BAN HỌC TẬP KHOA HỆ THỐNG THÔNG TIN

HỆ ĐIỀU HÀNH

Trainer

Bùi Hữu Nghĩa

Huỳnh Văn Thoại

CHƯƠNG 5: ĐỒNG BỘ

I. Vấn đề đồng bộ

II. Các giải pháp "BUSY WAITING"

III. Các giải pháp "SLEEP AND WAKEUP"

I. VẤN ĐỀ

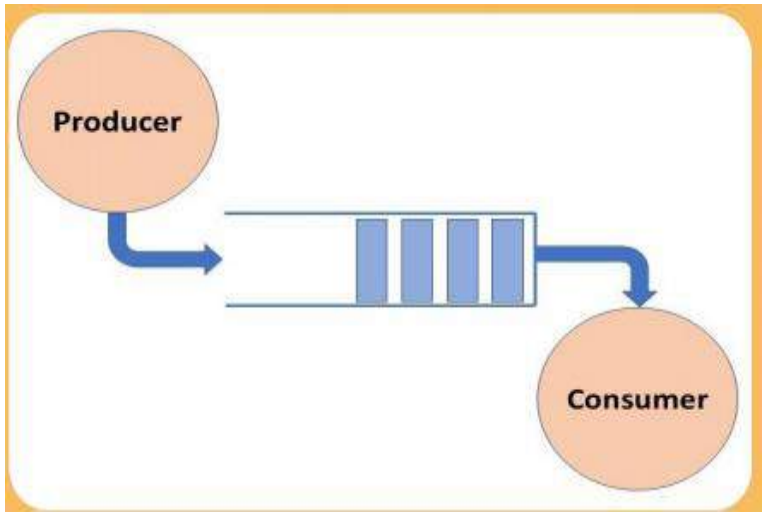
- Khảo sát các process/thread thực thi đồng thời và chia sẻ dữ liệu (qua shared memory. file).

- Nếu không có sự kiểm soát khi truy cập các dữ liệu chia sẻ thì có thể đưa đến trường hợp không nhất quán dữ liệu (data inconsistency).

- Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế đảm bảo sự thực thi có trật tự của các process đồng thời.

BÀI TOÁN PRODUCER - CONSUMER

- **P** không được ghi dữ liệu vào buffer đã đầy
- **C** không được đọc dữ liệu từ buffer đang trống
- **P** và **C** không được thao tác trên buffer cùng lúc



QUÁ TRÌNH PRODUCER

```
item nextProduce;
while(1){
 while(count == BUFFER_SIZE);
  /*khong lam gi*/
 buffer[in] = nextProducer;
 count++;
 in = (in+1)%BUFFER_SIZE;}
```

QUÁ TRÌNH CONSUMER

```
item nextConsumer;
while(1){
 while(count == 0);
  /*khong lam gi*/
 nextConsumer = buffer[out];
 count--;
 out = (out+1)%BUFFER_SIZE;}
```

Vấn đề bài toán: Khi chúng ta cho phép cả hai quá trình *Producer* và *Consumer* thao tác đồng thời trên biến count thì sẽ dẫn đến trạng thái không đúng (trình trạng vùng đệm đầy, hoặc trống).Tương tự nếu có nhiều quá trình truy xuất và thao tác cùng dữ liệu đồng thời thì kết quả sẽ phụ thuộc vào thứ tự thực thi.

Critical Section ~ CS

Bài toán đặt ra: Một hệ thống gồm n quá trình (P0, P1, ..Pn-1). Mỗi process Pi có đoạn code:

```
while(1){
        entry section

                critical section

        exit section

                remainder section
}
```

Critical section là đoạn mã chứa các thao tác lên dữ liệu chia sẻ
trong mỗi tiến trình.(biến count ví dụ trên là một CS)

Vấn đề: Thiết kế một giao thức mà các process có thể dùng để cộng tác, mỗi process phải yêu cầu quyền để đi vào vùng tranh chấp của nó.

Điều kiện giải quyết bài toán CS,thỏa mãn 3 tính chất

• Loại trừ tương hỗ (Mutual exclution): Nếu process Pi đang thực thi trong vùng tranh chấp của nó thì không process nào khác được thực thi trong vùng tranh chấp đó

• Tiến trình (Progress): Một tiến trình tạm dừng bên ngoài vùng tranh chấp không được ngăn cản các tiến trình khác vào vùng tranh chấp

• Chờ đợi có 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 trình trạng đói tài nguyên(starvation)

Busy Waiting

- Sử dụng các biến cờ hiệu
- Sử dụng việc kiểm tra luân phiên
- Giải pháp của Peterson
- Cấm ngắt
- Chỉ thị TSL

Sleep & Wakeup

- Semaphore
- Monitor
- Message

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

Sleep & Wakeup

- Từ bỏ CPU khi chưa được vào vùng tranh chấp
- Cần Hệ điều hành hỗ trơ

II. CÁC GIẢI PHÁP "BUSY WAITING"

Các giải pháp phần mềm

- Sử dụng việc kiểm tra luân phiên
- Sử dụng các biến cờ hiệu
- Giải pháp của Peterson

Các giải pháp phần cứng

- Cấm ngắt
- Chỉ thị TSL

CÁC GIẢI PHÁP PHẦN MỀM

Giải thuật 1: Sử dụng việc kiểm tra luân phiên

Biến chia sẻ:

int turn; /*khởi đầu turn = 0*/

Nếu turn = i thì Pi được phép vào CS, với i = 0 hoặc 1

- Process Pi:

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

- Thỏa mãn **Mutual exclusion (1)**
- Nhưng **không** thoả mãn yêu cầu về **progress (2)** và **bounded waiting (3)**

Giải thuật 2: Sử dụng các biến cờ hiệu

Biến chia sẻ:

Boolean flag[2]; /*khởi đầu flag[0] = flag[1] = false*/

Nếu flag[i] = true thì Pi "sẵn sàng" vào CS

- Process Pi:

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

- Thỏa mãn **Mutual exclusion (1)**
- Nhưng **không** thoả mãn yêu cầu về **progress (2)**

Giải thuật 2: Giải pháp của Peterson

Biến chia sẻ:

Kết hợp giải thuật 1 và 2

- Process Pi:

```
do {
flag[i] = true; /* Pi "sẵn sàng" */
Turn = j;       /* Pi "nhường" Pj*/
          while ( flag[j] and turn == j );
    critical section
flag[i] = false;
    remainder section
} while (1);
```

- Thỏa mãn **Cả 3 yêu cầu (1)(2)(3)**

Giải thuật Bakery: n process

Trước khi vào CS, process Pi nhận một con số. Process nào giữ con số nhỏ nhất thì được vào CS

- Trường hợp Pi và Pj cùng nhận được một chỉ số:
- Nếu i < j thì Pi được vào trước. (Đối xứng)
- Khi ra khỏi CS, Pi đặt lại số của mình bằng 0
- Cơ chế cấp số cho các process thường tạo các số theo cơ chế tăng dần, ví dụ 1, 2, 3, 3, 3, 3, 4, 5,…
- Kí hiệu
- (a,b) < (c,d) nếu a < c hoặc nếu a = c và b < d
- max(a0,…,ak) là con số b sao cho b ≥ ai với mọi i = 0,…, k

```
/* shared variable */
boolean choosing[ n ]; /*initially, choosing[ i ] = false */
int num[ n ];
/* initially, num[ i ] = 0 */
do {
   choosing[ i ] = true;
   num[ i ] = max(num[0], num[1],…, num[n - 1]) + 1;
   choosing[ i ] = false;
   for (j = 0; j < n; j++) {
      while (choosing[ j ]);
      while ((num[ j ] != 0) && (num[ j ], j) < (num[ i ], i));
   }
      critical section
   num[ i ] = 0;
      remainder section
}while (1);
```

Khuyết điểm của các giải pháp phần mềm

- Các process khi yêu cầu được vào vùng tranh chấp đều phải liên tục kiểm tra điều kiện (busy waiting), tốn nhiều thời gian xử lý của CPU
- Nếu thời gian xử lý trong vùng tranh chấp lớn, một giải pháp hiệu quả nên có cơ chế block các process cần đợi.

CÁC GIẢI PHÁP PHẦN CỨNG

Cấm ngắt

- Trong hệ thống uniprocessor: mutual exclusion được đảm bảo
  - o Nhưng nếu system clock được cập nhật do interrupt thì…
- Trong hệ thống multiprocessor: mutual exclusion không được đảm bảo
  - oChỉ cấm ngắt tại CPU thực thi lệnh disable_interrupts
  - oCác CPU khác vẫn có thể truy cập bộ nhớ chia se

Process Pi:

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

Lệnh TestAndSet

- Lệnh này đọc và ghi một biến trong một thao tác atomic (không chia cắt được)
- Định nghĩa:

❑ Shared data:

```
boolean  lock = false;
```

❑ Process P_i:

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

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

Thỏa mãn **Mutual exclusion (1)**
Nhưng khi Pi ra khỏi CS, quá trình chọn lựa process Pj vào CS kế tiếp là tùy ý ⇒ không bảo đảm điều kiện bounded waiting(3). Do đó có thể xảy ra starvation (bị bỏ đói).

Swap và mutual exclusion

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

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

- Biến chia sẻ (khởi tạo là **false**)

```
bool  lock;
bool  key;
```

- Process P_i

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

Không thõa mãn Bounded Waiting(3)

Giải thuật dùng TestAndSet thoả mãn 3 yêu cầu

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

Đặc điểm:

- Biến chia sẻ(khởi tạo false)
  - o bool *waiting[n]*
  - o bool *lock*

```
do
{
    waiting[i] = true;
    key = true;
    while (waiting{i] && key)
        Swap(&lock, &key);
    waiting[i] = false;
    /critical section/
    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /remainder section/
} while(1);
```

III. CÁC GIẢI PHÁP "SLEEP AND WAKEUP"

Các giải pháp “Sleep & Wake up”

- Semaphore
- Critical Region
- Monitor

Semaphore

- Là công cụ đồng bộ cung cấp bởi OS mà không đòi hỏi busy waiting
- Semaphore S là một biến số nguyên.
- Ngoài thao tác khởi động biến thì chỉ có thể được truy xuất qua hai tác vụ có tính đơn nguyên (atomic) và loại trừ (mutual exclusive):
  - wait(S) hay còn gọi là P(S): giảm giá trị semaphore (S=S-1) . Kế đó nếu giá trị này âm thì process thực hiện lệnh wait() bị blocked.
  - signal(S) hay còn gọi là V(S): tăng giá trị semaphore (S=S+1) . Kế đó nếu giá trị này không dương, một process đang blocked bởi một lệnh wait() sẽ được hồi phục để thực thi.
- Tránh busy waiting: khi phải đợi thì process sẽ được đặt vào một blocked queue, trong đó chứa các process đang chờ đợi cùng một sự kiện.

