Hệ điều hành

Chương 5: Đồng bộ

5.1 Tổng quan về đồng bộ

5.2 Nhóm giải pháp Busy Waiting

5.3 Nhóm giải pháp Sleep & Wake up

Chương 6: Deadlocks

6.1 Tổng quan về deadlock

6.2 Các tính chất của deadlock

6.3 Phương hap giải quyết deadlock

Chương 7: Quản lý bộ nhớ

7.1 Khái niệm

7.2 Địa chỉ bộ nhớ

7.3 Các mô hình quản lý bộ nhớ

Chương 8: Bộ nhớ ảo

8.1 Tổng quan

8.2 Cài đặt bộ nhớ ảo

8.3 Phân trang theo yêu cầu

8.4 Giải thuật thay trang

8.5 Vấn đề cấp phát Frames

8.6 Vấn đề Thrasing

Chương 5: Đồng bộ

Mục tiêu:

- Hiểu vấn đề vùng tương trục
- Hiểu cơ chế hoạt động hiệu báo Semaphores để đồng bộ quá trình
- Hiểu cơ chế hoạt động của Monitors đề đồng bộ hóa quá trình
- Vận dụng các giải pháp để giải quyết các bài toán đồng bộ hóa cơ bản

5.1 Tổng quan về đồng bộ

5.1.1 Giới thiệu bài toán Producer-Consumer

Producer	Consumer

`while (true) {`
`  /* produce an item in next_produced */`
 	
`  while (count == BUFFER_SIZE)`
`  ; /* do nothing */`
 	
`  buffer[in] = next_produced;`
`  in = (in + 1) % BUFFER_SIZE;`
`  count++;`
`}`
	`while (true) {`
	`  while (count == 0)`
	`  ; /* do nothing */`
	
	`  next_consumed = buffer[out];`
	`  out = (out + 1) % BUFFER_SIZE;`
	`  count--;`
	
	`  /* consume the item in next_consumed */`
	`}`

* *count* dùng chung

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.

Khái niệm: Điều kiện tranh chấp (Race condition):

- Nhiều process truy xuất và thao tác đồng thời lên dữ liệu chia sẻ
- Kết quả cuối cùng của việc truy xuất phụ thuộc vào thứ tự thực thi của các lệnh thao tác lên dữ liệu

Biện pháp ngăn chặn điều kiện tranh chấp: Bảo đảm sao cho tại mỗi thời điểm chỉ có một process được thao tác lên dữ liệu chia sẻ. Cần có cơ chế để đồng bộ hoạt động của các process này

5.1.2 Vấn đề vùng tương trục (Critical Section ~ CS)

Bài toán đặt ra: Một hệ thống gồm n quá trình ($P_0, P_1, P_2, .., P_{n-1}$). Mỗi process P_i có đoạn code:

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

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

Khái niệm: 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)

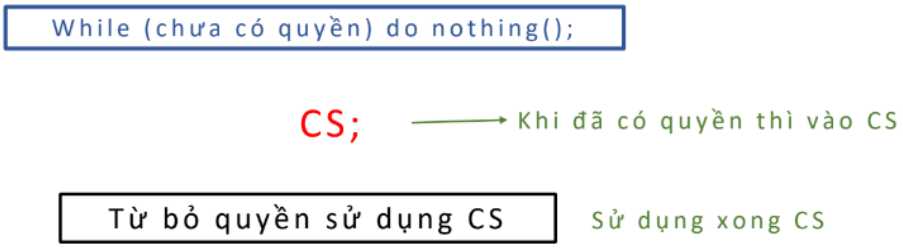
Điều kiện giải quyết bài toán CS,thỏa mãn 3 tính chất:

- **Loại trừ tương hỗ (Mutual exclution):** Nếu process P_i 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)

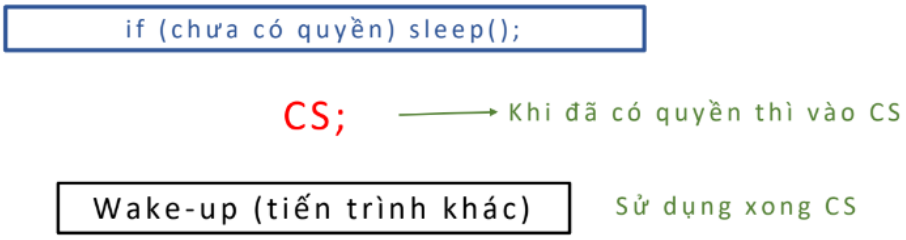
Có 2 nhóm giải pháp:

- 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



- Sleep & Wake up
  - Từ bỏ CPU khi chưa được vào vùng tranh chấp
  - Cần hệ điều hành hỗ trợ



5.2 Nhóm giải pháp Busy Waiting

5.2.1 Giải thuật Peterson: Là kết hợp hai ý tưởng quan trọng trong giải thuật "Luân phiên" và "Cờ hiệu" (áp dụng cho 2 quá trình cùng lúc)

Giải thuật Luân phiên	Giải thuật Cờ hiệu
-Áp dụng cho chỉ hai quá trình cùng một lúc -Biến chia sẻ: *turn*	

do {
while (turn != i);
critical section
turn = j;
remainder section
} **while** (1);
	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 tính chất **Mutual exclution** -Không thỏa **Progress & Bounded Waiting** vì tính chất "strict alternation" của giải thuật	-Thỏa tính chất **Mutual exclution** -Không thỏa tính chất **Progress**

Đặc điểm **Peterson**: dùng cả biến turn và flag

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Thỏa 3 tính chất:

