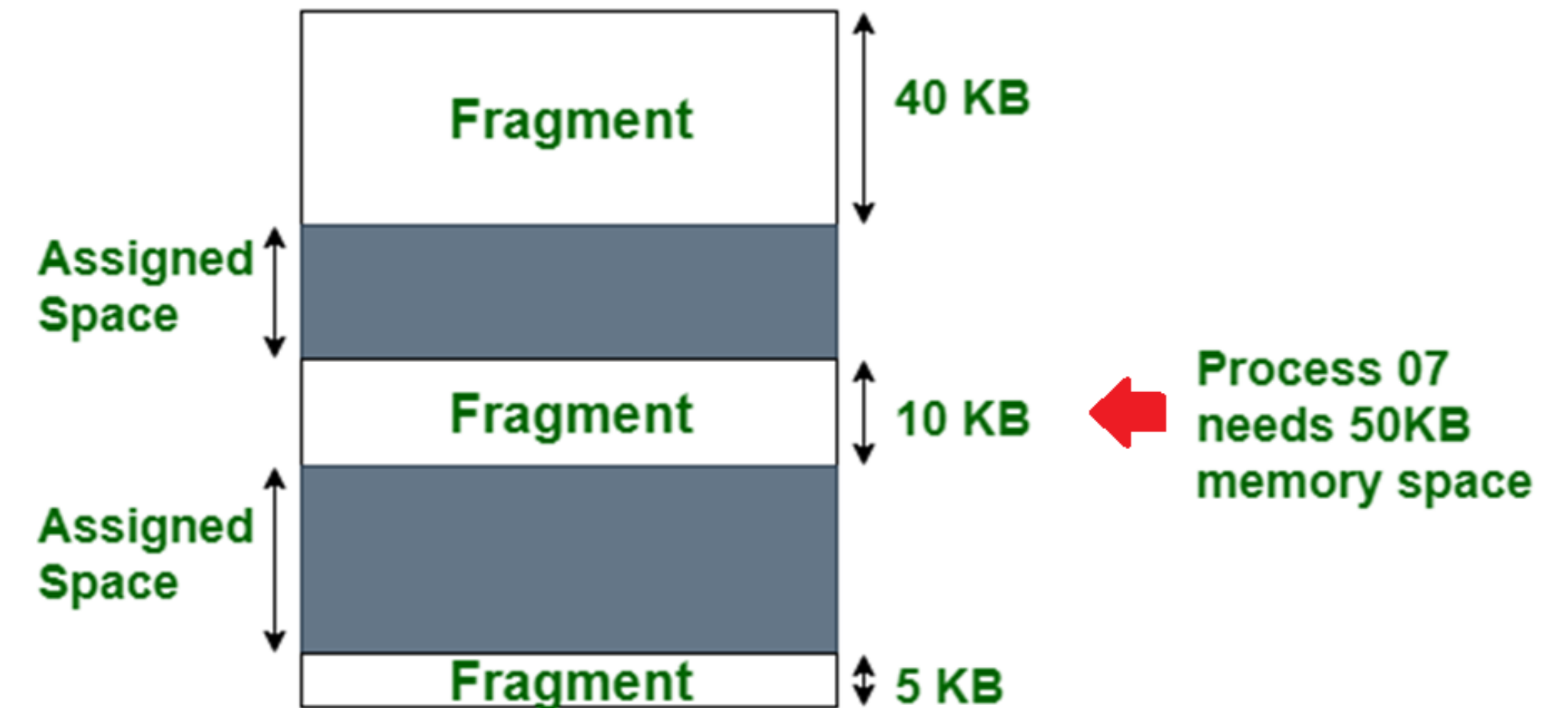


Sharing is learning

Hiện tượng phân mảnh bộ nhớ (fragmentation)

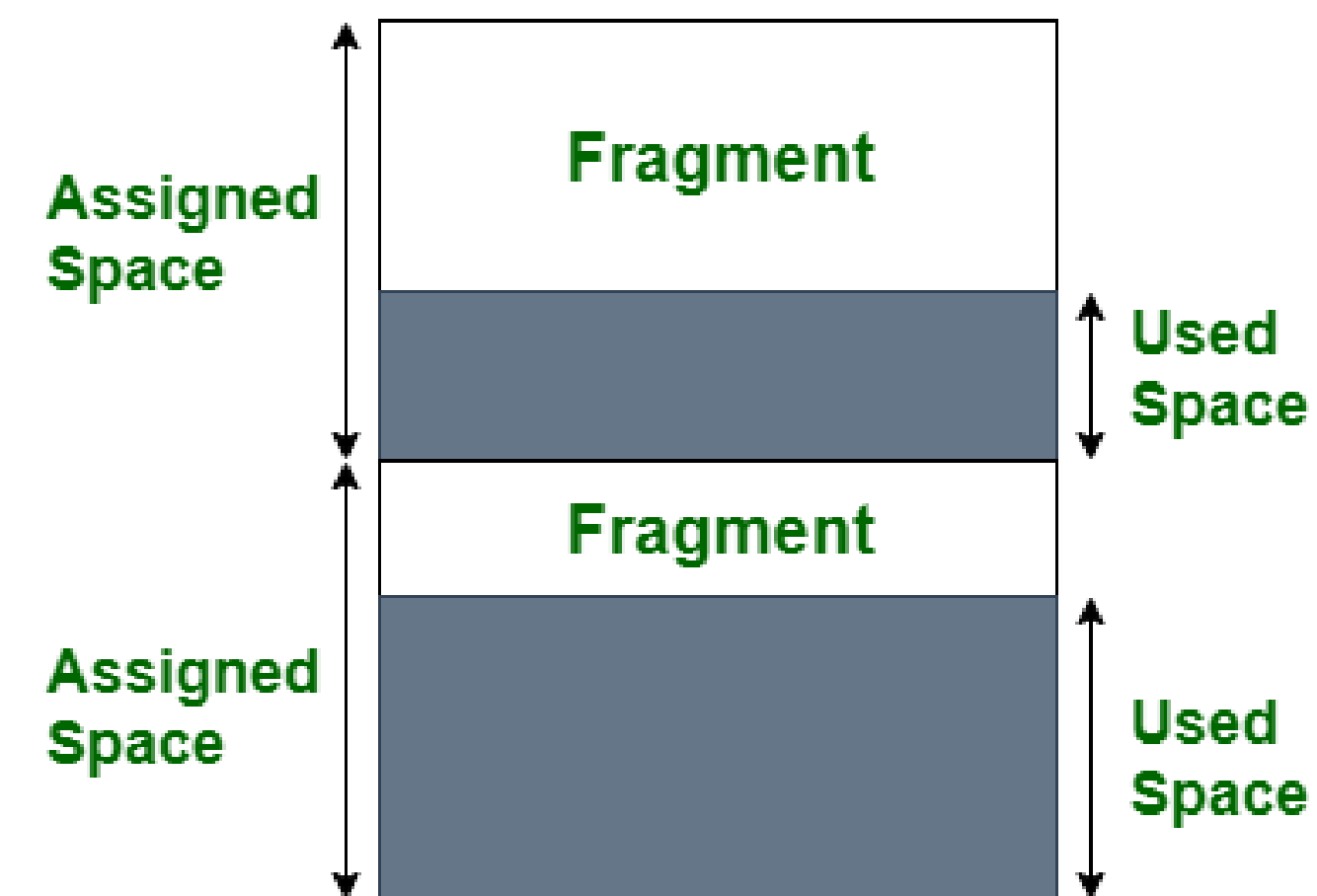
- **Hiện tượng phân mảnh ngoại:** Kích thước không gian bộ nhớ trống đủ thỏa mãn yêu cầu cấp phát, tuy nhiên không liên tục

=> Giải pháp: Dùng cơ chế compaction



- **Hiện tượng phân mảnh nội:** Kích thước vùng nhớ cấp phát lớn hơn yêu cầu (xảy ra khi bộ nhớ được chia thành các khối có kích thước cố định)

=> Giải pháp: sử dụng các chiến lược placement.