Semaphore

Minh họa:

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

Các tác vụ semaphore thực hiện:

Wait

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                sleep();
        }
}
```

Signal

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Semaphore

- Giả sử hệ điều hành cung cấp hai tác vụ (system call):
  - o block(): tạm treo process nào thực thi lệnh này
  - o wakeup(P): hồi phục quá trình thực thi của process P đang blocked

Semaphore

- Khi một process phải chờ trên semaphore S, nó sẽ bị blocked và được đặt trong hàng đợi semaphore
  - Hàng đợi này là danh sách liên kết các PCB
- Tác vụ signal() thường sử dụng cơ chế FIFO khi chọn một process từ hàng đợi và đưa vào hàng đợi ready
- block() và wakeup() thay đổi trạng thái của process
  - block: chuyển từ running sang waiting
  - wakeup: chuyển từ waiting sang ready

Monitors

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

Cấu tạo: Là một module phần mềm, bao gồm:

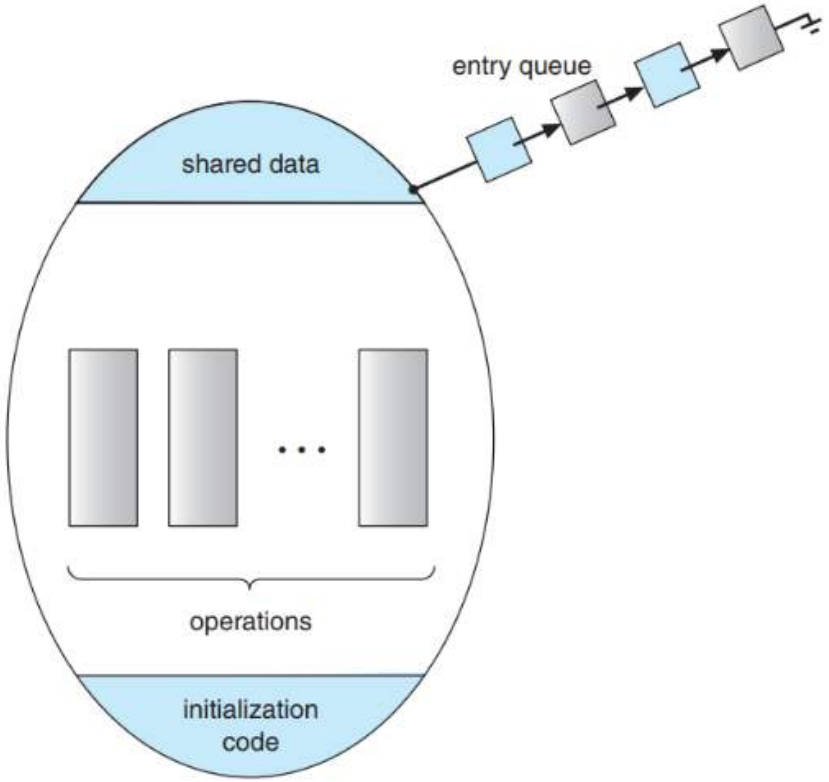
- Một hoặc nhiều thủ tục (procedure)
- Một đoạn code khởi tạo (initialization code)
- Các biến dữ liệu cục bộ (local data variable)

Monitors

Đặc tính:

- Dùng các thủ tục của monitor để truy xuất local variable
- Process "vào monitor" bằng cách gọi một trong các thủ tục đó
- Chỉ có một process có thể vào monitor tại một thời điểm ⇒ mutual exclusion được bảo đảm

Monitors



Mô hình 1 monitor đơn giản

CHƯƠNG 6: DEADLOCKS

I. Khái niệm
II. Mô hình hệ thống
III. Các phương pháp giải quyết deadlock

BAN HỌC TẬP KHOA HỆ THỐNG THÔNG TIN

I. KHÁI NIỆM

TÌNH HUỐNG DEADLOCK

- Tập các tiến trình (từ 2 trở lên) bị block
- Mỗi tiến trình giữ tài nguyên và chờ tài nguyên tiến trình khác giữ
- Ví dụ:
  - Hệ thống có 2 file trên đĩa
  - P1 và P2 mỗi tiến trình mở một file và yêu cầu mở file kia

TIẾN TRÌNH BỊ DEADLOCK

- Đợi một sự kiện không bao giờ xảy ra

TIẾN TRÌNH BỊ STAVATION

- Bị trì hoãn một thời gian dài lặp đi lặp lại trong khi hệ thống đáp ứng cho những tiến trình khác

ĐIỀU KIỆN CẦN ĐỂ XẢY RA DEADLOCK

- Loại trừ tương hỗ (Mutual Exclusion) : có tài nguyên thuộc chế độ không chia sẻ (nonsharable)
- Không trưng dụng (No preemption): tài nguyên không bị lấy lại mà được trả lại

Ví dụ: Tài nguyên R được tiến trình P giữ:

- P chưa thực thi, R không được tiến trình khác trưng dụng
- P thực thi xong, R được trả lại

ĐIỀU KIỆN CẦN ĐỂ XẢY RA DEADLOCK

- Giữ và chờ cấp thêm (Hold and wait): tiến trình giữ tài nguyên và đợi thêm tài nguyên do tiến trình khác đang giữ
- Chu trình đợi (Circular wait): tồn tại một tập các tiến trình, sao cho:
  - P1 đợi tài nguyên mà P2 giữ
  - P2 đợi tài nguyên mà P3 giữ
  - …
  - Pn đợi tài nguyên mà P1 giữ

⇒Deadlock chỉ xảy ra khi 4 điều kiện xảy ra đồng thời

BAN HỌC TẬP KHOA HỆ THỐNG THÔNG TIN

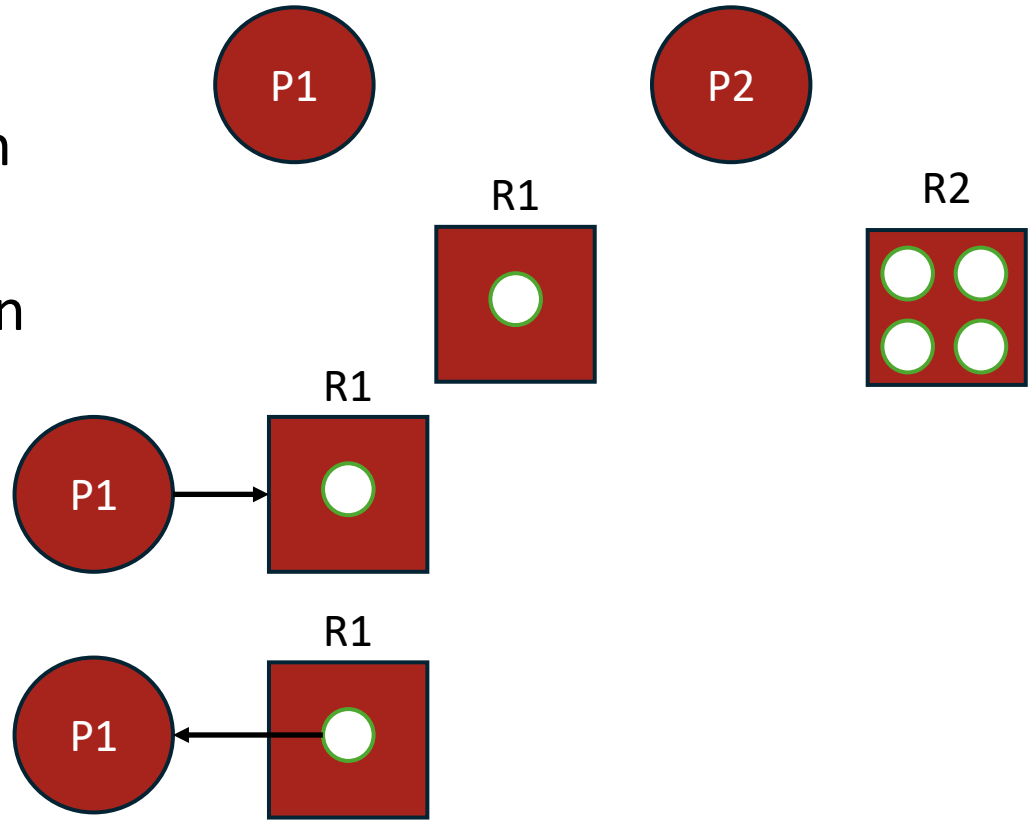
II. MÔ HÌNH HÓA HỆ THỐNG

ĐỂ MÔ HÌNH HÓA HỆ THỐNG

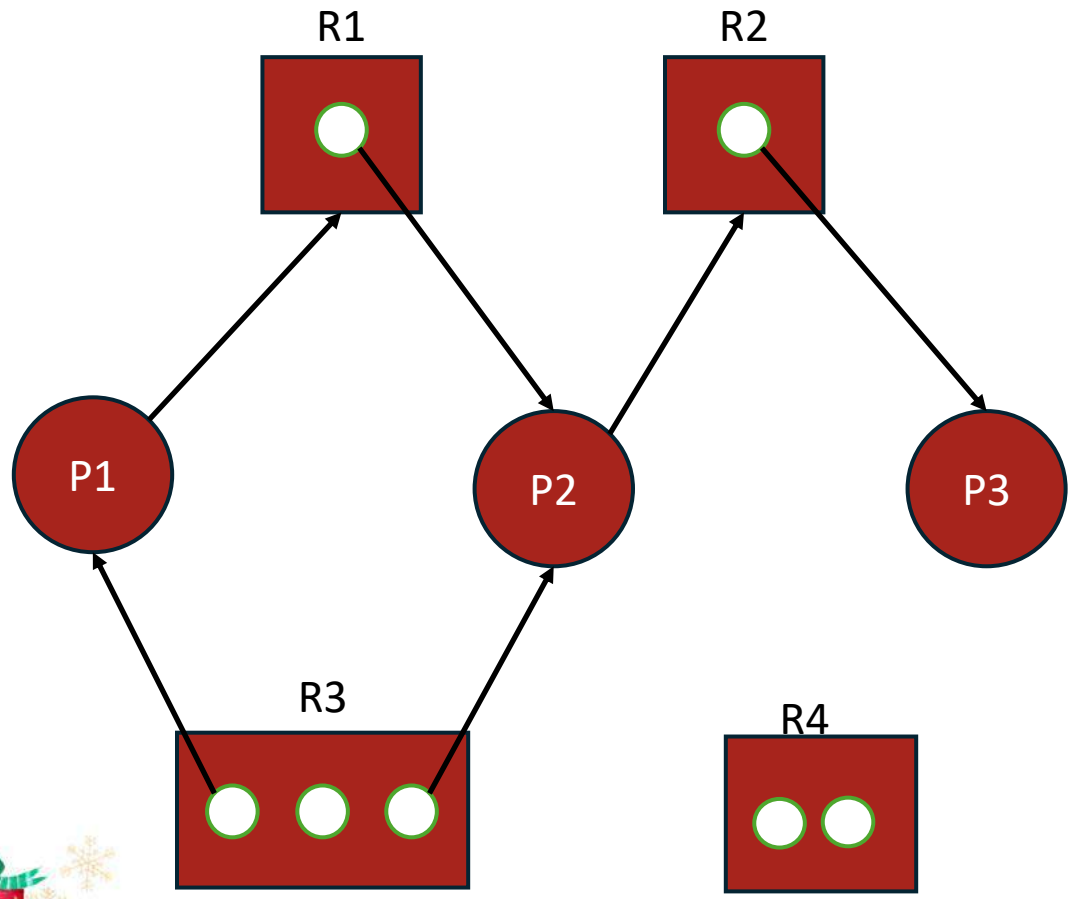
- Hệ thống gồm 1 tập giới hạn các tài nguyên. Kí hiệu R1, R2, … , Rn
- 1 tài nguyên có m thực thể. Kí hiệu là W1, W2, …, Wm
- Tiến trình sử dụng tài nguyên theo trình tự: yêu cầu → sử dụng → hoàn trả

ĐỒ THỊ CẤP PHÁT TÀI NGUYÊN - RAG

- Đỉnh:
  - Tiến trình: P1, P2, …, Pn
  - Tài nguyên R1, R2, …, Rn