- Biến turn chỉ có thể là 0 hoặc 1 nên tại mỗi thời điểm chỉ có một process ở trong CS => thỏa **Mutual Exclusion**
- Quá trình P_i có thể bị ngăn chặn vào miền tương trục khi *flag[j] = true* & *turn = j* . Khi P_i ra khỏi miền tương trục thì sẽ gán *flag[i] = false* => thỏa **Progress** & **Bounded Waiting**

5.2.2 Giải thuật bakery (n process)

Minh họa

```
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);
```

Đặc điểm:

- Mỗi process nhận một con số.Process nào giữ số nhỏ nhất thì được vào CS
- Nếu cùng số thì so sánh số thứ tự

VD: nếu P_i và P_j cùng số thì (i < j) => P_i vào trước

- Khi ra khỏi CS, P_i đặt lại bằng 0
- Cơ chế cấp số cho các process: tăng dần

5.2.3 Từ software đến hardware

Khuyết điểm của các giải pháp software:

- Các process khi yêu cầu vào vùng tranh chấp phải liên tục kiểm tra điều kiện (busy waiting) => tốn nhiều thời gian xử lý của CPU
- Nếu thời gian xử lý trong vùng CS lớn => cần có cơ chế block process

Các giải pháp hardware:

- Dùng các lệnh đặc biệt
- Cấm ngắt(disable interrupts)

<table>
<tr><th colspan="2">Cấm ngắt</th></tr>
<tr><td>Trong hệ thống uniprocessor</td><td>Trong hệ thống multiprocessor</td></tr>
<tr><td colspan="2">Process $\mathbf{P_i}$:
do {
disable_interrupts();
critical section
enable_interrupts();
remainder section
} while (1);
disable_interrupts(): Tiến trình hiện tại ngăn các tiến trình khác vào vùng tranh chấp
enable_interrupts(): Cho phép những tiến trình khác tiến vào vùng tranh chấp</td></tr>
<tr><td>-Mutual exclution được đảm bảo
-Gây ảnh hưởng đến các thành phần khác của hệ thống có sử dụng ngắt như system clock
-Cần phải liên tục tạm dừng và phục hồi ngắt dẫn đến hệ thống tốn chi phí quản lý và kiểm soát</td><td>-Mutual exclution 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</td></tr>
</table>

Một số giải thuật khác:

-Chỉ thị TSL (Test and Set Lock)

Minh họa:

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

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

Đặc điểm:

- Đảm bảo được tính chất Mutual Exclution
- Qúa trình chọn lựa process P_j vào CS kế tiếp là tùy ý
- Không đảm bảo được điều kiện Bounded Waiting=> Xảy ra tình trạng starvation

-Swap

Minh họa:

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

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

Đặc điểm:

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

-TSL thỏa mãn 3 tính chất (Mutual exclusion – progress – bounded waiting)

Minh họa:

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

```
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);
```

Đặc điểm:

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

5.3 Nhóm giải pháp Sleep & Wake up

Ý tưởng:

- Hệ điều hành cung cấp hai lệnh SLEEP và WAKEUP
- Nếu tiến trình gọi lệnh SLEEP, hệ điều hành chuyển tiến trình sang *ready list*, lấy lại CPU cấp cho tiến trình khác
- Nếu tiến trình gọi lệnh WAKEUP, hệ điều hành chọn một tiến trình trong *ready list*, cho tiến trình đó thực hiện tiếp
- Khi một tiến trình chưa đủ điều kiện vào CS, nó gọi SLEEP để tự khóa, cho đến khi một tiến trình khác gọi WAKEUP để giải phóng cho nó
- Khi ra khỏi CS, tiến trình gọi WAKEUP để đánh thức một tiến trình đang chờ, tạo cơ hội cho tiến trình này vào CS

5.3.1 Semaphore

Minh họa:

Định nghĩa semaphore:

```
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);
        }
}
```

Đặc điểm:

- Là công cụ đồng bộ cung cấp bởi OS, không đòi hỏi busy waiting
- Semaphore S là một biến số nguyên
- Semaphore chỉ có thể được truy xuất qua hai tác vụ có tính đơn nguyên (atomic) và loại trừ (mutual exclution)
    - Wait(S): giảm giá trị semaphore. Kế đó nếu giá trị này âm thì process thực hiện wait sẽ bị sleep()
        - Dùng để **giành** tài nguyên

        - Hàng đợi là DSLK các PCB(Process Control Block)
    - Signal(S): tăng giá trị semaphore. Kế đó nếu giá trị này không dương, một process đang sleep bởi lệnh wait() sẽ được wakeup() để thực thi
        - Dùng để **giải phóng** tài nguyên
        - Sử dụng cơ chế FIFO
- Tránh **busy waiting:** Khi phải chờ đợi thì process được đặt vào một block queue, trong đó chứa các process đang chờ đợi cùng một sự kiện
- Có 2 loại:
    - Counting semaphore: S là số nguyên
    - Binary semaphore: S =0 | S = 1

Ví dụ:

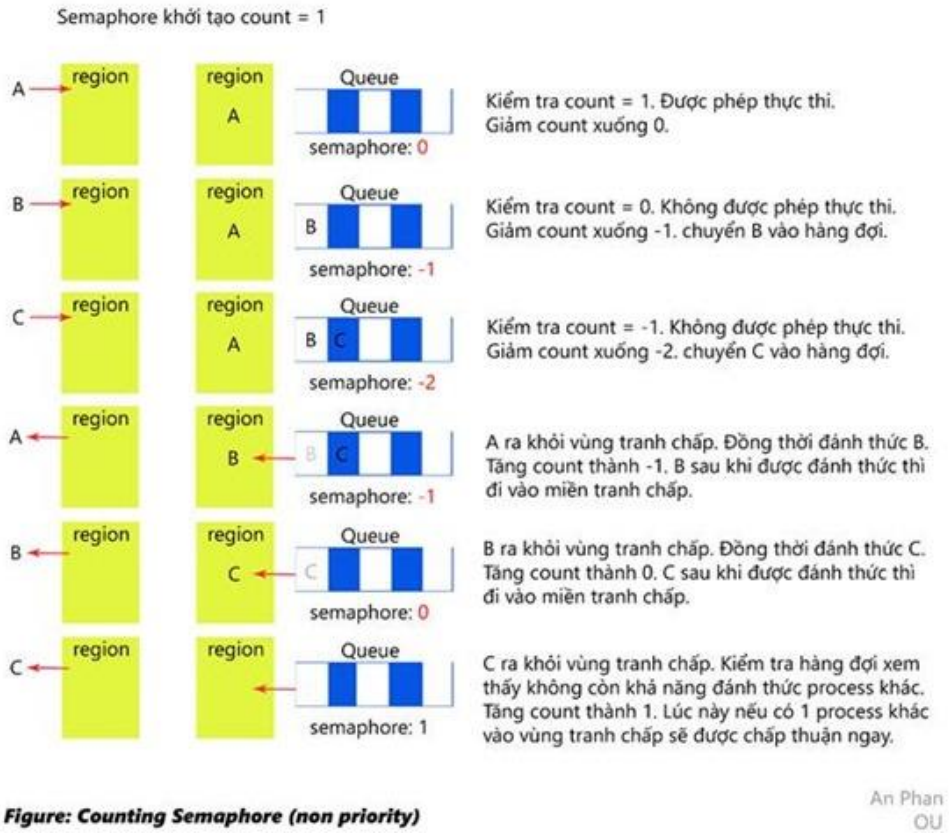


Figure: Counting Semaphore (non priority)

Hạn chế:

- Dễ xảy ra Deadlock
- Có thể xảy ra Starvation
- Semaphore có thể gây ra hiệu ứng đảo ngược ưu tiên

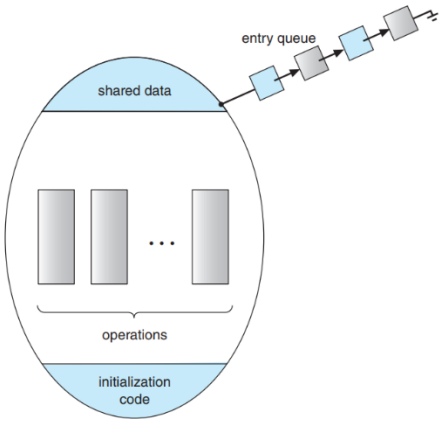
5.3.2 Monitors

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)



Mô hình 1 monitor đơn giản

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

Chương 6: Deadlocks

6.1 Tổng quan về deadlock

6.1.2 Định nghĩa:

- Một tiến trình gọi là **deadlock** 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 một deadlock
- 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
    - VD: Một tiến trình sẵn sàng để xử lý nhưng nó không bao giờ nhận được CPU

6.1.2 Mô hình hóa hệ thống

- Các loại tài nguyên, kí hiệu R1,R2,..Rm, bao gồm:
    - CPU cycle, không gian bộ nhớ, thiết bị I/O, file, semaphore,..
    - Mỗi loại tài nguyên Ri có Wi 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
    - Wail/ signal

6.2 Các tính chất của deadlock

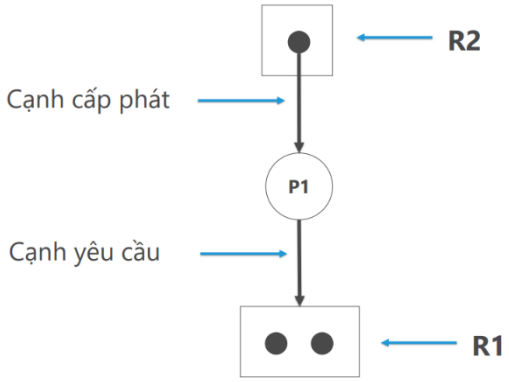
6.2.1 Điều kiện xảy ra deadlocks

- **Mutual exclution:** một tài nguyên chỉ có thể được giữ bởi một process tại một thời điểm

- **Hold & Wait:** Một tiến trình đang giữ ít nhất một tài nguyên và đợi thêm tài nguyên do quá trình khác giữ
- **No preemption:** Tài nguyên không thể bị lấy lại mà chỉ có thể được trả lại từ tiến trình đang giữ tài nguyên đó khi nó muốn
- **Circular wait:** Một tập hợp các process đang chờ đợi lẫn nhau ở dạng vòng tròn (chu trình)

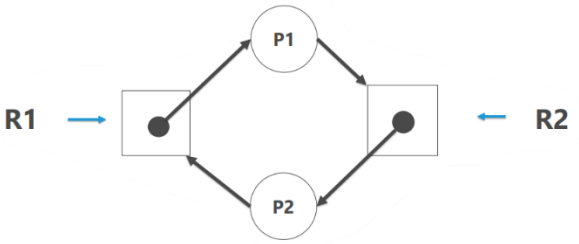
6.2.2 Đồ thị cấp phát tài nguyên(RAG)