Internal Fragmentation

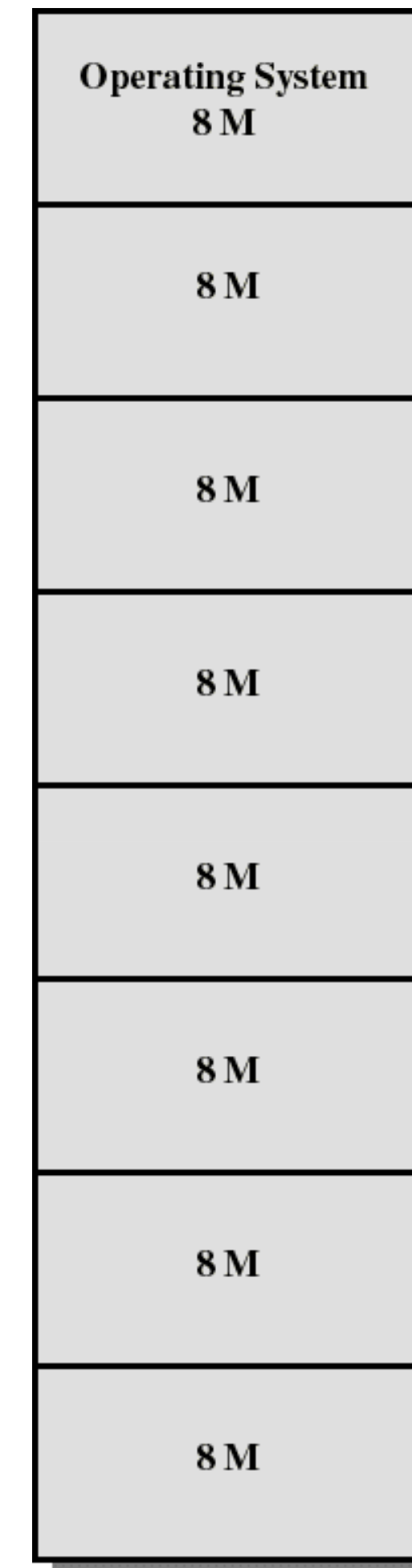
Các mô hình quản lý bộ nhớ



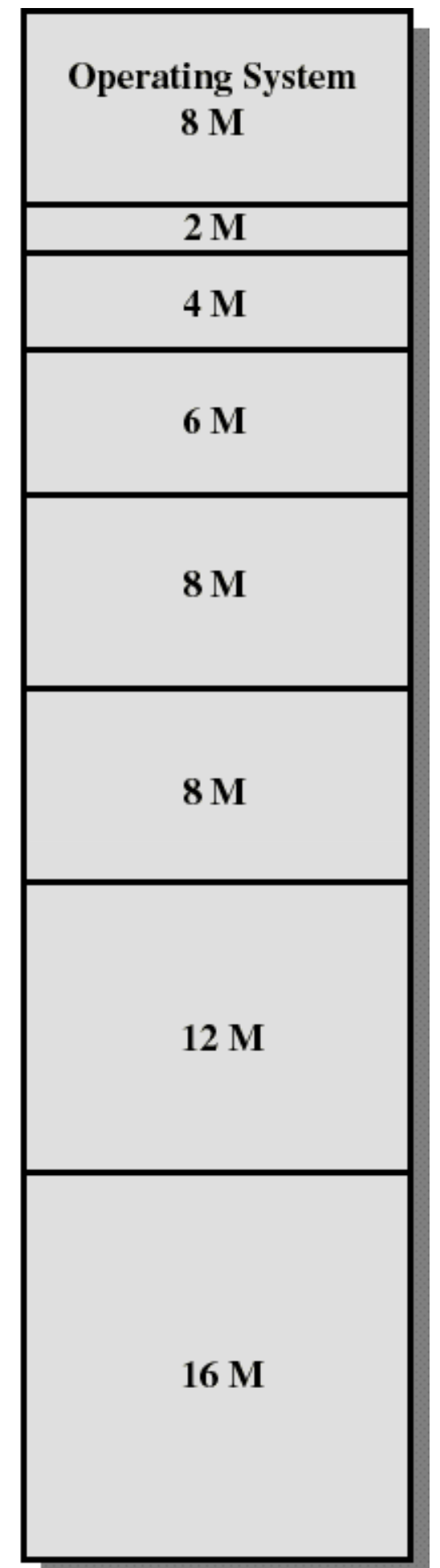
Sharing is learning

Phân chia cố định (Fixed partitioning)

- Bộ nhớ chính chia thành nhiều phần, có kích thước bằng hoặc khác nhau.
- Process nào nhỏ hơn kích thước partition thì có thể nạp vào.
- Nếu process có kích thước lớn hơn => Overlay.
- **Nhận xét:** kém hiệu quả do bị phân mảnh nội.



Equal-size partitions



Unequal-size partitions

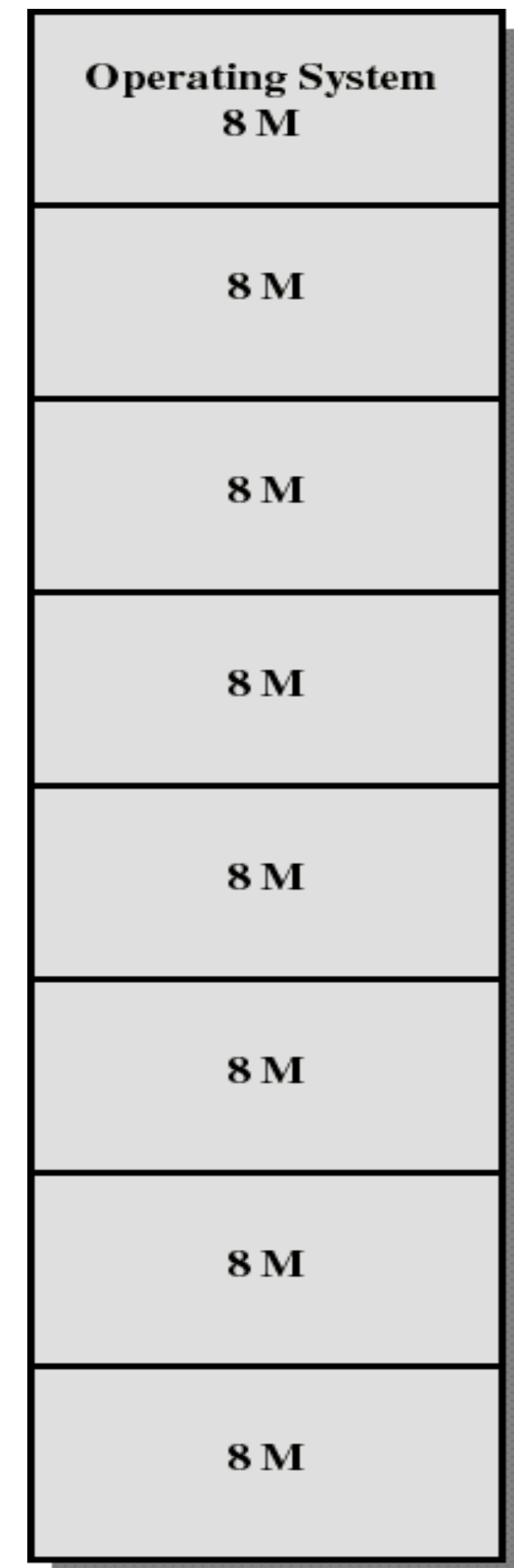
Các mô hình quản lý bộ nhớ



Sharing is learning

Phân chia cố định (Fixed partitioning)

- **Chiến lược placement với partition cùng kích thước:**
 - Còn partition trống => nạp vào.
 - Không còn partition trống => swap process đang bị blocked ra bộ nhớ phụ, nhường chỗ cho process mới.



Equal-size partitions

Các mô hình quản lý bộ nhớ



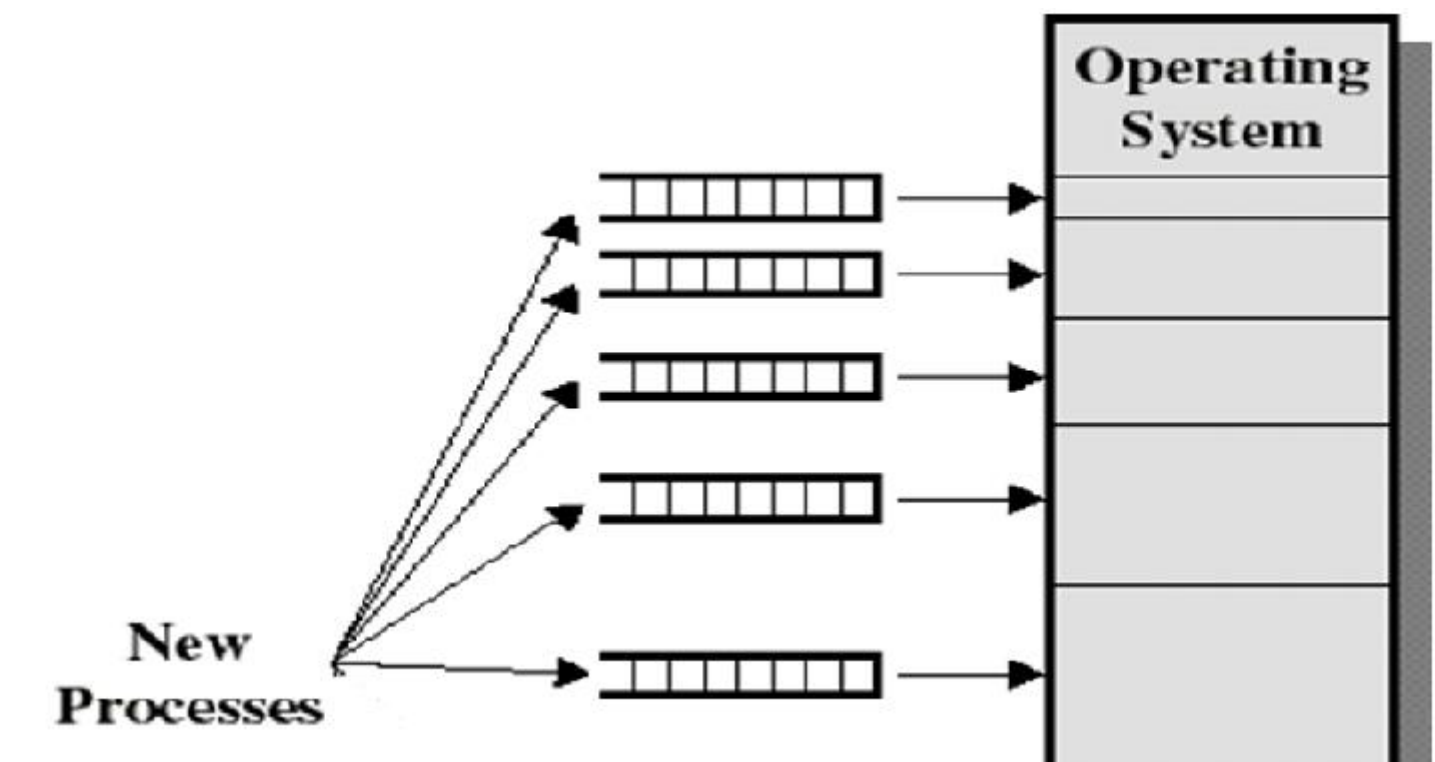
Sharing is learning

Phân chia cố định (Fixed partitioning)

– Chiến lược placement với partition khác kích thước:

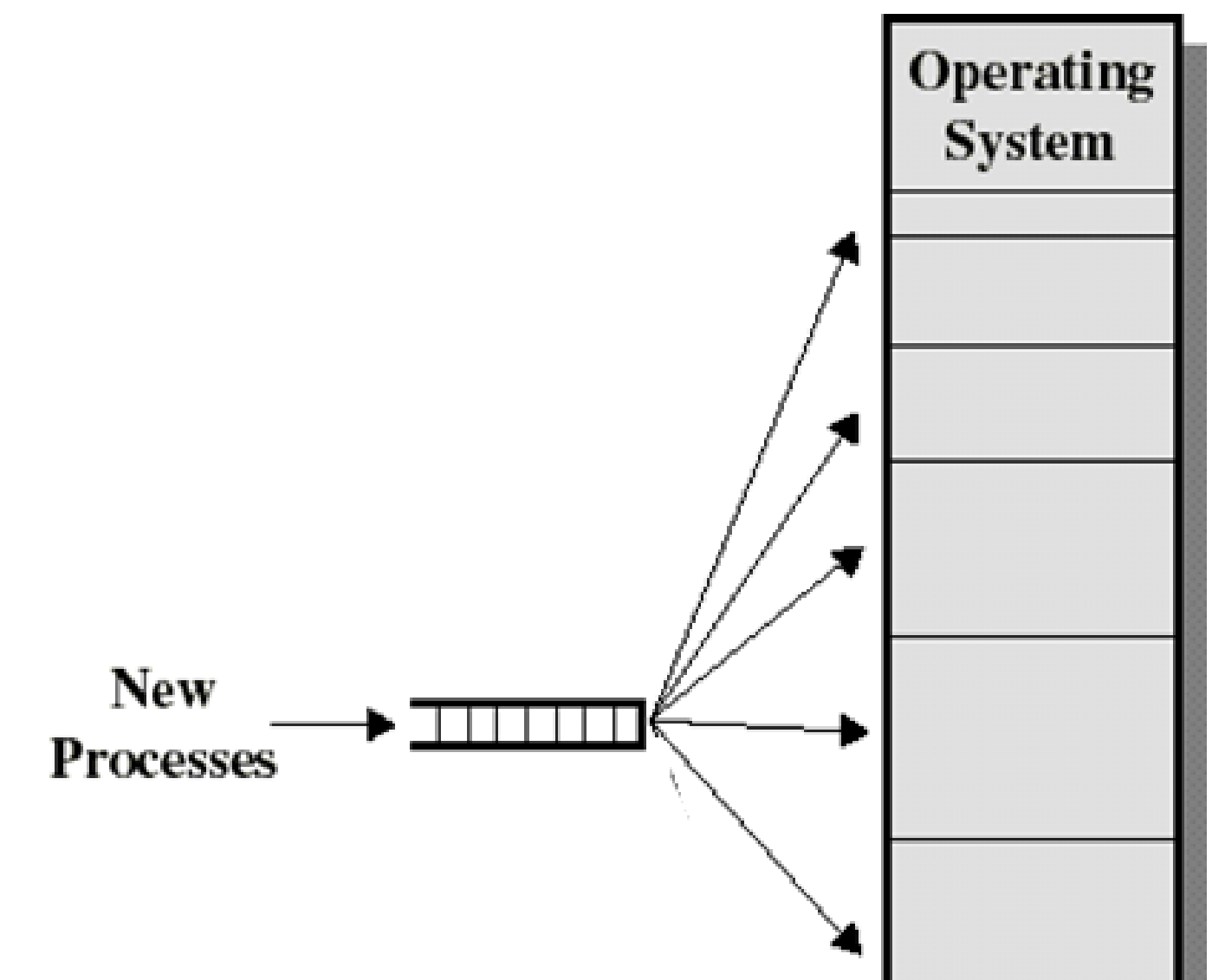
+ Giải pháp 1: Sử dụng nhiều hàng đợi

- Mỗi process xếp hàng vào partition nhỏ nhất phù hợp.
- **Ưu điểm:** giảm thiểu phân mảnh nội.
- **Nhược điểm:** Có thể có hàng đợi trống.



+ Giải pháp 2: Sử dụng 1 hàng đợi

- Chỉ có một hàng đợi chung cho tất cả partition.
- Khi cần nạp process => chọn partition nhỏ nhất.



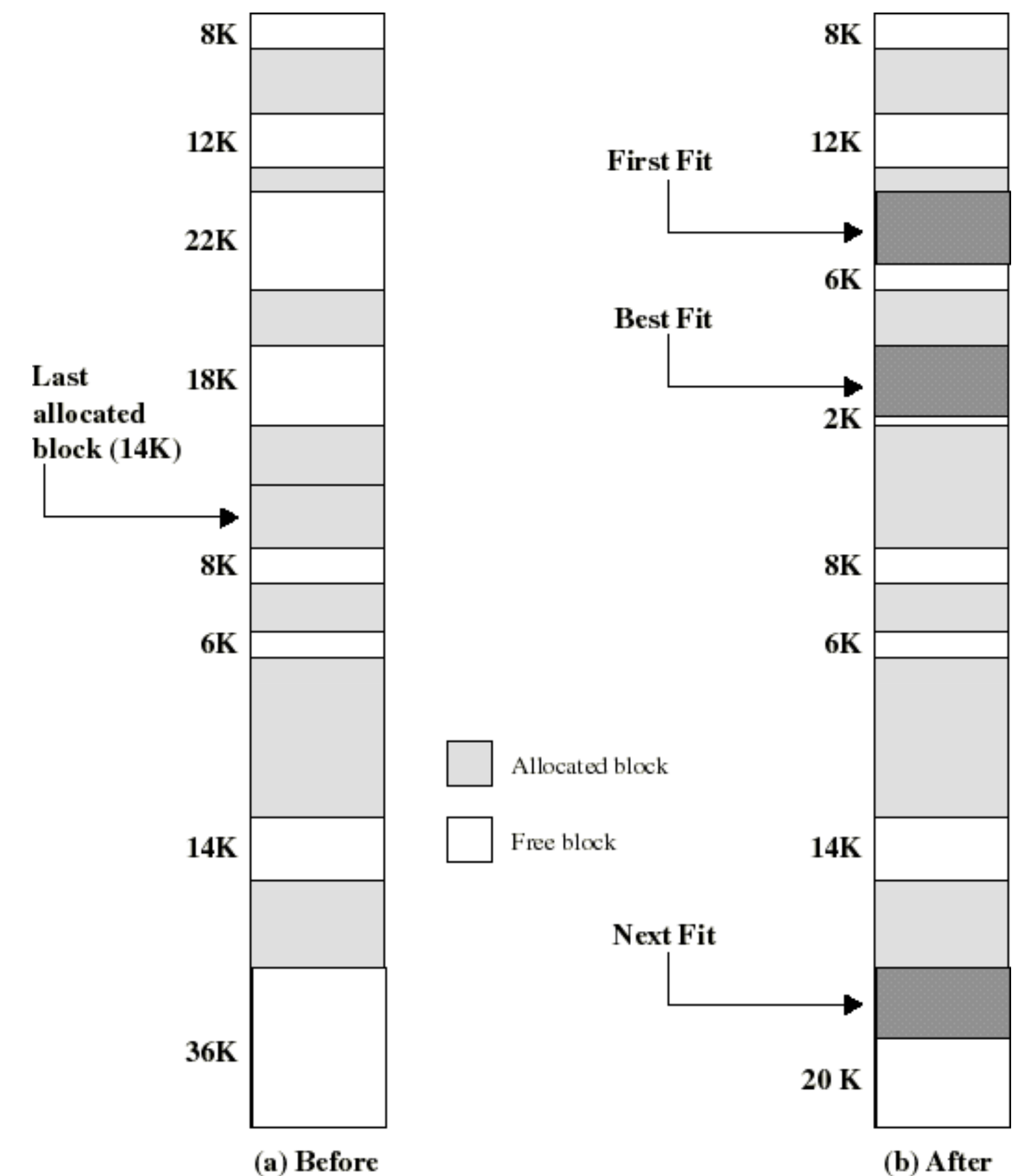
Các mô hình quản lý bộ nhớ



Sharing is learning

Phân chia động (dynamic partitioning)

- Số lượng partition và kích thước không cố định, có thể khác nhau.
- Mỗi process được cấp phát chính xác dung lượng bộ nhớ cần thiết.
- **Nhận xét:** gây hiện tượng phân mảnh ngoại.
- **Chiến lược placement (giúp giảm chi phí compaction):**
 - Best-fit: chọn khối nhớ trống phù hợp 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

Các mô hình quản lý bộ nhớ



Sharing is learning

Ví dụ:

Bộ nhớ chính được phân thành các phân vùng 600K, 500K, 200K và 300K (theo thứ tự). Các tiến trình có kích thước 212K, 417K, 112K và 426K (theo thứ tự) sẽ được cấp phát như thế nào?

212K

417K

112K

426K

600K

500K

200K

300K

Các mô hình quản lý bộ nhớ



Sharing is learning

Chiến lược placement:

Best – fit: chọn khối nhớ trống phù hợp nhất.

212K

417K

112K

426K

600K
500K
200K
300K

Các mô hình quản lý bộ nhớ



Sharing is learning

Chiến lược placement:

First – fit: chọn khối nhớ trống phù hợp đầu tiên.

212K

417K

112K

426K

600K
500K
200K
300K

Không còn vùng nhớ trống thỏa yêu cầu!!!

Các mô hình quản lý bộ nhớ



Sharing is learning

Chiến lược placement:

Worst – fit: chọn khối nhớ trống lớn nhất.