- Cạnh:
  - Cạnh yêu cầu:
  - Cạnh cấp phát



Ví dụ 1:



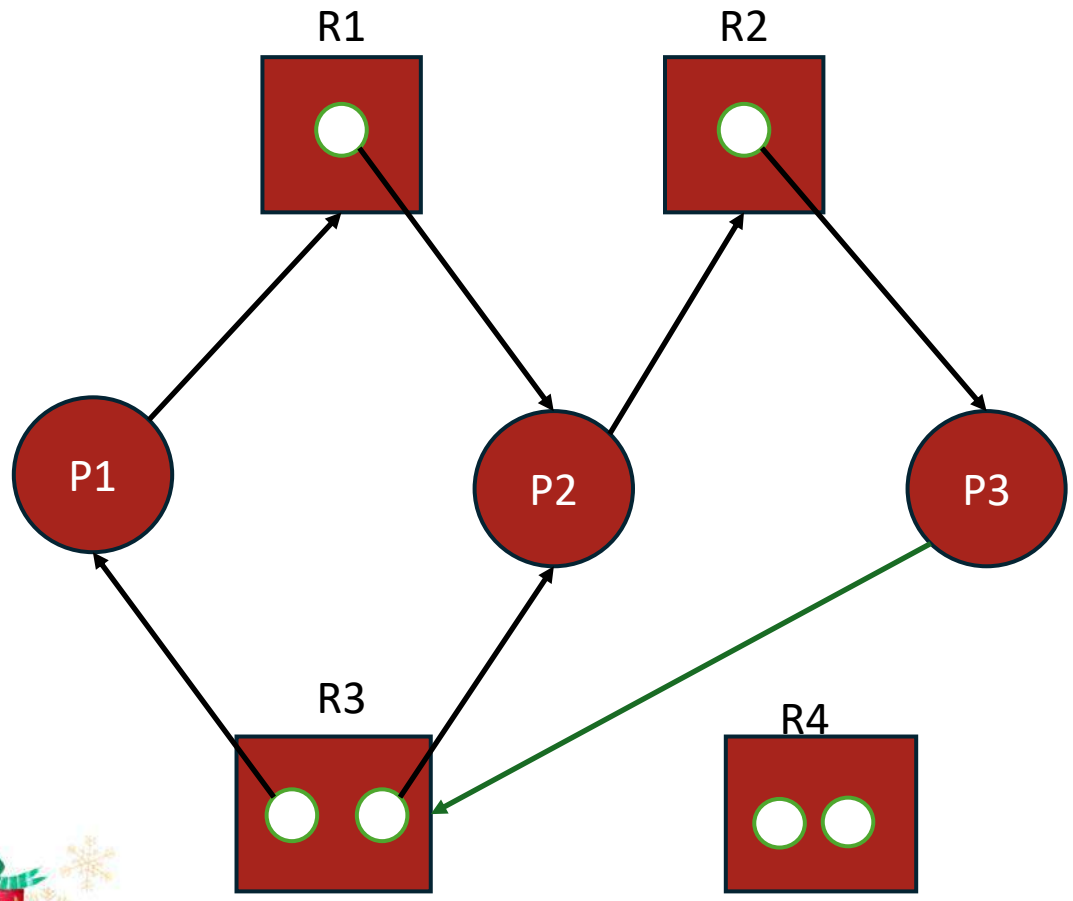
Chu trình: không có

Chuỗi thứ tự thực thi:

P3 → P2 → P1

⇒ không xảy ra deadlock

Ví dụ 2:



Chu trình:

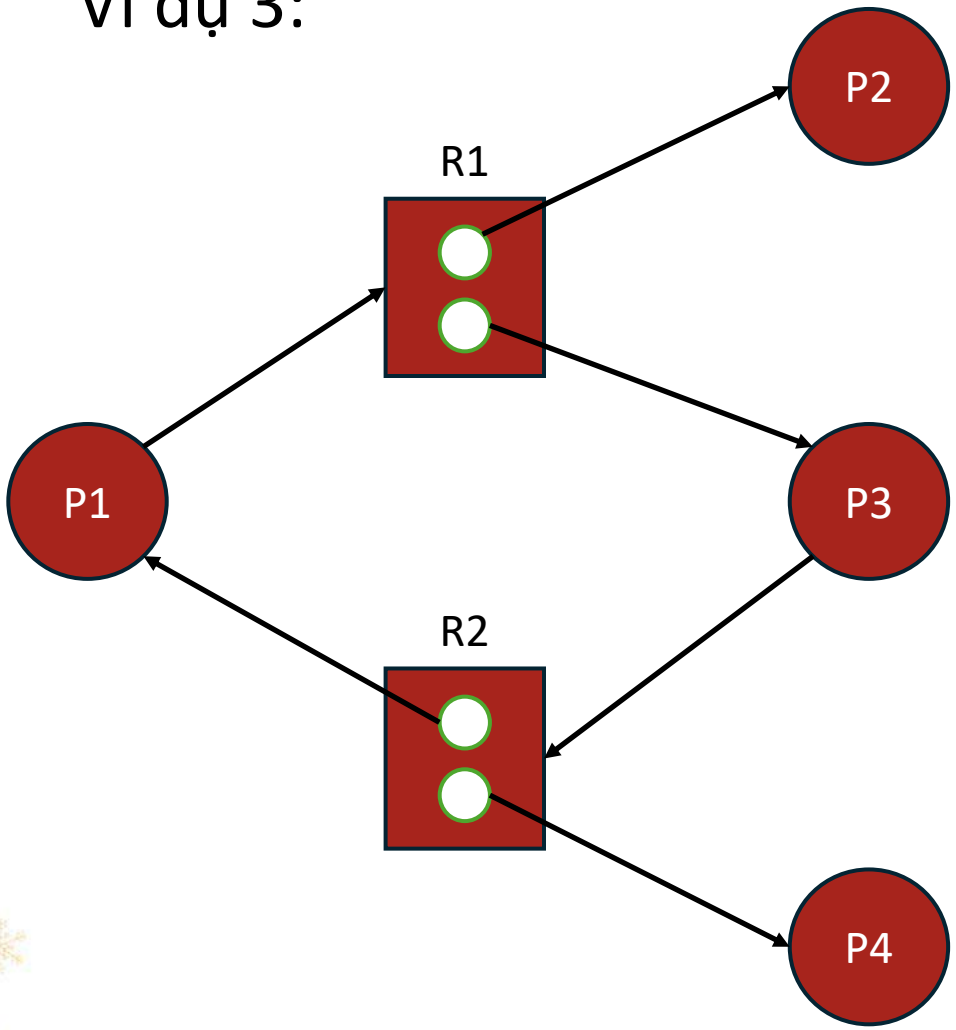
P1, R1, P2, R2, P3, R3, P1

Chuỗi thứ tự thực thi:

Không có

⇒ xảy ra deadlock

Ví dụ 3:



Chu trình:

P1, R1, P3, R2, P1

Chuỗi thứ tự thực thi:

P2 → P1 → P3 → P4

⇒ không xảy ra deadlock

NHẬN XÉT

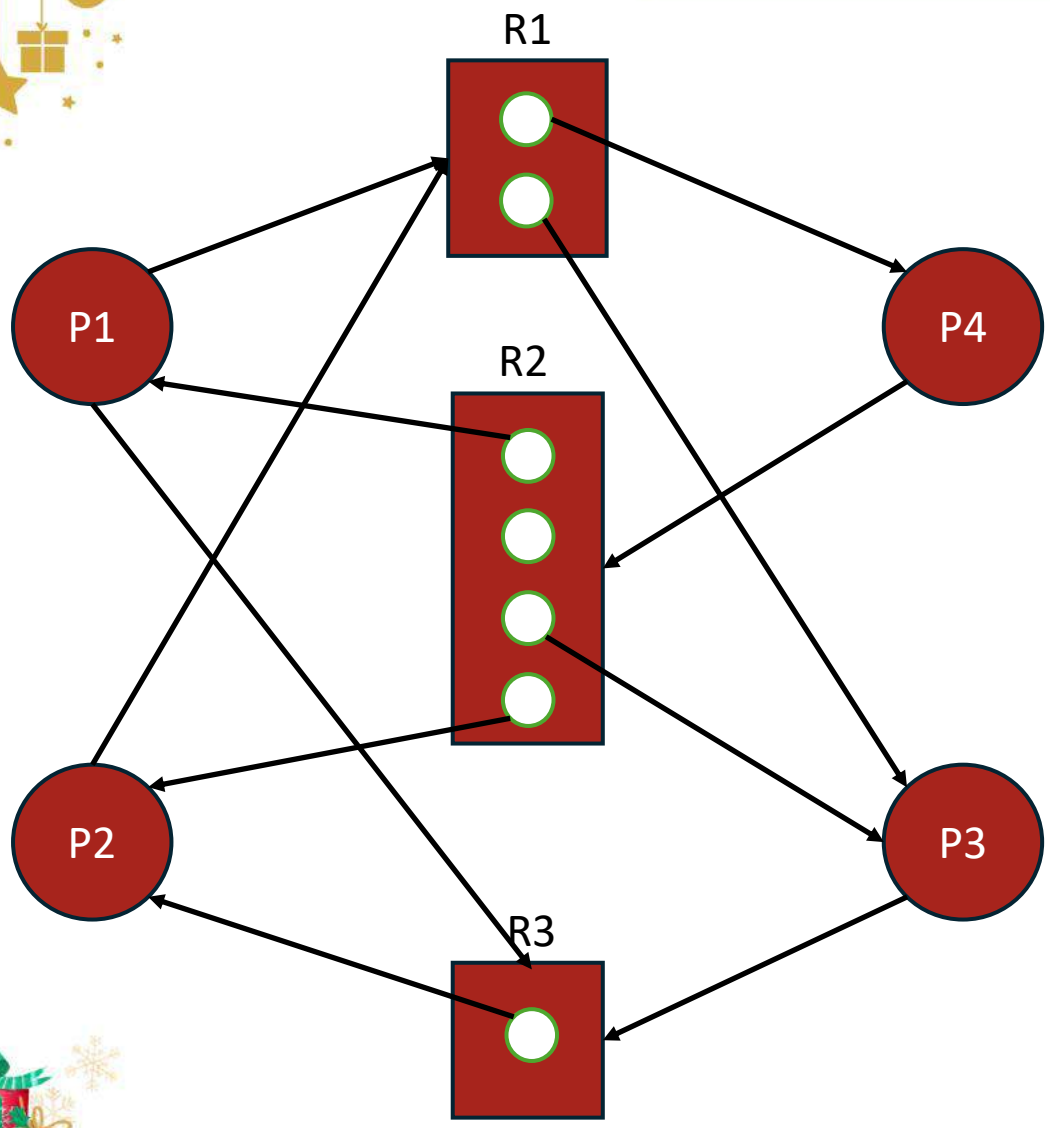
- RAG không có chu trình → chắc chắn không có deadlock
- RAG có chu trình:
  - Mỗi tài nguyên đều có 1 thực thể → xảy ra deadlock
  - Tài nguyên có nhiều thực thể → có thể xảy ra deadlock
    + Có chuỗi thứ tự thực thi → không xảy ra deadlock
    + Không có chuỗi thứ tự thực thi → xảy ra deadlock

CÂU HỎI

Cho 1 hệ thống có 4 tiến trình P1, P2, P3, P4 và 3 loại tài nguyên: R1 có 2 thực thể, R2 có 4 thực thể, R3 có 1 thực thể:

- P1 giữ 1 thực thể R2, yêu cầu 1 thực thể từ R1 và 1 thực thể R3
- P2 giữ 1 thực thể R2 và 1 thực thể R3, yêu cầu 1 thực thể từ R1
- P3 giữ 1 thực thể R1 và 1 thực thể R2, yêu cầu 1 thực thể R3
- P4 giữ 1 thực thể R1, yêu cầu 1 thực thể R2

Hỏi hệ thống có xảy ra deadlock không?



Chuỗi thứ tự thực thi:

P4 → P2 → P3 → P1

⇒ không xảy ra deadlock

BAN HỌC TẬP KHOA HỆ THỐNG THÔNG TIN

III. CÁC PHƯƠNG PHÁP GIẢI QUYẾT DEADLOCK

CÁC PHƯƠNG PHÁP GIẢI QUYẾT DEADLOCK

- Ngăn deadlock
- Tránh deadlock
- Phát hiện Deadlock và phục hồi hệ thống
- Xem như deadlock không xảy ra

NGĂN DEADLOCK

- Ngăn mutual exclusion

  + đối với tài nguyên không chia sẻ (printer): không làm được

  + đổi với tài nguyên chia sẻ (read-only flie): không cần thiết

- Hold and wait:

  + Cách 1: Mỗi tiến trình yêu cầu toàn bộ tài nguyên cần thiết một lần. Nếu có đủ tài nguyên thì hệ thống sẽ cấp phát, nếu không đủ tài nguyên thì tiến trình bị block

  + Cách 2: Khi yêu cầu tài nguyên, tiến trình không được giữ tài nguyên nào. Nếu đang có thì phải trả lại trước khi yêu cầu

NGĂN DEADLOCK

- Ngăn no preemption: Nếu tiến trình A có giữ tài nguyên và đang yêu cầu tài nguyên khác nhưng tài nguyên này chưa được cấp phát thì
  + Cách 1: Hệ thống lấy lại mọi tài nguyên mà A đang giữ
  + Cách 2: Hệ thống sẽ xem tài nguyên mà A yêu cầu
- Ngăn Circular Wait:
  +Mỗi tiến trình chỉ có thể yêu cầu thực thể của một loại tài nguyên theo thứ tự tăng dần (định nghĩa bởi hàm F) của loại tài nguyên.
  + Khi một tiến trình yêu cầu một thực thể của loại tài nguyên Rj thì nó phải trả lại các tài nguyên Ri với F(Rj) > F(Ri)

NGĂN DEADLOCK

- Không cho phép (ít nhất) một trong 4 điều kiện cần cho deadlock xảy ra
- Nhận xét: sử dụng tài nguyên không hiệu quả

TRÁNH DEADLOCK

- Yêu cầu mỗi tiến trình khai báo số tài nguyên tối đa cần để thực hiện
- Giải thuật tránh deadlock
  + Kiểm tra trạng thái hệ thống
  + Yêu cầu tài nguyên
- đảm bảo hiệu suất sử dụng tài nguyên tối đa đến mức có thể

TRẠNG THÁI HỆ THỐNG

- Được xác định dựa trên: số tài nguyên tối đa (Max), số tài nguyên đang giữ (Allocation) và số tài nguyên còn lại (Need) của các tiến trình
- Có 2 trạng thái:
  - Trạng thái an toàn (safe): tồn tại ít nhất một chuỗi an toàn
  - Trạng thái không an toàn (unsafe): không tồn tại một chuỗi an toàn