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

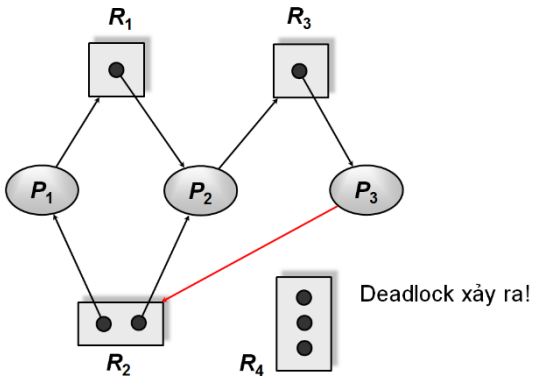


6.2.3 Mối liên hệ giữa RAG và deadlocks

- RAG không chứa chu trình => không có deadlock
- RAG chứa một(hay nhiều) chu trình
  - Nếu mỗi loại tài nguyên chỉ có một thực thể => deadlock **(1)**
  - Nếu mỗi loại tài nguyên có nhiều thực thể
    => có thể xảy ra deadlock **(2)**



(2) Một thực thể



(1) Nhiều thực thể

6.3 Phương pháp giải quyết deadlock

- Ngăn chặn deadlocks (deadlocks prevention)
- Tránh deadlocks (deadlocks avoidance)
- Cho phép hệ thống vào trạng thái deadlocks, nhưng sau đó phát hiện deadlocks và phục hồi hệ thống (deadlocks detection and recovery)
- Xem như deadlocks không bao giờ xảy ra trong hệ thống (deadlocks ignorance)
    - Được sử dụng nhiều nhất trong các cơ chế trên window và Linux,nếu xảy ra deadlocks cần khởi động lại máy

Khác biệt giữa **ngăn** và **tránh** deadlocks:

- Ngăn deadlocks: không cho phép(ít nhất) một trong 4 điều kiện cần cho deadlocks
- Tránh deadlock: Các quá trình cần cung cấp thông tin về tài nguyên nó cần để hệ thống cấp phát tài nguyên một cách thích hợp

6.3.1 Ngăn deadlocks

Ngăn deadlocks bằng cách ngăn một trong 4 điều kiện cần của deadlocks

- Ngăn **Mutual exclution**
    - Đối với tài nguyên không chia sẻ (printer): Không làm được
    - Đối với tài nguyên chia sẻ (read-only file): Không cần thiết
- Ngăn **Hold&Wait** !(Hold and 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 OS 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*
- Ngăn **No preemption:** A có tài nguyên, A yêu cầu tài nguyên khác nhưng chưa được cấp
    - Cách 1: Hệ thống lấy lại mọi tài nguyên A đang giữ
    - Cách 2: Hệ thống sẽ xem xét tài nguyên A yêu cầu

    - Được giữ bởi một tiến trình khác đang đợi thêm tài nguyên => hệ thống lấy lại và cấp cho A
    - Được giữ bởi tiến trình không đợi tài nguyên, A phải đợi => tài nguyên của A bị lấy
- Ngăn **Circular wait:** Gán số thứ tự cho tất cả các tài nguyên trong hệ thống. Một process không thể yêu cầu một tài nguyên ít ưu tiên hơn

6.3.2 Tránh deadlocks

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

Trạng thái safe và unsafe

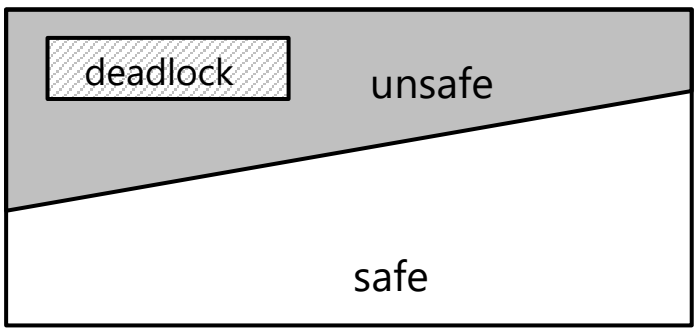
Một trạng thái của hệ thống gọi là an toàn (safe) nếu tồn tại một *chuỗi an toàn*

- Một chuỗi quá trình <P1,P2,…,Pn> là một chuỗi an toàn nếu: Với mọi i = 1,2,…,n yêu cầu tối đa về tài nguyên của Pi có thể được thỏa bởi:
    - Tài nguyên mà hệ thống đang có sẵn sàng
    - Cùng với tài nguyên mà tất cả các Pj (i<j) đang giữ

Nếu hệ thống ở trạng thái safe => không deadlocks

Nếu hệ thống đang ở trạng thái unsafe => có thể bị deadlocks

=> Tránh deadlocks bằng cách đảm bảo hệ thống không đi đến trạng thái unsafe



Ví dụ về chuỗi an toàn: Hệ thống có 12 tape drive và 3 tiến trình P_0, P_1, P_2 . Tại thời điểm t_0

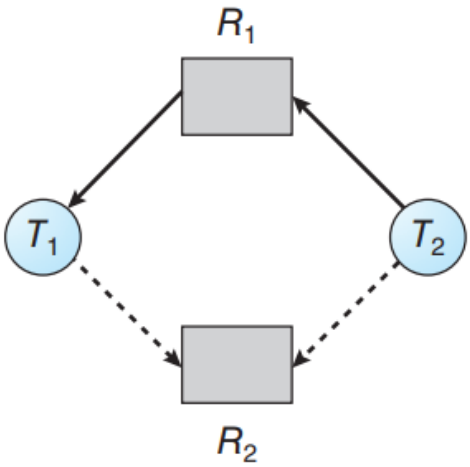
	Cần tối đa	Đang giữ	Cần thêm
P0	10	5	5
P1	4	2	2
P2	9	2	7

Giải: Còn 3 tape drive sẵn sàng => chuỗi $<P_1, P_0, P_2>$ là chuỗi an toàn => hệ thống an toàn