212K

417K

112K

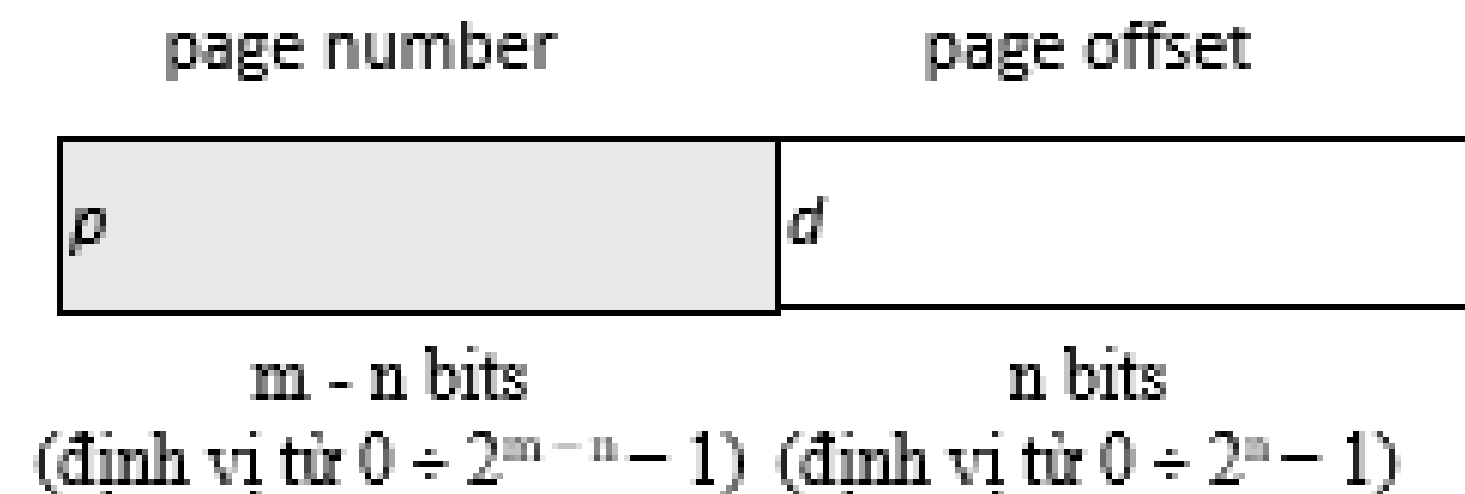
426K

600K
500K
200K
300K

Không còn vùng nhớ trống thoả yêu cầu!!!

Phân chia trang (paging)

- Chia nhỏ bộ nhớ vật lý thành các khối nhỏ có kích thước bằng nhau (frames). Kích thước của frame là lũy thừa của 2 (từ 512 bytes đến 16MB).
- Chia bộ nhớ luận lý của tiến trình thành các khối nhỏ có kích thước bằng nhau (pages). Địa chỉ luận lý gồm có: số hiệu trang (page number) và độ dời của địa chỉ tính từ đầu trang đó (page offset).



- Bảng phân trang (page table) dùng để ánh xạ địa chỉ luận lý thành địa chỉ thực.

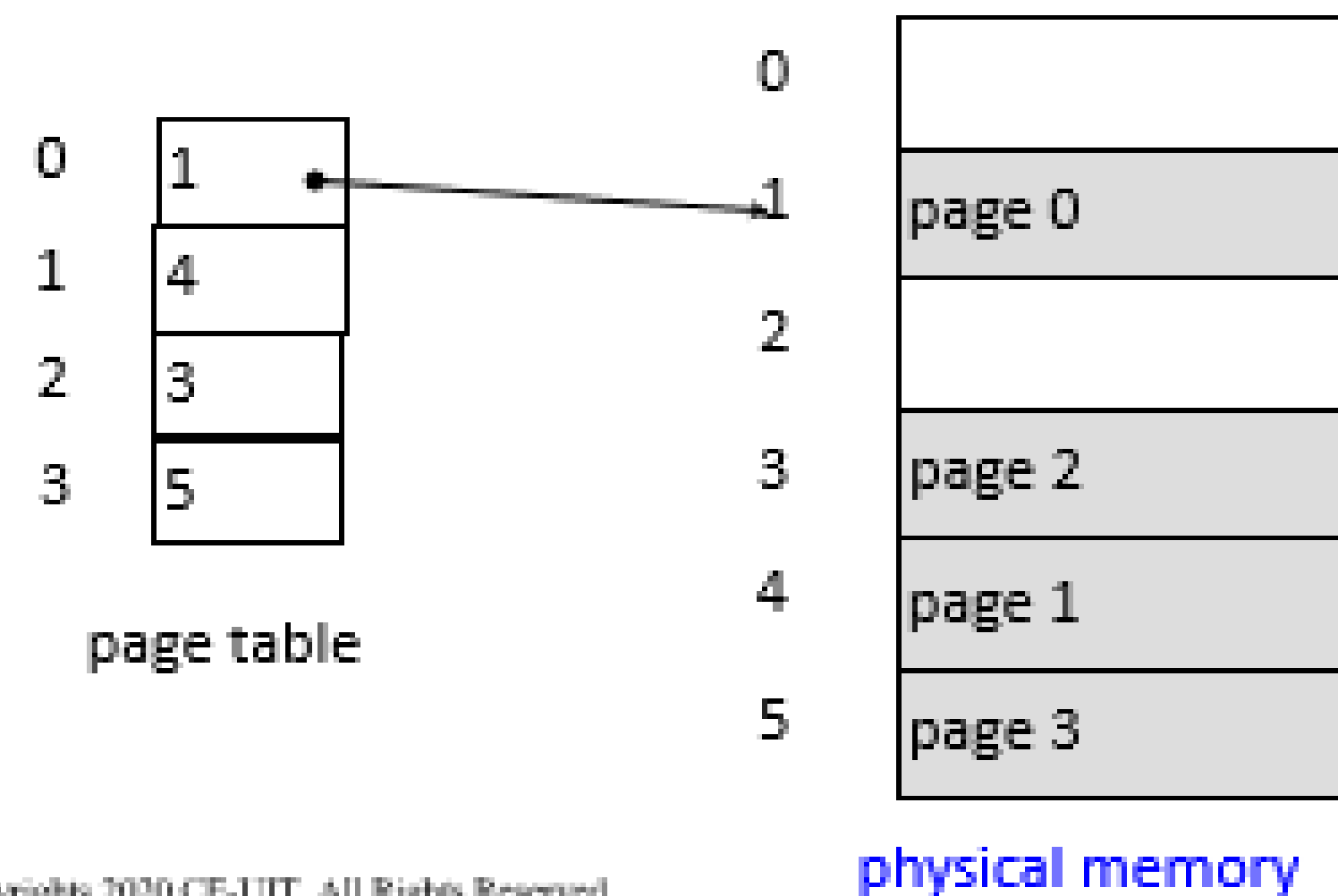
Cơ chế phân trang



Sharing is learning

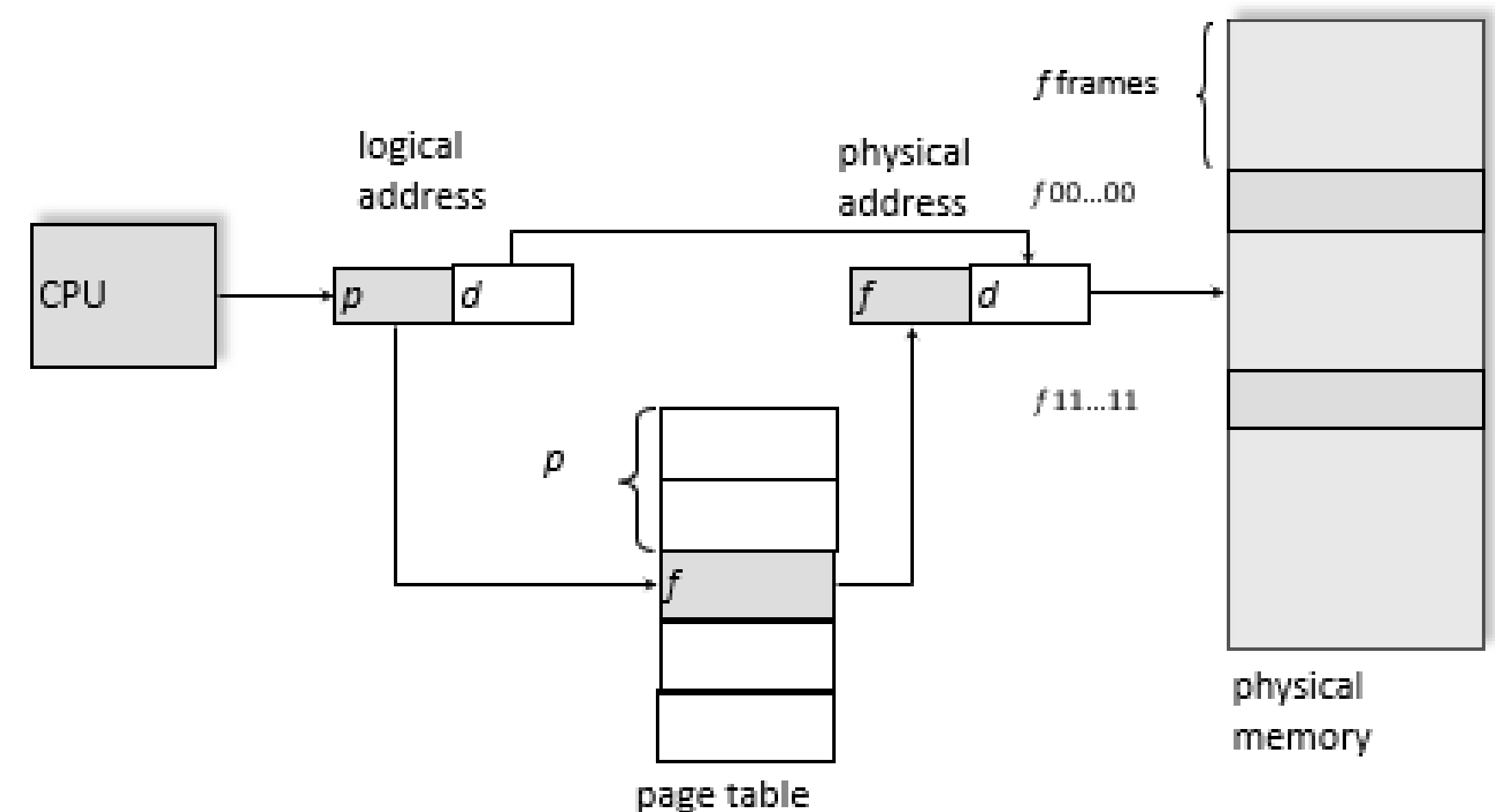
Chuyển đổi địa chỉ trong paging

- Mỗi page sẽ ứng với một frame và được tìm kiếm thông qua page table



Copyrights 2020 CE-UIT. All Rights Reserved.