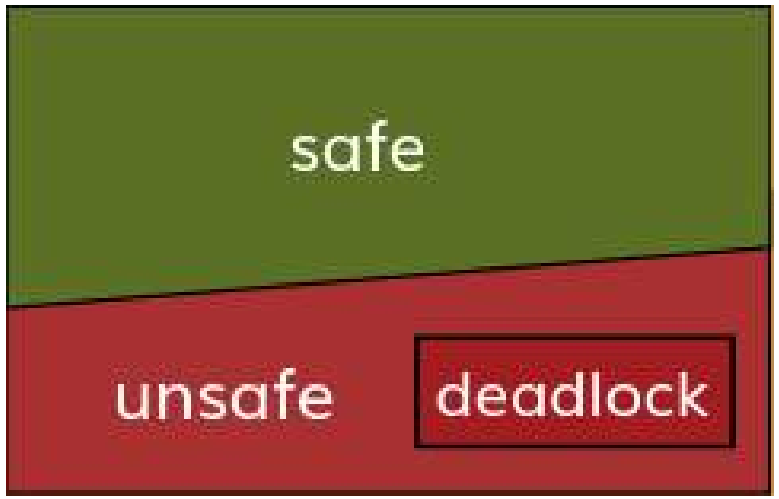
TRẠNG THÁI HỆ THỐNG

- Nếu hệ thống ở trạng thái safe

→ Không xảy ra deadlock

- Nếu ở trạng thái unsafe

→ Có thể xảy ra deadlock



GIẢI THUẬT TRÁNH DEADLOCK

Giải thuật kiểm tra hệ thống :

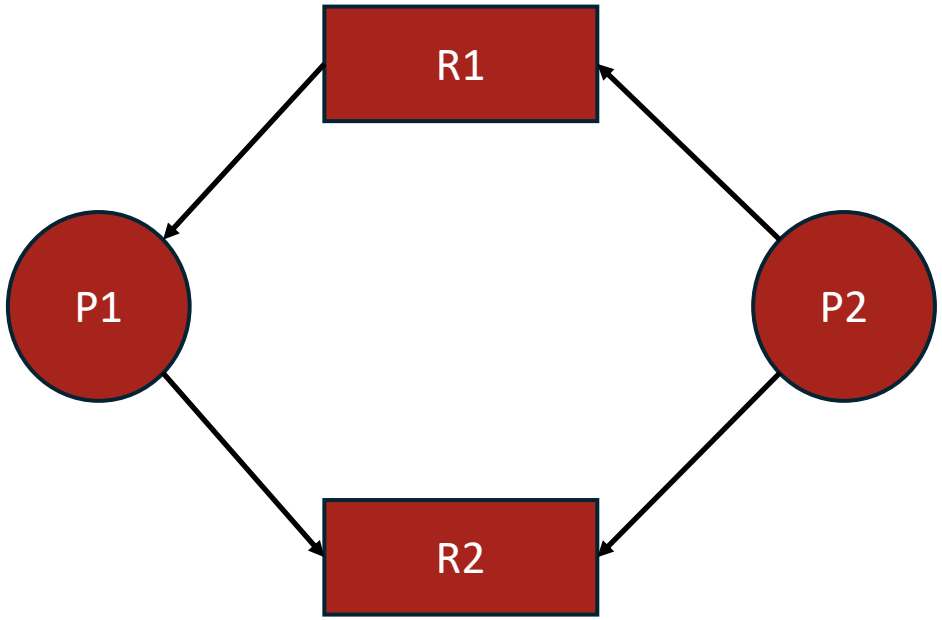
- Mỗi tài nguyên có một thực thể

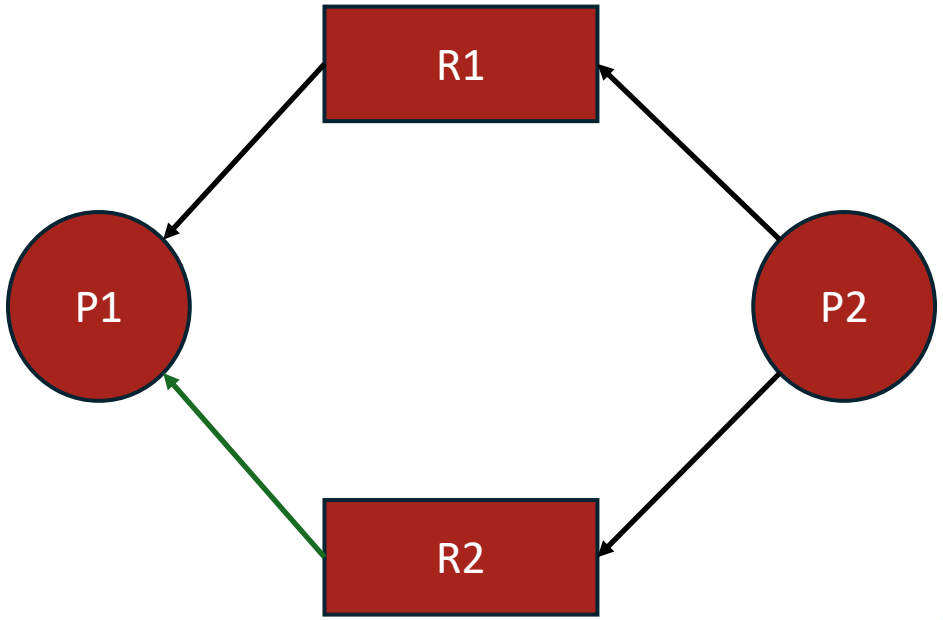
→ Giải thuật đồ thị cấp phát tài nguyên

- Tài nguyên có nhiều thực thể

→ Giải thuật Banker

GIẢI THUẬT ĐỒ THỊ CẤP PHÁT TÀI NGUYÊN





CẤU TRÚC DỮ LIỆU CHO GIẢI THUẬT BANKER

n: số tiến trình; m: số loại tài nguyên

- Available: vector độ dài m
  + Available[j] = k: loại tài nguyên Rj có k instance sẵn sàng
- Max: ma trận n x m
  + Max[i, j] = k: tiến trình Pi yêu cầu tối đa k instance của loại tài nguyên Rj
- Allocation: vector độ dài n x m
  + Allocation[i, j] = k Pi đã được cấp phát k instance của Rj
- Need: vector độ dài n x m
  + Need[i, j] = k Pi cần thêm k instance của Rj
  + ⇒ Need[i, j] = Max[i, j] - Allocation[i, j]

⇒ Ký hiệu $Y \leq X \Leftrightarrow Y[i] \leq X[i]$, ví dụ $(0, 4, 2, 1) \leq (2, 7, 3, 5)$

GIẢI THUẬT BANKER

1. Gọi Word và Finish là hai vector độ dài là m và n. Khởi tạo
   + Work = Available
   + Finish[i] = False, i = 0, 1, …, n – 1
2. Tìm I thỏa
   + Finish[i] = False
   + Need $\leq$ Work (hàng thứ i của Need)

   Nếu không tồn tại i như vậy, đến bước 4.
3. Work = Work + Allocationi
   + Finish[i] = true
   + Quay về bước 2
4. Nếu Finish[i] = true, i = 1,…, n, thì hệ thống đang ở trạng thái safe

Xét một hệ thống có 3 tiến trình: P1, P2, P3 và 2 loại tài nguyên:

R1 (5 thực thể), R2. (7 thực thể)

Tại thời điểm t0 trạng thái của hệ thống như sau:

	MAX		ALLOCATION	
Process	R1	R2	R1	R2
P1	3	5	2	2
P2	2	3	1	2
P3	1	7	1	1

AVAILABLE	
R1	R2
1	2

Hệ thống trên có an toàn hay không?

	MAX		ALLOCATION		NEED	
Process	R1	R2	R1	R2	R1	R2
P1	3	5	2	2	1	3
P2	2	3	1	2	1	1
P3	1	7	1	1	0	6

AVAILABLE	
R1	R2
1	2

Chuỗi thứ tự thực thi:

P2 → P1 → P3

Chuỗi an toàn:

⇒ Hệ thống an toàn

GIẢI THUẬT TRÁNH DEADLOCK

Giải thuật yêu cầu tài nguyên: áp dụng khi tiến trình P yêu cầu thêm tài nguyên

- Điều kiện đối với yêu cầu (Request):

  Request $\leq$ Need

  Request $\leq$ Available

- Dùng giải thuật kiểm tra hệ thống:

  Trạng thái safe: Request thực sự được đáp ứng

  Tráng thái unsafe: P đợi và phục hồi trạng thái

Xét một hệ thống có 3 tiến trình: P1, P2, P3 và 2 loại tài nguyên:

R1 (5 thực thể), R2. (7 thực thể)

Tại thời điểm t0 trạng thái của hệ thống như sau:

	MAX		ALLOCATION	
Process	R1	R2	R1	R2
P1	3	5	2	2
P2	2	3	1	2
P3	1	7	1	1

AVAILABLE	
R1	R2
1	2

Tại thời điểm t1 nếu P1 yêu cầu thêm tài nguyên (0,1) thì hệ thống có đáp ứng không?

Kiểm tra:

(-) Request P1 (0,1) ≤ Need P1 (1,3)

(-) Request P1 (0,1) ≤ Available (1,2)

Giả sử hệ thống đáp ứng yêu cầu thêm tài nguyên (0,1) của tiến trình P1

Trạng thái mới của hệ thống sau khi đáp ứng:

	MAX		ALLOCATION		NEED	
Process	R1	R2	R1	R2	R1	R2
P1	3	5	2	3	1	2
P2	2	3	1	2	1	1
P3	1	7	1	1	0	6

AVAILABLE	
R1	R2
1	1

⇒ Chuỗi an toàn: <P2, P1, P3> ⇒ Trạng thái mới là safe

Kết luận: Có thể cấp phát tài nguyên cho P1

PHÁT HIỆN DEADLOCK

- Cho phép deadlock xảy ra
- Sau đó phát hiện deadlock và phục hồi hệ thống

GIẢI THUẬT PHÁT HIỆN DEADLOCK