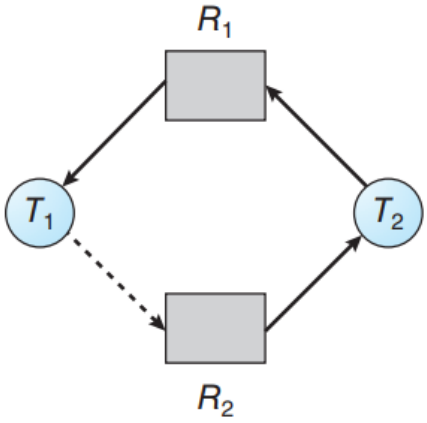
6.3.3 Các giải thuật tránh deadlocks

TH: Mỗi tài nguyên chỉ có một thực thể

Giải thuật đồ thị cấp phát tài nguyên



RAG để tránh deadlocks



Trạng thái không an toàn trong RAG

TH: Mỗi tài nguyên có nhiều thực thể

Giải thuật Banker

Điều kiện:

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

- Khi tiến trình yêu cầu tài nguyên thì có thể phải đợi
- Khi tiến trình đã có được đầy đủ tài nguyên thì phải hoàn trả trong một khoảng thời gian hữu hạn nào đó

Các bước thực hiện giải thuật Banker

b1.Tìm Need = Max – Allocation

b2.Tìm tiến trình P_i thỏa:

a. P_i chưa hoàn thành thực thi

b. $Need_i \leq$ Available

⇨ Nếu không còn P_i thỏa => chuyển sang b4

b3.Available = Available + $Allocation_i$

Thêm P_i vào chuỗi

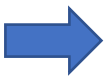
Chuyển sang b2

b4.Nếu chuỗi có tồn tại đủ hết các P

⇨ Hệ thống tồn tại chuỗi an toàn

Ví dụ: Sơ đồ cấp phát trong hệ thống

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



Available	Chuỗi an toàn
3 3 2	
5 3 2	P_1
7 4 3	P_3
7 4 5	P_4
10 4 7	P_2
10 5 7	P_0

Giải thuật Banker yêu cầu tài nguyên cho một tiến trình

Phương pháp giải:

b1. Nếu $\mathbf{request_i} \leq \mathbf{need_i}$ thì đến b2. Nếu không, báo lỗi vì tiến trình đã vượt yêu cầu tối đa

b2. Nếu $\mathbf{request_i} \leq \mathbf{available}$ thì qua b3. Nếu không, $\mathbf{P_i}$ phải chờ vì tài nguyên không còn đủ để cấp phát

b3. Giả định cấp phát tài nguyên đáp ứng yêu cầu của $\mathbf{P_i}$ bằng cách cập nhật trạng thái hệ thống như sau:

- $\mathbf{available = available - request_i}$
- $\mathbf{allocation_i = allocation_i + request_i}$
- $\mathbf{need_i = need_i - request_i}$

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

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

Available			
R1	**R2**	**R3**	**R4**
4	3	3	5

Giải: Request P3(1,3,1,2) ≤ Need P3(1,4,1,2)

Request P3(1,3,1,2) ≤ Available P3(4,3,3,5)

Giả sử hệ thống đáp ứng yêu cầu thêm tài nguyên (1,3,1,2) cho P3

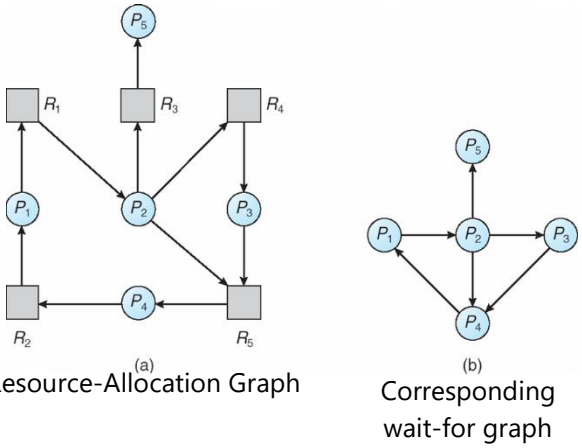
Tiến trình	Allocation				Max				**Need**				**Available**				
	R1	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**	
P1	3	1	1	2	5	3	4	3	2	2	3	1	3	0	2	3	
P2	1	1	2	1	3	4	6	1	2	3	4	0					
P3	3	4	5	7	3	5	5	7	0	1	0	0					
P4	3	5	2	2	4	6	4	5	1	1	2	3		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

Ta thấy: sau khi cấp phát cho P3 thì không thể tìm được bất kì chuỗi an toàn nào trong hệ thống => hệ thống không an toàn. Vậy hệ thống không thể đáp ứng yêu cầu của P3

6.3.4 Phát hiện và phục hồi Deadlock

Phát hiện Deadlock

- Đối với mỗi loại tài nguyên chỉ có một thực thể
  ⇨ Sử dụng wait-for graph



Resource-Allocation Graph

Corresponding wait-for graph

- Đối với mỗi loại tài nguyên có nhiều thực thể
  ⇨ Cách làm:
  - Thực hiện tương tự giải thuật **Banker**
  - Thay cột Need bằng cột Request
  - Liệt kê chuỗi an toàn => Tiến trình không có trong chuỗi an toàn thì deadlock xảy ra tại tiến trình đó

Phục hồi Deadlock

Khi deadlock xảy ra,để phục hồi

- Báo người vận hành
- Hệ thống tự động phục hồi bằng cách bẻ gãy chu trình deadlock
  - Chấm dứt một hay nhiều tiến trình (1)
  - Lấy lại tài nguyên từ một hay nhiều tiến trình (2)

(1)Chấm dứt quá trình:

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

- Sử dụng giải thuật phát hiện deadlock để xác định còn deadlock hay không

Chấm dứt dựa trên:

- 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

(2) Lấy lại tài nguyên từ một hay nhiều tiến trình

Lấy lại từ một tiến trình, cấp cho tiến trình khác cho đến khi không còn deadlock