Bảng phân trang (page table)



Chuyển đổi địa chỉ thông qua page table

Cơ chế phân trang



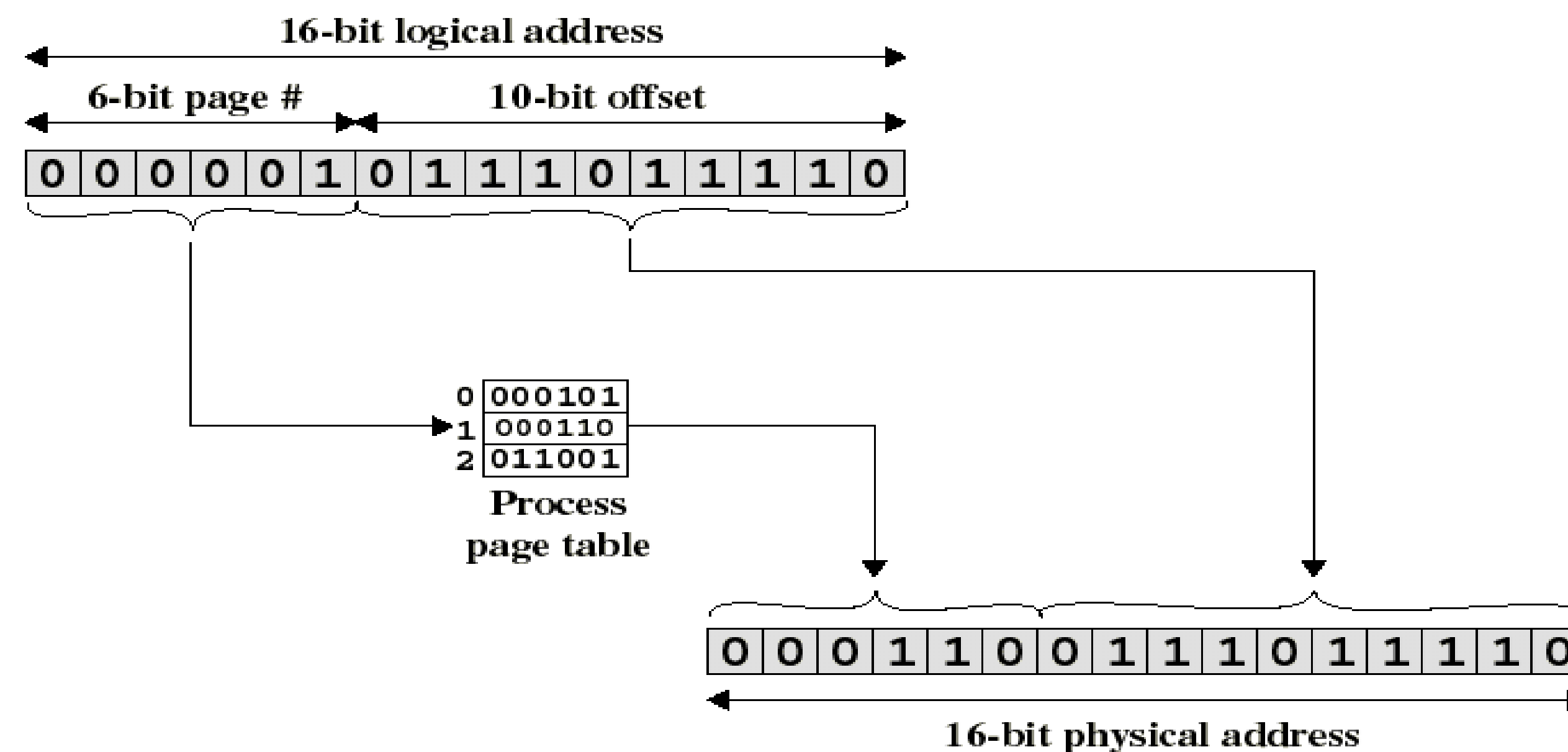
Sharing is learning

Ví dụ chuyển đổi địa chỉ trong paging

Nếu kích thước của không gian địa chỉ ảo là 2^{16} và kích thước của trang là 2^{10} . Hãy chuyển đổi địa chỉ luận lý sau sang 16 bit địa chỉ vật lý.

0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 0

Lời giải:



Câu hỏi trắc nghiệm

Xét một không gian địa chỉ có 8 trang, mỗi trang có kích thước 1KB, ánh xạ vào bộ nhớ vật lý có 32 khung trang.

- a) Địa chỉ logic gồm bao nhiêu bit?
- b) Địa chỉ physic gồm bao nhiêu bit?
- c) Bảng trang có bao nhiêu mục? Mỗi mục trong bảng trang cần bao nhiêu bit?

Giải:

- a) Địa chỉ logic: $\text{page} + \text{offset} = 3 + 10 = 13$
- b) Địa chỉ vật lý: $\text{frame} + \text{offset} = 5 + 10 = 15$
- c) Số mục của bảng trang = số trang = 8. Mỗi mục trong bảng trang cần 5 bit.

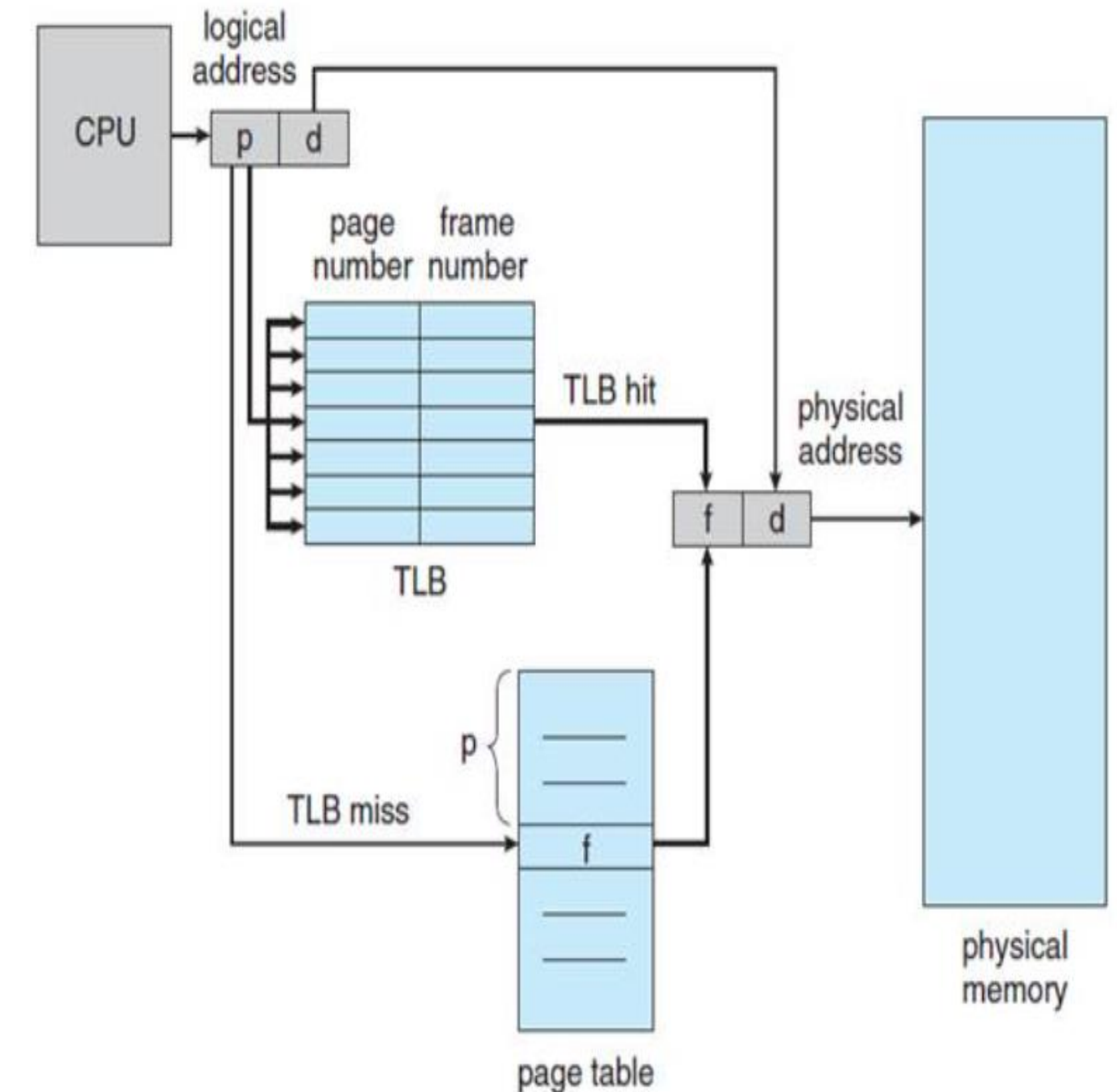
TLB (Translation Look-aside Buffer)



Sharing is learning

Translation look-aside buffers (TLBs)

- Là một bộ phận cache phần cứng có tốc độ truy xuất và tìm kiếm cao.
- Cấu tạo:
 - + Một cột page number (số trang của bộ nhớ luận lý).
 - + Một cột frame number.
- Số mục của TLB nằm trong khoảng từ 32 – 1024 mục.
- Công dụng: TLB được dùng để tìm kiếm nhanh frame number và từ đó cho ra địa chỉ thực (physical address) từ địa chỉ luận lý.