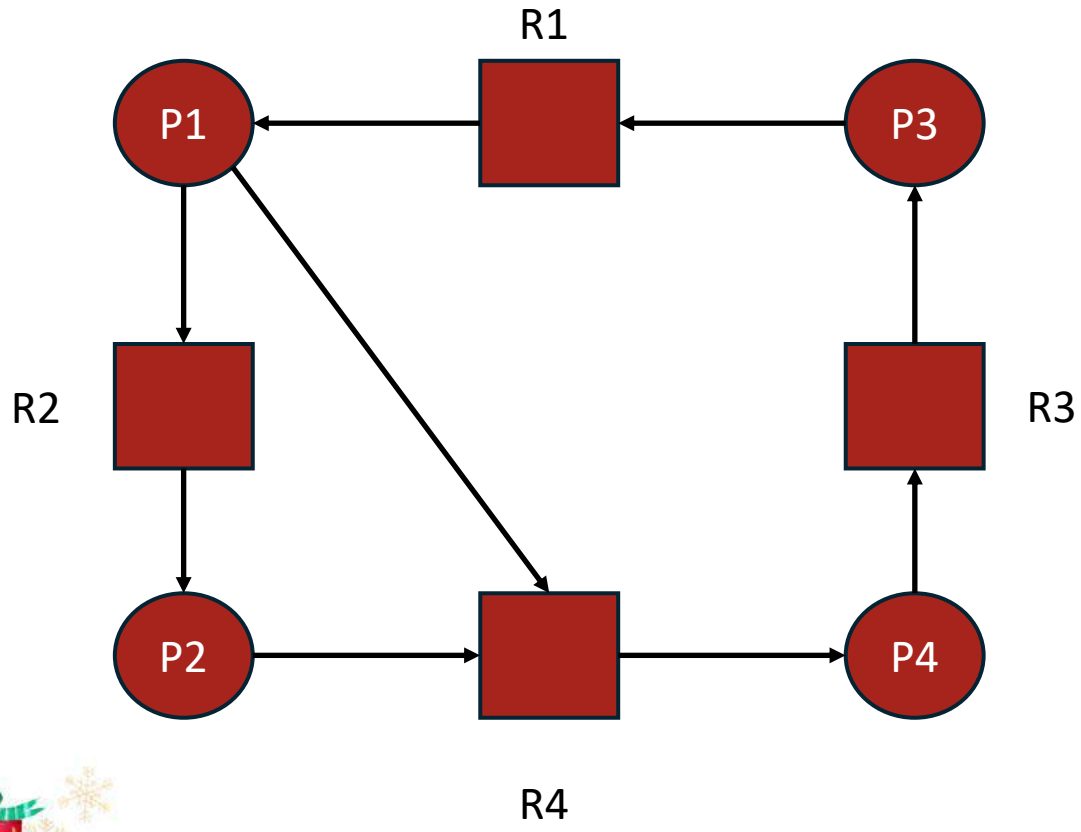
- Tài nguyên có một thực thể

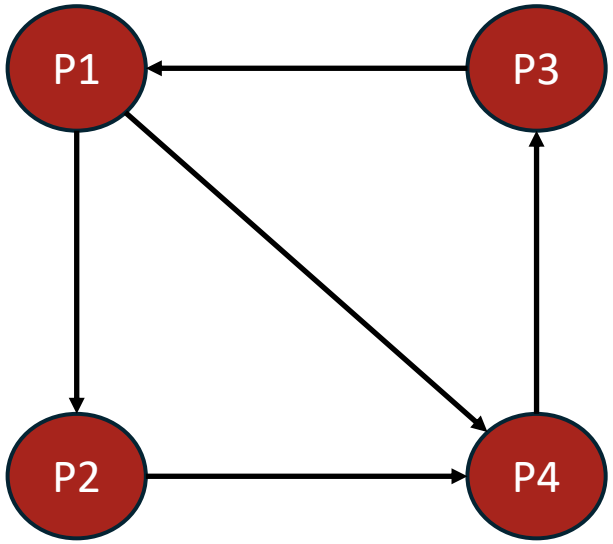
  Đồ thị wait-for

- Tài nguyên có nhiều thực thể

  Giải thuật phát hiện deadlock

SƠ ĐỒ CẤP PHÁT TÀI NGUYÊN





GIẢI THUẬT PHÁT HIỆN DEADLOCK

1. Gọi Work và Finish là vector kích thước m và n. Khởi tạo:

   Work = Available

   For i = 1, 2,…, n, nếu Request ≠ 0 thì Finish[i] := false còn không thì Finish[i] := true

2. Tìm i thỏa mãn

   Finish[i] = false

   Requesti ≤ Work

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

3. Work = Work + Allocationi

   Finish[i] = true

   quay về bước 2.

4. Nếu Finish[i] = false, với một số i = 1,…, n, thì hệ thống đang ở trạng thái deadlock. Hơn thế nữa, Finish[i] = false thì Pi bị deadlocked

Xét một hệ thống có 3 tiến trình: P1, P2, P3 và 2 loại tài nguyên: R1 (6thực thể), R2. (7 thực thể)
Tại thời điểm t0 trạng thái của hệ thống như sau:

	ALLOCATION		REQUEST		AVAILABLE	
Process	R1	R2	R1	R2	R1	R2
P1	0	1	0	0	1	0
P2	2	3	1	2		
P3	3	1	0	1		
P4	0	2	0	0		

Hệ thống trên có an toàn hay không?

PHỤC HỒI DEADLOCK

- Báo người vận hành
- Hế thống từ động phục hồi bằng cách bẻ gãy chu trình deadlock

Chấm dứt một hày nhiều tiến trình

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

+ Dựa trên nhiều yếu tố khác nhau

Lấy lại tài nguyên từ một hay nhiều tiến trình

+ Lấy lại tài nguyên từ một tiến trình, cấp phát cho tiến trình khác cho đến khi không còn deadlock nữa.

+ Chọn “nạn nhân”

+ Trở lại trạng thái trước deadlock (Rollback)

+ Đói tài nguyên (Starvation)

XEM DEADLOCK KHÔNG TỒN TẠI

- Được sử dụng nhiều nhất (trong Window, Linux)
- Nếu có deadlock cần khởi động lại máy

CHƯƠNG 7: QUẢN LÝ BỘ NHỚ

1. Khái niệm
2. Địa chỉ bộ nhớ
3. Các mô hình quản lí bộ nhớ

1 Khái niệm cơ sở

- 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**
- Quản lý bộ nhớ là công việc của hệ điều hành với sự hỗ trợ của phần cứng nhằm phân phối, sắp xếp các process trong bộ nhớ sao cho hiệu quả

Nhiệm vụ của hệ điều hành trong quản lý bộ nhớ

- 5 nhiệm vụ chính: 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 một phần cố định của bộ nhớ (low memory), phần còn lại phân phối các process (high memory)

2. Địa chỉ bộ nhớ

Địa chỉ vật lý (physical address – địa chỉ thực): là một vị trí thực trong bộ nhớ chính

- *Địa chỉ tuyệt đố*i(absolute address): địa chỉ tương đương với địa chỉ thực
- Địa chỉ vật lý = địa chỉ frame + offset

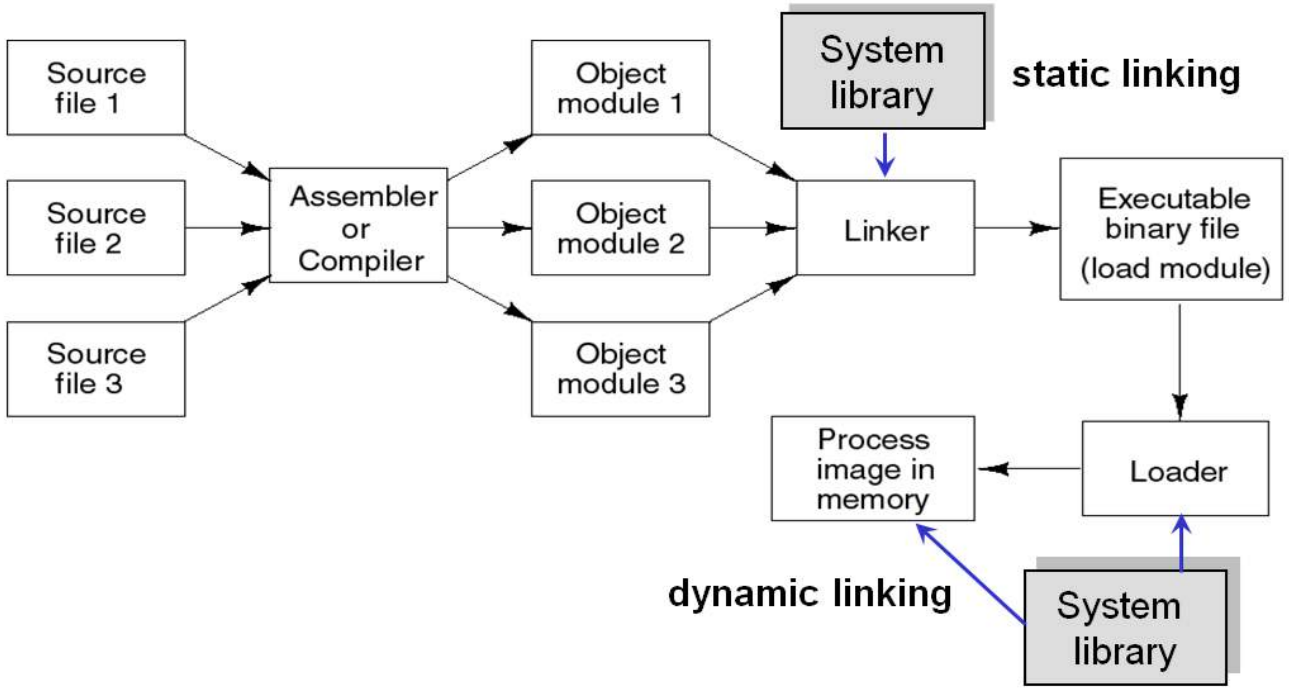
Địa chỉ luận lý (logical address – virtual address – địa chỉ ảo): vị trí nhớ được diễn tả trong một chương trình

- *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

Nạp chương trình vào bộ nhớ

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



3. Các mô hình quản lý bộ nhớ

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

1. Phân chia cố định (fixed partitioning)

2. Phân chia động (dynamic partitioning)

3. Phân trang đơn giản (simple paging)

Hiện tượng phân mảnh bộ nhớ (fragmentation)

1. *Hiện tượng phân mảnh nội:*

Kích thước vùng nhớ được cấp phát lớn hơn vùng nhớ yêu cầu.

=> Có thể sử dụng các chiến lược placement

2. *Hiện tượng phân mảnh ngoại:*