Chọn "nạn nhân" để tối thiểu chi phí

Trở lại trạng thái trước deadlock(rollback)

- 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

Đói tài nguyên (starvation): Bảo đảm không có tiến trình nào luôn luôn bị lấy lại tài nguyên mỗi khi có deadlock

Chương 7: Quản lý bộ nhớ

Mục tiêu:

- Hiểu các cách khác nhau để quản lý bộ nhớ
- Hiểu tiếp cận quản lý bộ phân trang

7.1 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)

7.2 Địa chỉ bộ nhớ

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

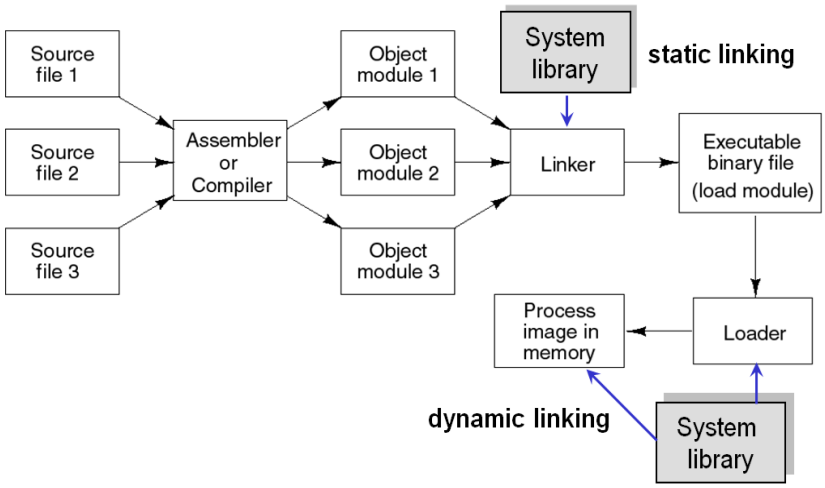
- Địa chỉ tuyệt đối(absolute address): địa chỉ tương đương với địa chỉ thực
- Địa chỉ vật lý = địa chỉ frame + offset

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

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

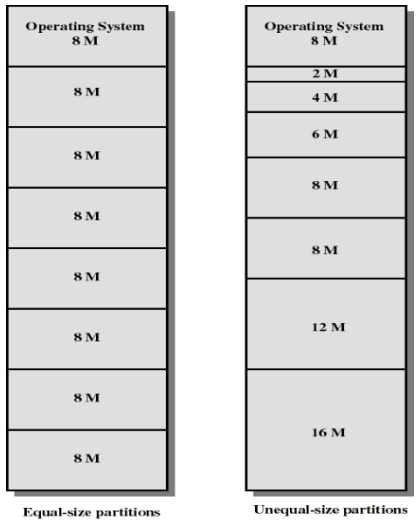


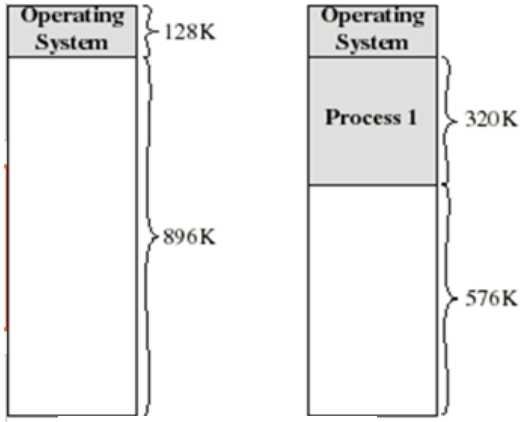
7.3 Các mô hình quản lý bộ nhớ

Khái niệm

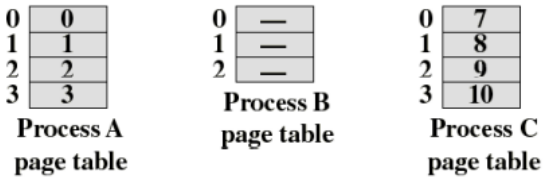
- Một process phải được nạp hoàn toàn vào bộ nhớ thì mới được thực thi
- Một số cơ chế quản lý bộ nhớ
  1. Phân chia cố định (fixed partitioning)
  2. Phân chia động (dynamic partitioning)
  3. Phân trang đơn giản (simple paging)

(1)





(2)



(3)

Hiện tượng phân mảnh bộ nhớ (fragmentation)

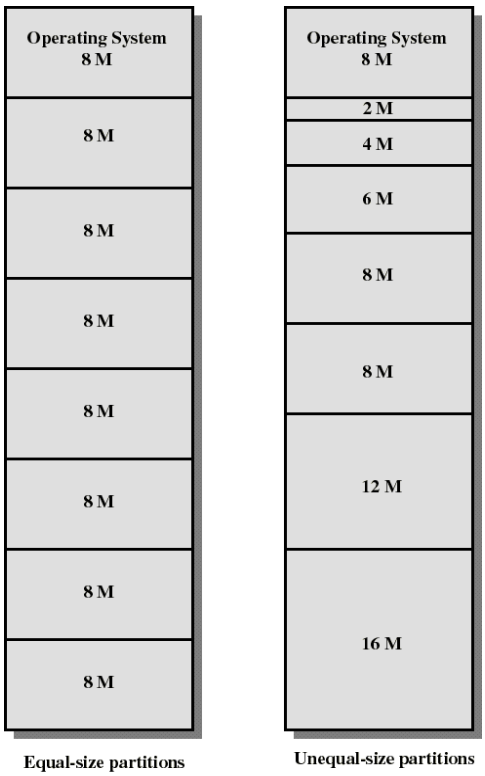
- **Phân mảnh ngoại (external fragmentation):** Kích thước không gian bộ nhớ trống đủ để thỏa mãn một yêu cầu cấp phát, tuy nhiên không gian nhớ này không liên tục
  - ⇨ Giải pháp: Dùng cơ chế kết khối (compaction) để gom lại thành vùng nhớ liên tục
- **Phân mảnh nội (internal fragmentation):** Kích thước vùng nhớ cấp phát có thể hơi lớn hơn vùng nhớ yêu cầu
  - ⇨ Giải pháp: Dùng chiến lược placement