TLB (Translation Look-aside Buffer)



Sharing is learning

Effective access time (EAT)

Thời gian truy xuất hiệu dụng (Effective Access Time, EAT)

- Thời gian tìm kiếm trong TLB: ϵ
- Thời gian truy xuất bộ nhớ: x
- Hit ratio: tỉ số giữa số lần chỉ số trang được tìm thấy (hit) trong TLB và số lần truy xuất khởi nguồn từ CPU.
 - + Kí hiệu hit ratio: α

Thời gian cần thiết để có được chỉ số frame:

- Khi chỉ số trang có trong TLB (hit): $\epsilon + x$
- Khi chỉ số trang không có trong TLB (miss): $\epsilon + x + x$

→ Thời gian truy xuất hiệu dụng:

$$\begin{aligned} \text{EAT} &= (\epsilon + x) \alpha + (\epsilon + 2x)(1 - \alpha) \\ &= (2 - \alpha) x + \epsilon \end{aligned}$$

TLB (Translation Look-aside Buffer)



Sharing is learning

Ví dụ 1:

- Associative lookup = 20ns
- Memory access = 100ns
- Hit ratio = 0.8
- $EAT = (\epsilon + x) \alpha + (\epsilon + 2x)(1 - \alpha)$
 $= (20 + 100) * 0.8 + (20 + 200) * 0.2$
 $= 140ns$

Ví dụ 2:

- Associative lookup = 20ns
- Memory access = 100ns
- Hit ratio = 0.98
- $EAT = (\epsilon + x) \alpha + (\epsilon + 2x)(1 - \alpha)$
 $= (20 + 100) * 0.98 + (20 + 200) * 0.02$
 $= 122ns$

TLB (Translation Look-aside Buffer)



Sharing is learning

Bài tập:

Xét một hệ thống sử dụng kỹ thuật phân trang, với bảng trang được lưu trữ trong bộ nhớ chính. Thời gian cho một lần truy xuất bộ nhớ bình thường là 200ns.

- a) Nếu không dung TLB, tổng cộng mất bao nhiêu thời gian cho một thao tác tìm dữ liệu trong hệ thống này?
- b) Nếu sử dụng TLB với hit ratio (tỷ lệ tìm thấy) là 75% thời gian để tìm trong TLB xem như bằng 0, tính thời gian truy xuất bộ nhớ trong hệ thống (effective memory reference time).

Giải:

- a) Không dung TLB, để tìm một dữ liệu khi biết địa chỉ luận lý của nó cần 2 thao tác truy xuất bộ nhớ. Mỗi lần truy xuất bộ nhớ tốn 200ns \rightarrow 2 thao tác truy xuất bộ nhớ tốn $2 * 200 = 400$ ns
- a) $EAT = (0 + 200) * 0.75 + (0 + 200 + 200) * 0.25 = 250$ ns

7. Phân đoạn theo yêu cầu:

Các thanh ghi được sử dụng trong phân đoạn:

- Segment – Table base register (STBR): Trỏ đến vị trí bảng phân đoạn trong bộ nhớ.
 - Segment – Table length register (STLR): Số lượng segment của chương trình.
- Một chỉ số segment s là hợp lệ nếu $s < \text{SLTR}$.

Chuyển đổi địa chỉ theo phân đoạn

Mỗi địa chỉ ảo là một bộ $\langle s, d \rangle$:

Số hiệu phân đoạn s : được sử dụng như chỉ mục đến bảng phân đoạn.

Địa chỉ tương đối d : có giá trị trong khoảng từ 0 đến giới hạn chiều dài của phân đoạn.

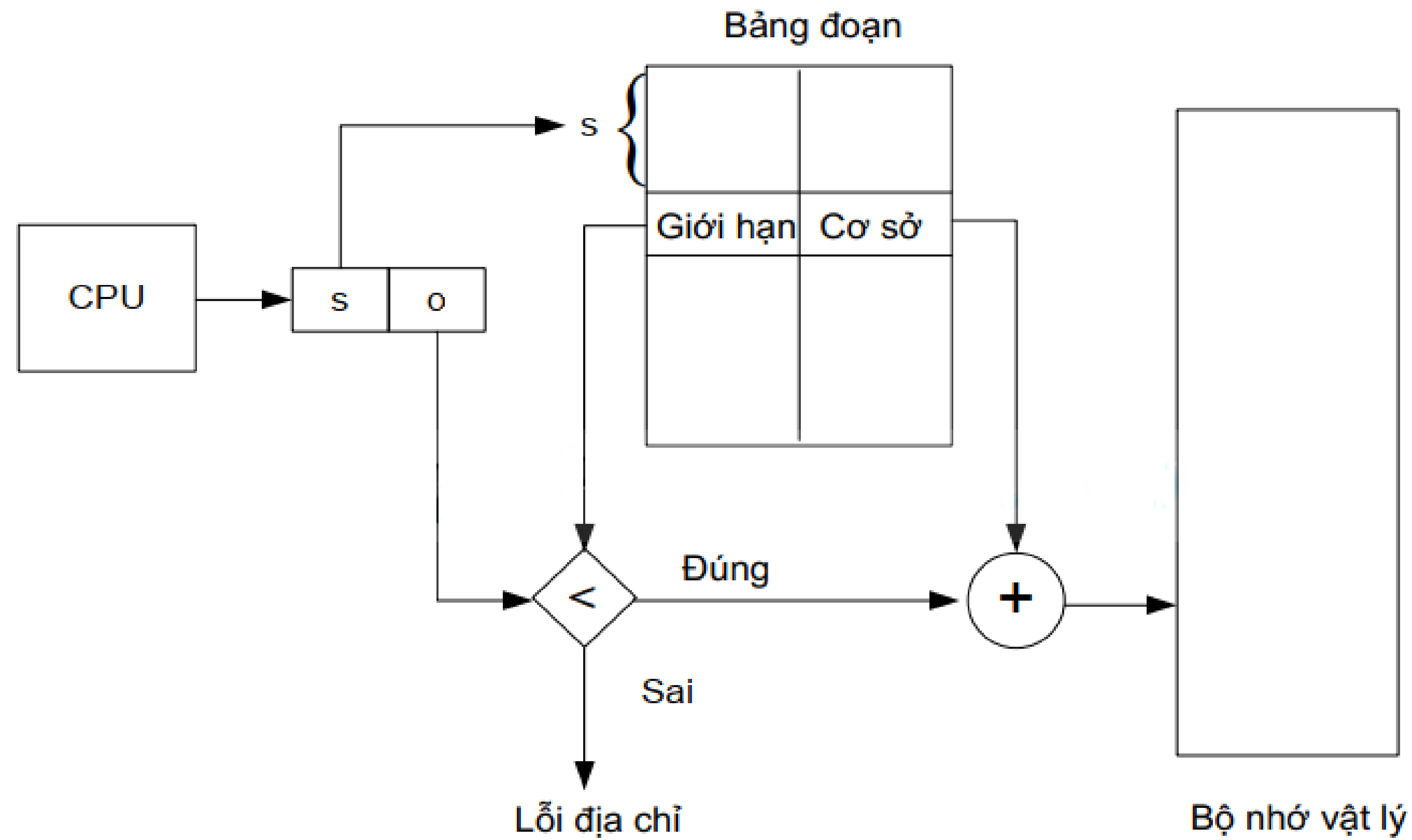
Nếu địa chỉ tương đối hợp lệ, nó sẽ được cộng với giá trị chứa trong thanh ghi nền để phát sinh địa chỉ vật lý tương ứng.

Bộ nhớ ảo



Sharing is learning

Chuyển đổi địa chỉ theo phân đoạn



Bộ nhớ ảo

Ví dụ:

Cho bảng phân đoạn sau. Hãy cho biết địa chỉ vật lý tương ứng với các địa chỉ luận lý sau:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

- a) 0, 430
- b) 1, 10
- c) 2, 500

Giải:

- a) Dữ liệu đang ở đoạn 0 và vị trí thứ 430 trong đoạn 0.
Đoạn 0 có length = 600, $430 < 600 \rightarrow$ Địa chỉ hợp lý.
Địa chỉ vật lý tương ứng = $219 + 430 = 649$
- b) Địa chỉ vật lý là 2310.
- c) Không có địa chỉ vật lý tương ứng.



Sharing is learning

4. Bộ nhớ ảo

Nội dung Training



Sharing is learning

I. Tổng quan

II. Cài đặt bộ nhớ ảo

III. Phân trang theo yêu cầu

IV. Giải thuật thay trang

V. Vấn đề cấp phát Frames

VI. Vấn đề Thrashing

1. Tổng quan

Bộ nhớ ảo (virtual memory): Bộ nhớ ảo là một kỹ thuật cho phép xử lý một tiến trình không được nạp toàn bộ vào bộ nhớ vật lý.

Ưu điểm:

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

Lựa chọn nào dưới đây KHÔNG phải là ưu điểm của bộ nhớ ảo?

(Đề CK 2018 – 2019)

- A. Số lượng tiến trình trong bộ nhớ nhiều hơn.
- B. Một tiến trình có thể thực thi ngay cả khi kích thước của nó lớn hơn bộ nhớ thực.
- ☒ C. Giảm thời gian truy xuất bộ nhớ.
- D. Giảm nhẹ công việc của lập trình viên.