Kích thước không gian bộ nhớ trống đủ thỏa mãn yêu cầu cấp phát, tuy nhiên không liên tục.

=> Có thể dùng cơ chế kết khối (compaction).

VD: Nếu hệ thống cấp phát vùng nhớ có kích thước 20480 byte cho tiến trình yêu cầu 20324 byte thì sẽ dẫn đến tình trạng gì?

A. Deadlock

B. Phân mảnh ngoại

C. Phân mảnh nội

D. Số lỗi trang tăng lên

Phân chia cố định (Fixed partitioning)

- Bộ nhớ chính được chia thành nhiều phần (partition), có kích thước bằng nhau hoặc khác nhau.
- Process nào nhỏ hơn hoặc bằng kích thước partition thì có thể nạp vào.
- Nếu process có kích thước lớn hơn => Overlay.

=> ***Kém hiệu quả do bị phân mảnh nội.***

PHÂN CHIA CỐ ĐỊNH

1. 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.

- Giải pháp 1: Sử dụng nhiều hàng đợi
  - Gán mỗi process vào partition nhỏ nhất phù hợp.
  - **Ưu điểm:** Giảm thiểu phân mảnh nội
  - **Nhược điểm:** Có thể có hàng đợi trống và hàng đợi dày đặc.
- Giải pháp 2: Sử dụng 1 hàng đợi
  - Chỉ có một hàng đợi chung cho mọi partition.

=> Chọn partition nhỏ nhất còn trống.

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

Phân chia động (dynamic partitioning)

Chiến lược placement (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.

Giả sử bộ nhớ chính được phân chia thành các vùng nhớ cố định thứ tự như sau: 1 (200KB), 2 (180KB), 3 (400KB), 4 (220KB), 5 (360KB). Biết con trỏ đang ở vùng nhớ thứ 2, vùng nhớ thứ 2 đã được cấp phát, các vùng nhớ khác vẫn còn trống. Hỏi tiến trình P có kích thước 210KB sẽ được cấp phát cho vùng nhớ nào, nếu dùng giải thuật first-fit?

A. 3 B. 1 C.4 D.2

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 lỹ 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:

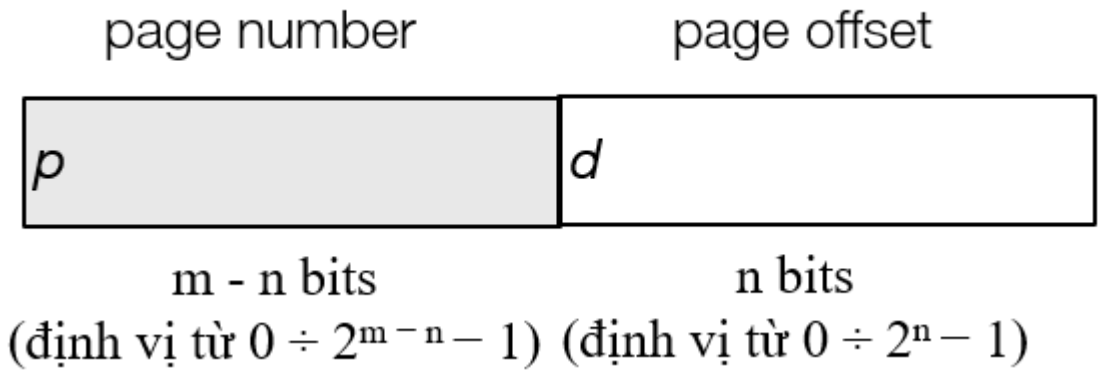
  o Số hiệu trang(page number) (p)

  o Địa chỉ tương đối trong trang (page offset) (d)

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

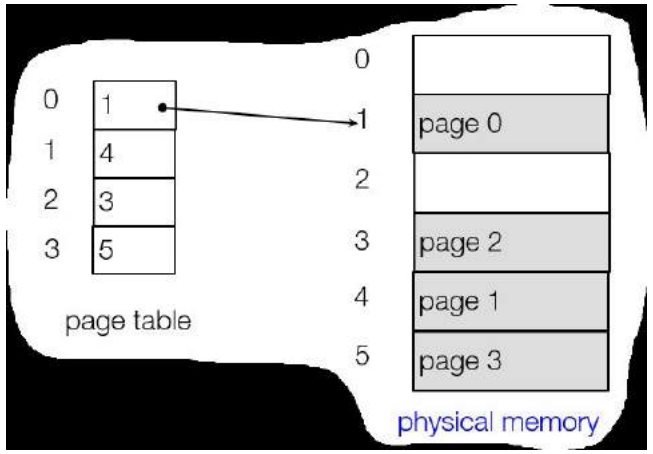
Chuyển đổi địa chỉ trong paging

• Nếu kích thước của không gian địa chỉ ảo là 2 m, và kích thước của trang là 2 n (đơn vị là byte hay word tùy theo kiến trúc máy) thì :

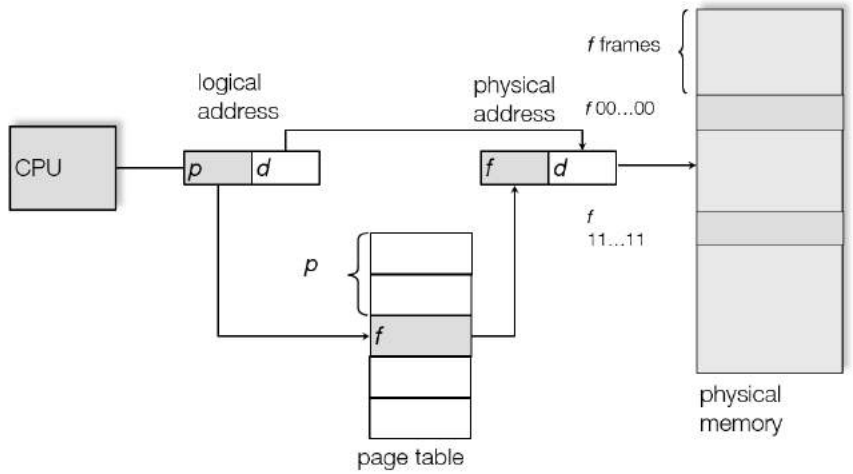


Chuyển đổi địa chỉ trong paging

- Bảng trang sẽ có tổng cộng 2m/2n = 2m - n mục (entry)
- Mỗi page sẽ ứng với một frame và được tìm kiếm thông qua page table



Page table



Chuyển đổi địa chỉ thông qua page table

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 luận lý gồm bao nhiêu bit?

b) Địa chỉ vật lý gồm bao nhiêu bit?

c) Bảng trang có bao nhiêu mục? Mỗi mục trong bảng trang cần bao nhiêu bit?

a) ĐCLL = ĐC page + ĐC offset = 8 * 1024 = 2^3 * 2^10 = 2 ^13 => 13 bit

b) ĐCVL = ĐC frame + ĐC offset = 32 * 1024 = 2 ^5 * 2 ^10 = 2 ^15 => 15 bit

c) Số mục trong bảng phân trang = Số trang = 8

Số bit của mỗi mục trong bảng trang = Số bit của 1 khung trang = 5 bit

Xét một không gian có bộ nhớ luận lý có 64 trang, mỗi trang có 1024 từ, mỗi từ là 2 byte được ánh xạ vào bộ nhớ vật lý có 32 khung trang.

a) Địa chỉ bộ nhớ vật lý có bao nhiêu bit?

b) Địa chỉ bộ nhớ luận lý có bao nhiêu bit?

c) Có bao nhiêu mục trong bảng phân trang? Mỗi mục chứa bao nhiêu bit?

a) ĐCVL = ĐC frame + ĐC offset = 32 * 2048 = 2 ^5 * 2 ^11 = 2 ^16 => 16 bit

b) ĐCLL = ĐC page + ĐC offset = 64 * 2048 = 2 ^6 * 2 ^11 = 2 ^17 => 17 bit