Phân chia cố định (Fixed partitioning)

- Bộ nhớ chính chia thành nhiều phần, có kích thước bằng hoặc khác nhau **(4)**
- 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

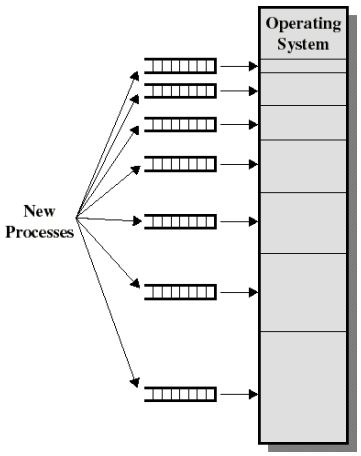
**Chiến lược placement*

- Với partition có cùng kích thước:
  - Còn partition trống => nạp vào
  - Không còn partition trống => Swap process đang bị blocked ra bộ nhớ phụ, nhương chỗ cho process mới

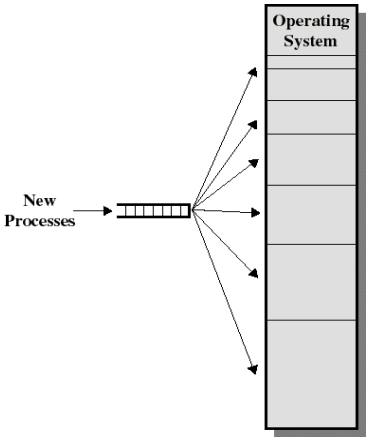


(4)

- Với partition khác kích thước:
  - Giải pháp 1: Sử dụng nhiều hàng đợi
    - Mỗi process xếp 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 một process vào bộ nhớ chính ⇒ chọn partition nhỏ nhất còn trống

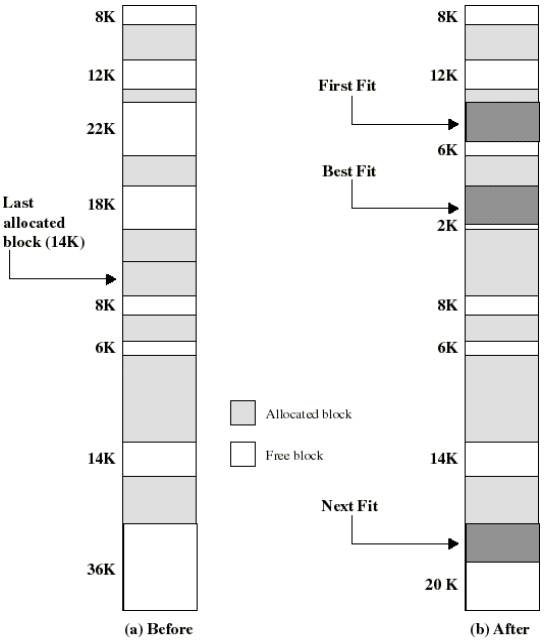


Phân chia động (dynamic partitioning)

- 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

**Chiến lược placement (giúp giảm chi phí compaction):*

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



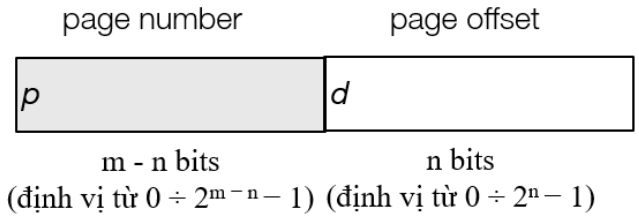
Example Memory Configuration Before and After Allocation of 16 Kbyte Block

Phân chia trang (Paging)

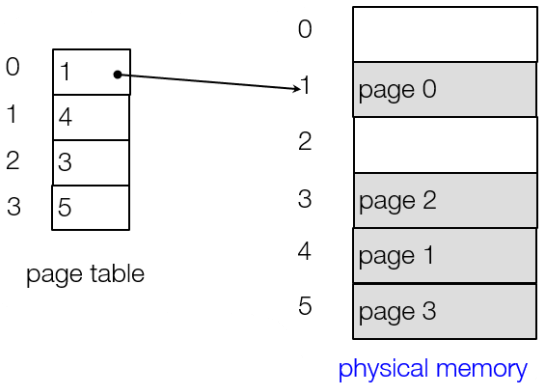
- Chia nhỏ bộ nhớ vật lý thành các khối nhỏ có kích thước bằng nhau (frames). Kích thước của frame là lỹ thừa của 2 (từ 512 bytes đến 16MB)
- Chia bộ nhớ luận lý của tiến trình thành các khối nhỏ có kích thước bằng nhau (pages). Địa chỉ luận lý gồm có:
  - Số hiệu trang(page number) (p)
  - Địa chỉ tương 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