2. Cài đặt bộ nhớ ảo

Có hai kỹ thuật

- Phân trang theo yêu cầu (Demand Paging)
- Phân đoạn theo yêu cầu (Demand Segmentation)

Phần cứng memory management phải hỗ trợ paging và/hoặc segmentation

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

3. Phân trang theo yêu cầu (Demand paging)

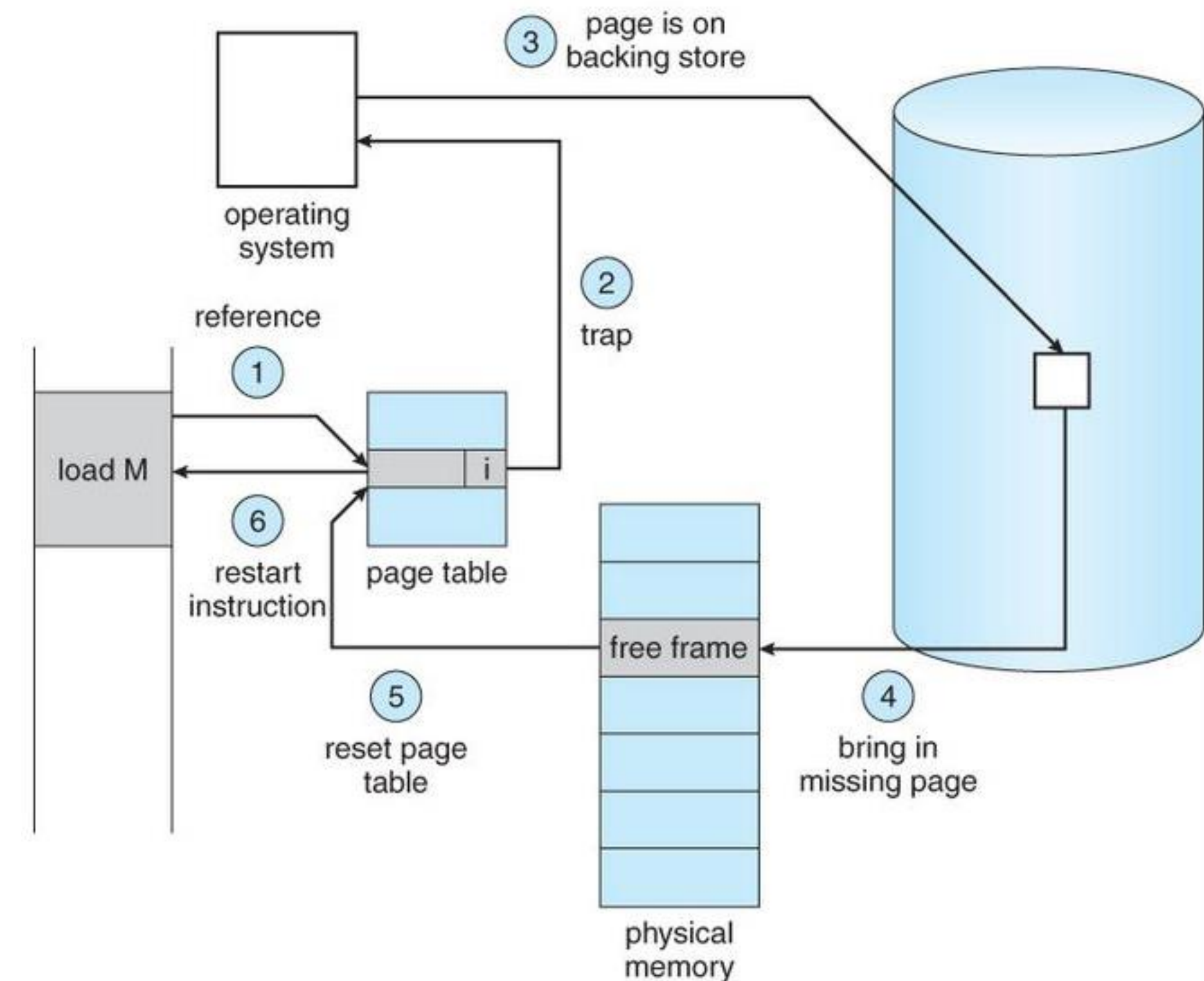
Các trang của tiến trình chỉ được nạp vào bộ nhớ chính khi được yêu cầu.

Page-fault: khi tiến trình tham chiến đến một trang không có trong bộ nhớ chính (valid bit) thì phần cứng sẽ gây ra một lỗi trang (page-fault).

Khi có page-fault thì phần cứng sẽ gây ra một ngắt (page-fault trap) kích khởi page-fault service routine (PFSR).

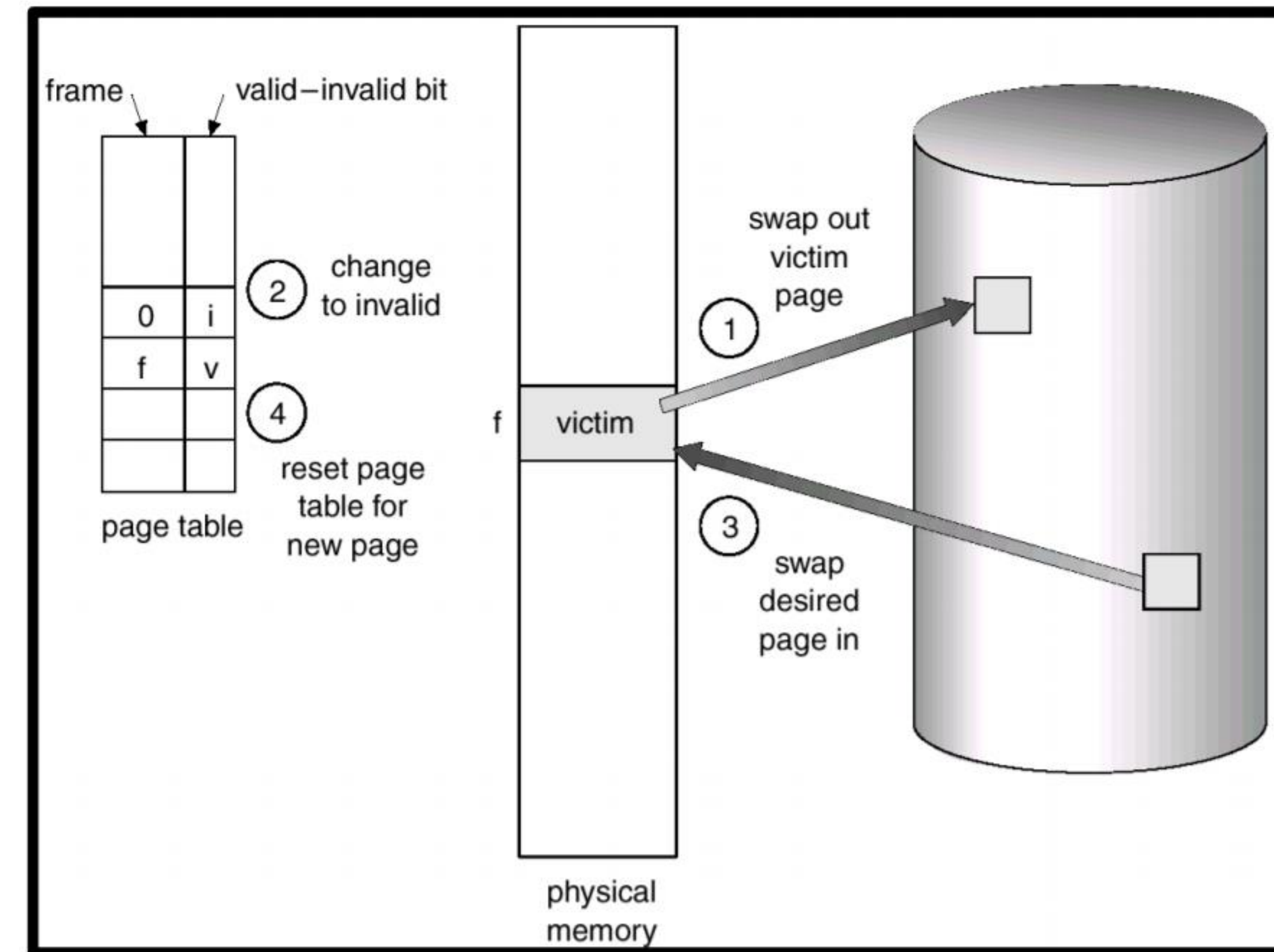
PFSR

- Chuyển process về trạng thái blocked
- Phát ra một yêu cầu đọc đĩa để nạp trang được tham chiếu vào một frame trống; trong khi đợi I/O, một process khác được cấp CPU để thực thi
- Sau khi I/O hoàn tất, đĩa gây ra một ngắt đến hệ điều hành; PFSR cập nhật page table và chuyển process về trạng thái ready.