c) Số mục trong bảng phân trang = Số trang = 64 Số bit của mỗi mục trong bảng trang = Số bit của 1

khung trang = 5 bit

a) Địa chỉ vật lý 6568 sẽ được chuyển thành địa chỉ ảo bao nhiêu? Biết rằng kích thước mỗi frame là 1K bytes.

a) Kích thước frame = 1024 bytes

f = 6568/1024 = 6 (Lấy phần nguyên)

Từ bảng trang, ta có: p = 0

d = 6568%1024= 424 (lấy phần dư)

ĐC ảo = 0*1024 + 424 = 424

1 1024	
1025 2048	

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

Bảng trang của P1

b) Địa chỉ ảo 3254 sẽ được chuyển thành địa chỉ vật lý bao nhiêu? Biết rằng kích thước mỗi frame là 2K bytes

b) Kích thước frame = 2048 bytes

p = 3254/2048 = 1 (Lấy phần nguyên)

Từ bảng trang, ta có: f = 4

d = 3254%2048= 1206 (lấy phần dư)

ĐC thực = 4*2048 + 1206 = 9398

6145

8192

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

Bảng trang của P1

Translation look-aside buffers (TLBs)

- **Thời gian truy xuất hiệu dụng (effective access time,EAT)**
  - o Thời gian tìm kiếm trong TLB: ε
  - o Thời gian truy xuất trong bộ nhớ: x
  - o 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**
  - o Khi chỉ số trang có trong TLB (hit): ε + x
  - o Khi chỉ số trang không có trong TLB (miss): ε+x+x
- **Thời gian truy xuất hiệu dụng**
  - o EAT = $(2-\propto)x + \varepsilon$

VD: Xét một hệ thống dùng kỹ thuật phân trang với bảng trang được lưu vào bộ nhớ chính. Nếu sử dụng TLBs với hit ratio α = 0.85 thì thời gian truy xuất bộ nhớ trong hệ thống (effective access time) EAT = 230ns. Biết thời gian một chu kỳ truy xuất bộ nhớ là x = 180ns. Hỏi thời gian tìm kiếm trong TLBs là bao nhiêu?

A. 23ns B. 27ns C. 153ns D. 207ns

$$AT = (2 - \alpha)\, x + \varepsilon$$

$$230 = (2 - 0.85)\, 180 + \varepsilon$$

$$\varepsilon = 23ns$$

Xét một hệ thống sử dụng kỹ thuật phân trang, với bảng trang được lưu trữ trong bộ nhớ chính.

a. 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?

b. Nếu sử dụng TLBs với hit-ratio là 75%, thời gian để tìm trong TLBs xem như bằng 0, tính thời gian truy xuất bộ nhớ trong hệ thống.

a) Một lần truy xuất bộ nhớ bình thường là 200ns. Một thao tác thực hiện 2 lần truy xuất. Vậy thời gian là 2*200 = 400ns.

b) EAT = $(2 - \alpha)\, x + \varepsilon = (2 - 0.75)\, 200 + 0 = 250ns$

BẢO VỆ BỘ NHỚ

Việc bảo vệ bộ nhớ được hiện thực bằng cách gắn với frame các bit bảo vệ (proctection bits) được giữ trong bảng phân trang. Các biến này được biểu diễn: **read -only, read-write, execute-only**.

CƠ CHẾ HOÁN VỊ (SWAPPING)

Một process có thể tạm thời bị swap ra khỏi bộ nhớ chính và lưu trên một hệ thống lưu trữ phụ. Sau đó, process có thể được nạp lại vào bộ nhớ để tiếp tục quá trình thực thi.

CHƯƠNG 8: BỘ NHỚ ẢO

1. Khái niệm tổng quan về bộ nhớ ảo
2. Vận dụng các kỹ thuật cài đặt được bộ nhớ ảo
3. Vấn đề trong bộ nhớ ảo

TỔNG QUANG BỘ NHỚ ẢO

- 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 bộnhớ vật lý
- Ưu điểm của bộ nhớ ảo:
  - Số lượng process trong bộ nhớ nhiều hơn.
  - Một process cóthể thực thi ngay cả khi kích thước của nó lớn hơn bộ nhớ thực.
  - Giảm nhẹ công việc của lập trình viên.

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 hổ trợ paging và/hoặc segmentation

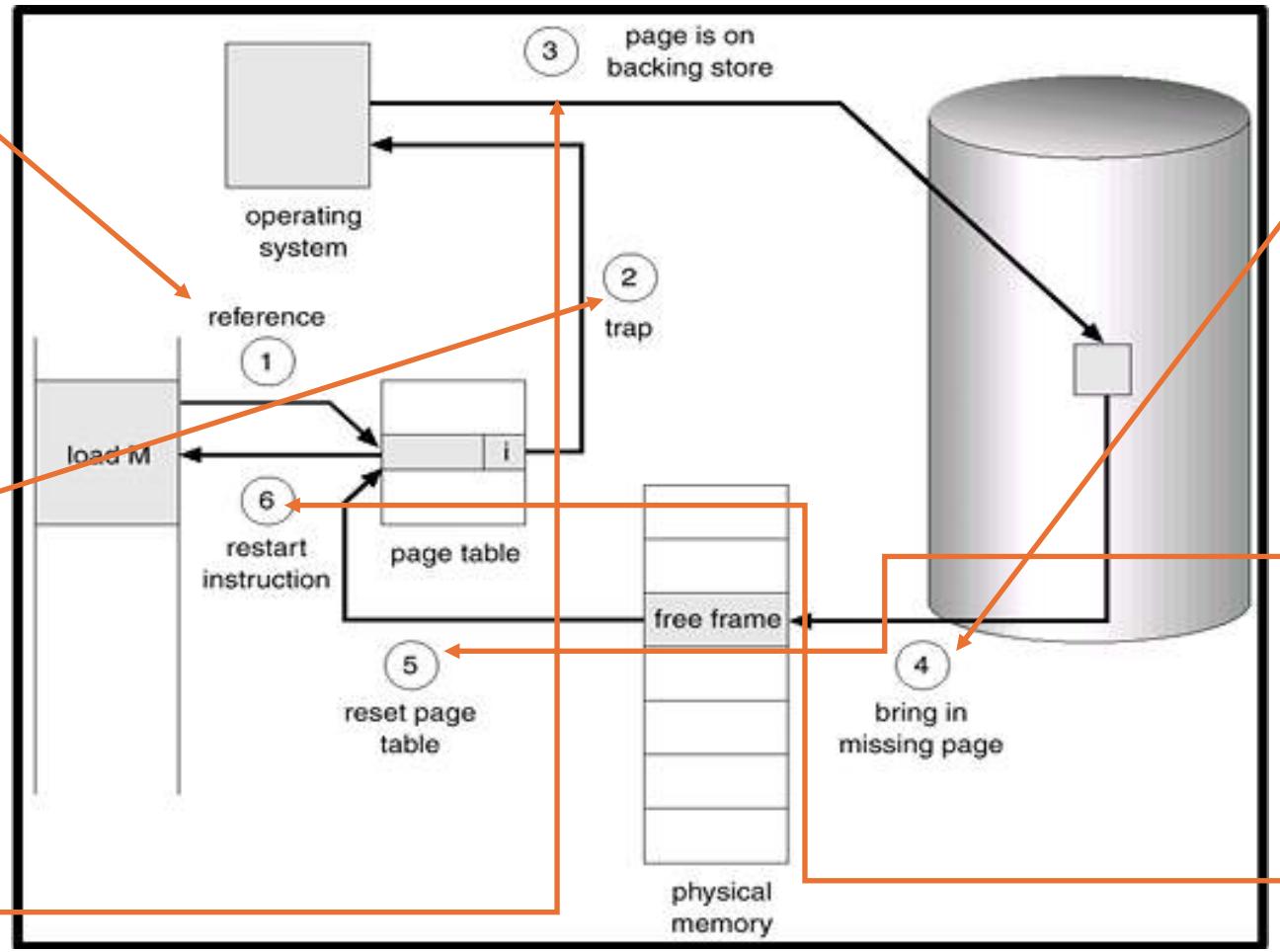