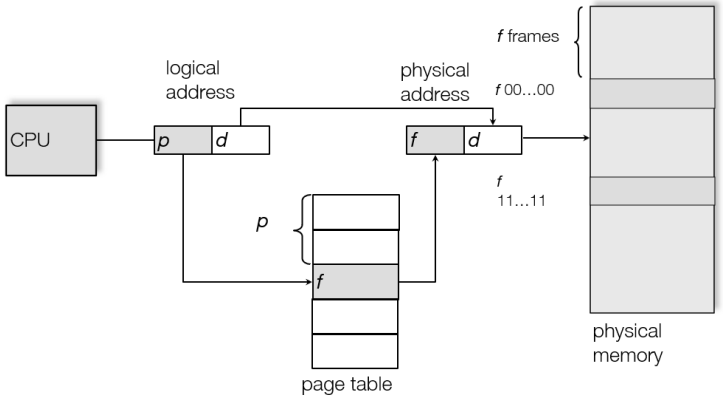
- Nếu kích thước của không gian địa chỉ ảo là 2^m, và kích thước của trang là 2^n (đơn vị là byte hay word tùy theo kiến trúc máy) thì



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

Translation look-aside buffers (TLBs)

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

Chương 8: Bộ nhớ ảo

8.1 Tổng quan

Bộ nhớ ảo (virtual memory): 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

8.2 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

8.3 Phân trang theo yêu cầu

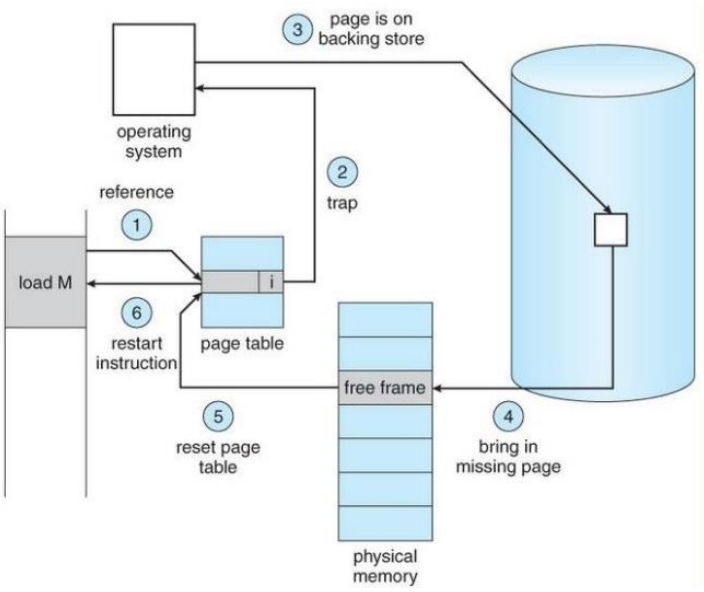
Các trang của tiến trình chỉ được nạp vào bộ nhớ chính khi được yêu cầu

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)

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.



*Nếu không tìm được frame trống

⇨ Dùng giải thuật thay trang chọn một trang hi sinh (victim page) Ghi victim page lên đĩa

8.4 Giải thuật thay trang

Dữ liệu cần biết

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

Xét ví dụ: Chuỗi truy xuất bộ nhớ: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

Giải thuật FIFO (first-in,first-out)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
*	*	*	*		*	*	*	*	*	*			*	*		.	*	*	*

⇨ 15 page-fault

Giải thuật OPT (optimal): Thay thế trang nhớ sẽ được tham chiếu trễ nhất trong tương lai

sử dụng 3 khung trang, khởi đầu đều trống:

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
*	*	*	*		*		*			*			*				*		

⇨ 9 page-fault

Giải thuật LRU (Least Recently Used)

sử dụng 3 khung trang, khởi đầu đều trống:

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
*	*	*	*		*		*	*	*	*			*		*		*		

⇨ 12 page-fault

Nghịch lý Belady

- Số page-fault tăng mặc dù quá trình đã được cấp nhiều frame hơn

8.5 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,...)

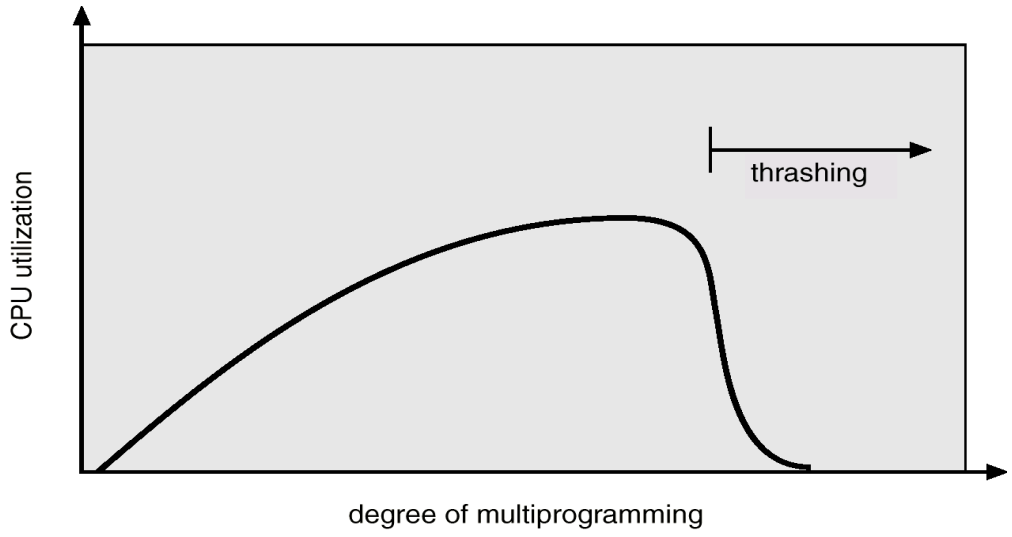
Chiến lược cấp phát động (variable-allocation)

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

8.6 Vấn đề Thrasing

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



Đã chỉnh sửa: 22/2/2023 (trang 17)