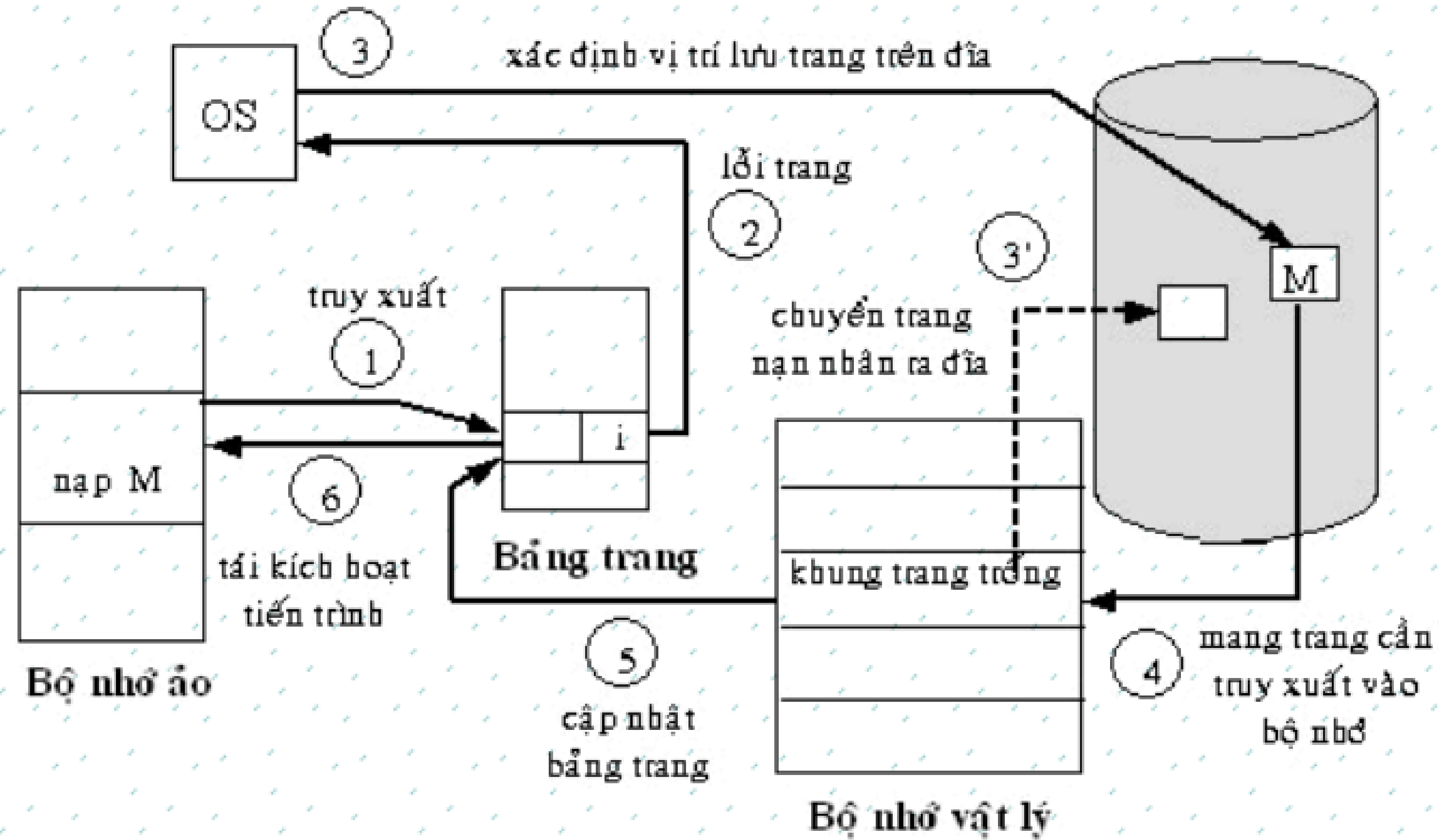
PFSR

- Xác định vị trí trên đĩa của trang đang cần.
- Tìm một frame trống
 - Nếu có frame trống thì dùng nó.
 - Nếu không có frame trống thì dùng một giải thuật thay trang để chọn một trang hi sinh (victim page).
- Đọc trang đang cần vào frame trống, cập nhật page table và frame table tương ứng.



Bộ nhớ ảo

PFSR



Hình 2.26 Các giai đoạn xử lý lỗi trang



4. Giải thuật thay trang

Dữ liệu cần biết:

- Số khung trang
- Tình trạng ban đầu (thường là trống)
- Chuỗi tham chiếu

4. Giải thuật thay trang

Ví dụ: Xét chuỗi truy xuất bộ nhớ sau

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

Giả sử có 4 khung trang và ban đầu các khung trang **đều trống**, có bao nhiêu lỗi trang xảy ra?

4. Giải thuật thay trang

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

Giải thuật **FIFO** (first-in, first-out)

	1	2	3	4	3	5	1	6	2	1	2	3	7	5	3	2	1	2	3	6
0	1	1	1	1	1	5	5	5	5	5	2	2	2	2	2	2	1	1	1	1
1		2	2	2	2	2	1	1	1	1	1	3	3	3	3	3	3	2	2	2
2			3	3	3	3	3	6	6	6	6	6	7	7	7	7	7	7	3	3
3				4	4	4	4	4	2	4	4	4	4	5	5	5	5	5	5	6
*	*	*	*	*		*	*	*	*		*	*	*	*			*	*	*	*

⇒ 16 page-fault

4. Giải thuật thay trang

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

Giải thuật **OPT** (optimal)

	1	2	3	4	3	5	1	6	2	1	2	3	7	5	3	2	1	2	3	6
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	6
1		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2			3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3				4	4	5	5	6	6	6	6	6	7	5	5	5	5	5	5	5
*	*	*	*	*		*		*					*	*						*

x

⇒ 9 page-fault

4. Giải thuật thay trang

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

Giải thuật **LRU** (Least Recently Used)

	1	2	3	4	3	5	1	6	2	1	2	3	7	5	3	2	1	2	3	6
0	1	1	1	1	1	5	5	5	5	5	5	3	3	3	3	3	3	3	3	3
1		2	2	2	2	2	1	1	1	1	1	1	1	5	5	5	5	5	5	5
2			3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	2	6
3				4	4	4	4	6	6	6	6	6	7	7	7	7	1	1	1	1
*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

⇒ 13 page-fault

5. Vấn đề cấp phát Frames

OS phải quyết định cấp cho mỗi process bao nhiêu frame.

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

5. Vấn đề cấp phát Frames

Chiến lược cấp phát tĩnh (fixed-allocation)

Số frame cấp cho mỗi process không đổi, được xác định vào thời điểm loading và có thể tùy thuộc vào từng ứng dụng (kích thước của nó,...)

Chiến lược cấp phát động (variable-allocation)

- ❖ Số frame cấp cho mỗi process có thể thay đổi trong khi nó chạy
 - Nếu tỷ lệ page-fault cao \Rightarrow cấp thêm frame
 - Nếu tỷ lệ page-fault thấp \Rightarrow giảm bớt frame
- ❖ OS phải mất chi phí để ước định các process

Bộ nhớ ảo



Sharing is learning

Trong kỹ thuật cài đặt bộ nhớ ảo sử dụng phân trang theo yêu cầu, khi sử dụng chiến lược cấp phát động, số lượng khung trang (frame) được cấp cho một tiến trình sẽ thay đổi như thế nào nếu tỷ lệ lỗi trang (page fault) cao?

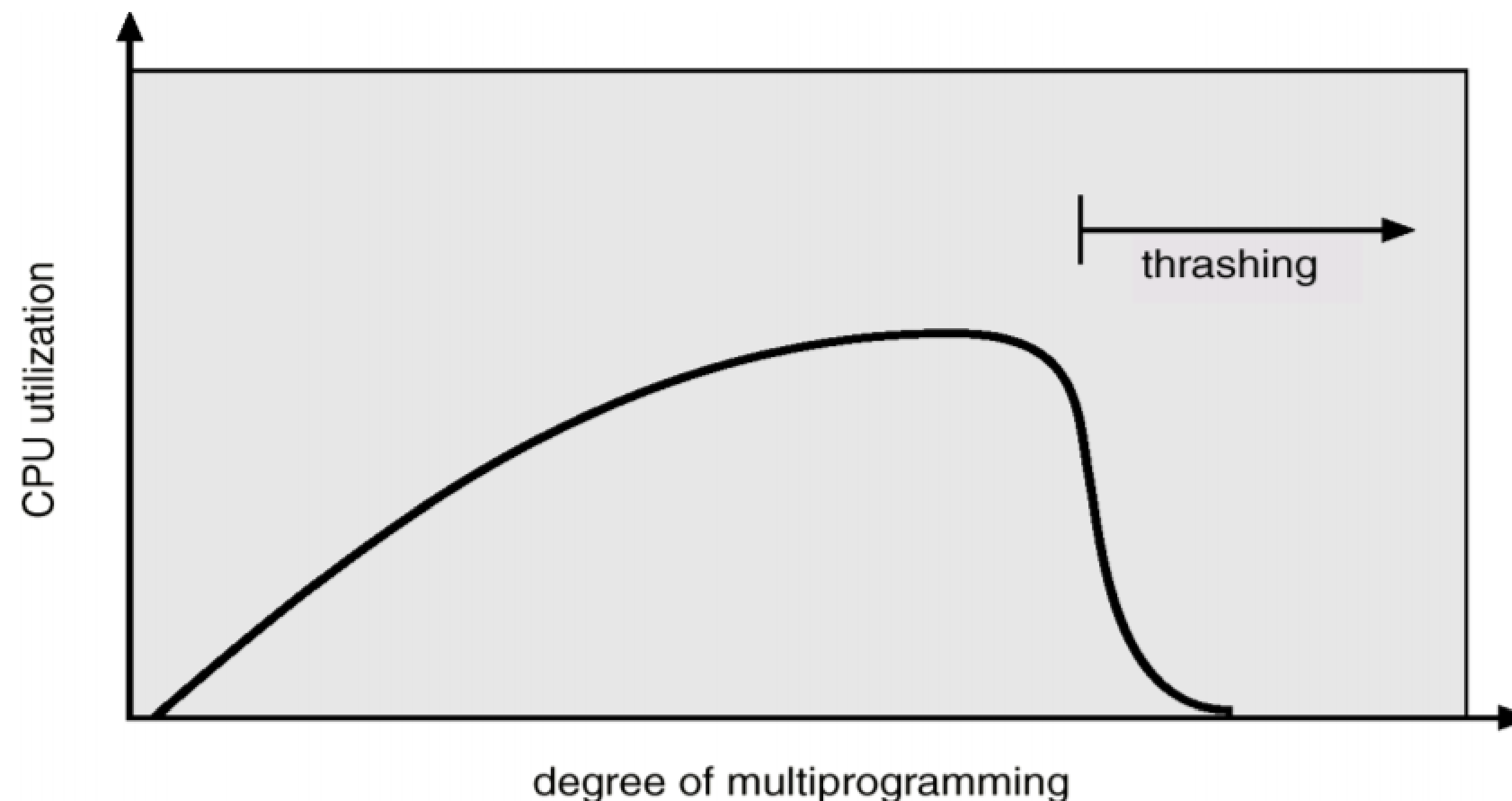
(Đề CK 2018 – 2019)

- A. Giảm xuống
- ☒ B. Tăng lên
- C. Không thay đổi
- D. Bị hệ thống thu hồi toàn bộ

6. Vấn đề Thrashing

Nếu một process không có đủ số frame cần thiết thì tỉ số page faults/sec rất cao.

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



BAN HỌC TẬP KHOA CÔNG NGHỆ PHẦN MỀM

CHUỖI TRAINING CUỐI HỌC KÌ I NĂM HỌC 2021 - 2022



Sharing is learning

HẾT

**CẢM ƠN CÁC BẠN ĐÃ THEO DÕI.
CHÚC CÁC BẠN CÓ KẾT QUẢ THI THẬT TỐT!**



Ban học tập

Khoa Công Nghệ Phần Mềm
Trường ĐH Công Nghệ Thông Tin
ĐHQG Hồ Chí Minh



Email / Group

bht.cnpm.uit@gmail.com
fb.com/groups/bht.cnpm.uit
<https://www.facebook.com/bhtcnpm>