PHÂN TRANG THEO YÊU CẦU (DEMAND PAGING)

- Các trang của tiến trình chỉ được nạp vào bộ nhớ chính khi được yêu cầu
- Page-fault: khi tiến trình tham chiếu đến một trang không có trong bộ nhớ chính (valid bit) thì phần cứng sẽ gây ra một lỗi trang (page–fault)

1. Chương trình tham chiếu đến một trang không có trong bộ nhớ chính

2. Phần cứng sẽ gây ra một ngắt (page-fault trap) khởi động dịch vụ page-fault service routine (PFSR) của hệ điều hành

3. Trang mà chương trình cần đang nằm trên bộ nhớ thứ cấp (bộ nhớ tạm, đĩa, …)



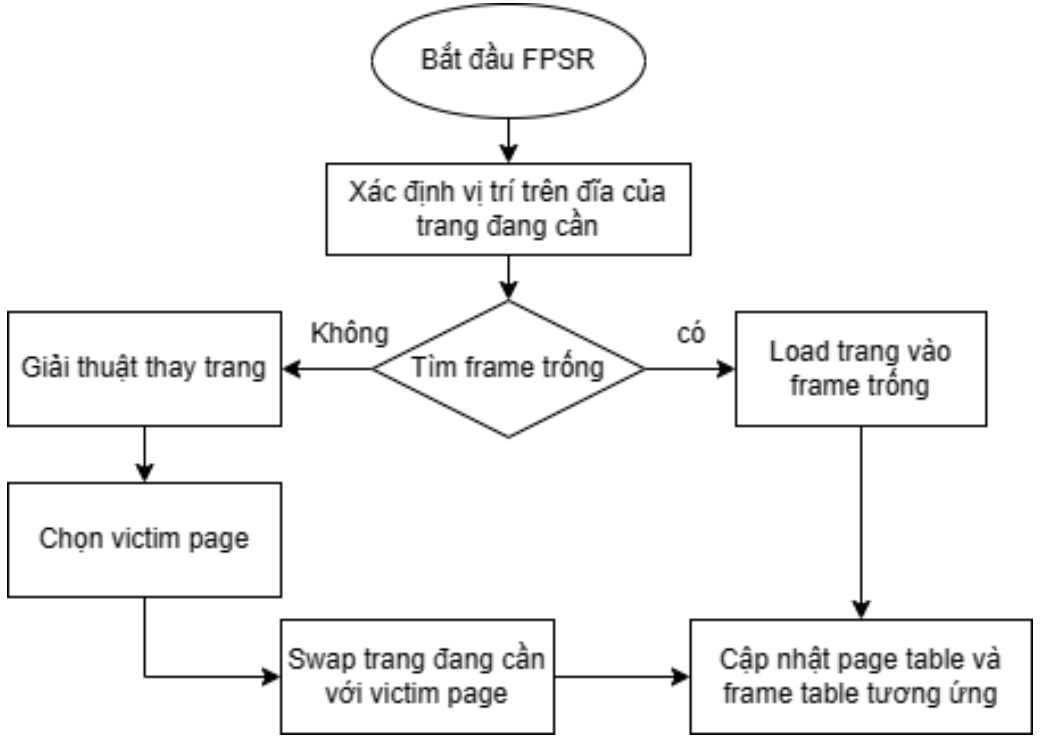
4. Tìm một frame trống và load trang vào frame trống ấy. Hoặc sử dụng các giải thuật thay thể trang (FIFO, LRU, OPT)

5. Cập nhật lại page table và frame table tương ứng

6. Khởi động lại process khi đã có nội dung page mà nó yêu cầu

PAGE-FAULT SERVICE ROUTINE

- 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.



GIẢI THUẬT THAY TRANG

Dữ liệu cần biết:

- Số khung trang
- Tình trạng ban đầu
- Chuỗi tham chiếu

Nghịch lý Belady: số page-fault tăng mặc dù tiến trình đã được cấp nhiều frame hơn.

	LRU	FIFO	OPT
Chọn trang	Trang ít được tham chiếu nhất	Trang ở trong bộ nhớ lâu nhất	Trang sẽ được tham chiếu trễ nhất trong tương lai
Nhược điểm	Đòi hỏi sự trợ giúp từ phần cứng để sắp xếp thứ tự các trang theo thời điểm tham chiếu	Cài đặt đơn giản nhưng dễ mắc phải nghịch lý Belady	Khó hiện thực vì không thể đoán chính xác thời điểm một trang được tham chiếu

Giả sử có 3 khung trang và các khung trang ban đầu là rỗng. Xác định số Page Fault sử dụng LRU?

0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

LRU: Trang ít được tham chiếu nhất

0	2	1	6	4	0	1	0	3	1	2	1
0	0	0	6	6	6	1	1	1	1	1	1
	2	2	2	4	4	4	4	3	3	3	3
		1	1	1	0	0	0	0	0	2	2
x	x	x	x	x	x	x		x		x	

Giả sử có 3 khung trang và các khung trang ban đầu là rỗng. Xác định số Page Fault sử dụng FIFO?

0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

FIFO: Trang ở trong bộ nhớ lâu nhất

0	2	1	6	4	0	1	0	3	1	2	1
0	0	0	6	6	6	1	1	1	1	1	1
	2	2	2	4	4	4	4	3	3	3	3
		1	1	1	0	0	0	0	0	2	2
x	x	x	x	x	x	x		x		x	

Giả sử có 3 khung trang và các khung trang ban đầu là rỗng. Xác định số Page Fault sử dụng OPT?

0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

OPT: Trang ít được tham chiếu nhất

0	2	1	6	4	0	1	0	3	1	2	1
0	0	0	0	0	0	0	0	3	3	3	3
	2	2	6	4	4	4	4	4	4	2	2
		1	1	1	1	1	1	1	1	1	1
x	x	x	x	x	x	x		x		x	

BÀI TẬP ÔN TẬP

Xét chuỗi truy xuất bộ nhớ sau:

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

Có bao nhiêu lỗi trang xảy ra khi sử dụng các thuật toán thay thế sau đây:

a. LRU với 4 khung trang rỗng
b. FIFO với 2 khung trang rỗng
c. OPT với 5 khung trang rỗng

Xét chuỗi truy xuất bộ nhớ sau:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

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

LRU với 4 khung trang, 10 lỗi trang

Xét chuỗi truy xuất bộ nhớ sau:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

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

FIFO với 2 khung trang, 18 lỗi trang

Xét chuỗi truy xuất bộ nhớ sau:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

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

OPT với 5 khung trang, 7 lỗi trang

BAN HỌC TẬP KHOA HỆ THỐNG THÔNG TIN