BÀI 4: QUẢN LÝ BỘ NHỚ

Nội dung

- Bộ nhớ vật lý.
- Tổ chức bộ nhớ ảo.
- Điều khiển bộ nhớ ảo.

Mục tiêu

- Mô tả các khái niệm cơ bản, các cơ chế tổ chức, chế độ bảo vệ và các chiến lược điều khiển bộ nhớ.
- Nắm được các khái niệm cơ bản về bộ nhớ ảo, hiểu được cách tổ chức theo trang, đoạn của bộ nhớ ảo.
- Nắm vững được các chiến lược điều khiển bộ nhớ ảo, chiến lược loại bỏ trang.
- Nắm vững các hình thức lưu trữ, tổ chức, quản lý bộ nhớ qua đó hiểu cách thức xây dựng hệ điều hành.

Thời lượng học

- 11 tiết.

TÌNH HUỐNG DẪN NHẬP

Tình huống

Nếu đặt toàn bộ không gian địa chỉ vào bộ nhớ vật lý, thì kích thước của chương trình bị giới hạn bởi kích thước vật lý. Thực tế, trong nhiều trường hợp, chúng ta không cần phải nạp toàn bộ chương trình vào bộ nhớ vật lý, vì tại một thời điểm chỉ có một chỉ thị của tiến trình được xử lý. Ví dụ, các chương trình đều có một đoạn code xử lý lỗi, nhưng đoạn code này hầu như rất ít sử dụng vì hiếm khi xảy ra lỗi, trong trường hợp này, không cần thiết phải nạp đoạn code xử lý lỗi ngay từ đầu.

Câu hỏi

Việc xử lý một chương trình có kích thước lớn chỉ với một vùng nhớ có kích thước nhỏ. Máy tính lưu trữ dữ liệu bằng cách nào?

4.1. Bộ nhớ thực

Mở đầu

Việc tổ chức và điều khiển bộ nhớ thực là một trong những yếu tố quan trọng nhất xác định cách xây dựng hệ điều hành. Để thực hiện các chương trình hay truy cập dữ liệu, chúng cần được nạp vào bộ nhớ vật lý. Bộ nhớ ngoài (bộ nhớ thứ cấp như ổ đĩa cứng) thường có dung lượng rất lớn và giá rẻ dùng để chứa chương trình và dữ liệu.

Tổ chức bộ nhớ

Trước kia bộ nhớ thực là tài nguyên đắt nhất. Do đó nó cần được tổ chức tốt để có thể sử dụng với hiệu quả cao nhất. Tổ chức bộ nhớ là cách mà chúng ta hình dung và sử dụng bộ nhớ vật lý. Ví dụ như chúng ta sẽ nạp vào bộ nhớ một chương trình hay nhiều chương trình cùng một lúc? Nếu như trong bộ nhớ có một số chương trình thì mỗi chương trình sẽ được cấp vùng nhớ bằng nhau hay chia bộ nhớ thành các phần (Section, Part) với kích thước khác nhau. Chúng ta có cần đòi hỏi các chương trình ứng dụng được thiết kế để nạp vào phần bộ nhớ cố định hay cho phép nạp vào bất cứ vùng nào phù hợp. Chúng ta có cần mỗi chương trình nằm trong một vùng nhớ liên tục hay có thể chia chương trình thành các khối nằm trong các vùng nhớ bất kỳ.

Điều khiển bộ nhớ

Không phụ thuộc vào cách tổ chức bộ nhớ, chúng ta cần giải quyết cần dùng các tiêu chuẩn nào để đạt được các thông số tối ưu. Các phương pháp điều khiển bộ nhớ xác định cách làm việc của bộ nhớ với tổ chức cụ thể nào đó trong các cách giải quyết khác nhau các vấn đề: chúng ta nạp chương trình vào chỗ nào, nếu như không còn đủ bộ nhớ trống thì chương trình nào đang nằm trong bộ nhớ sẽ phải đưa ra...

4.1.1. Phân cấp bộ nhớ

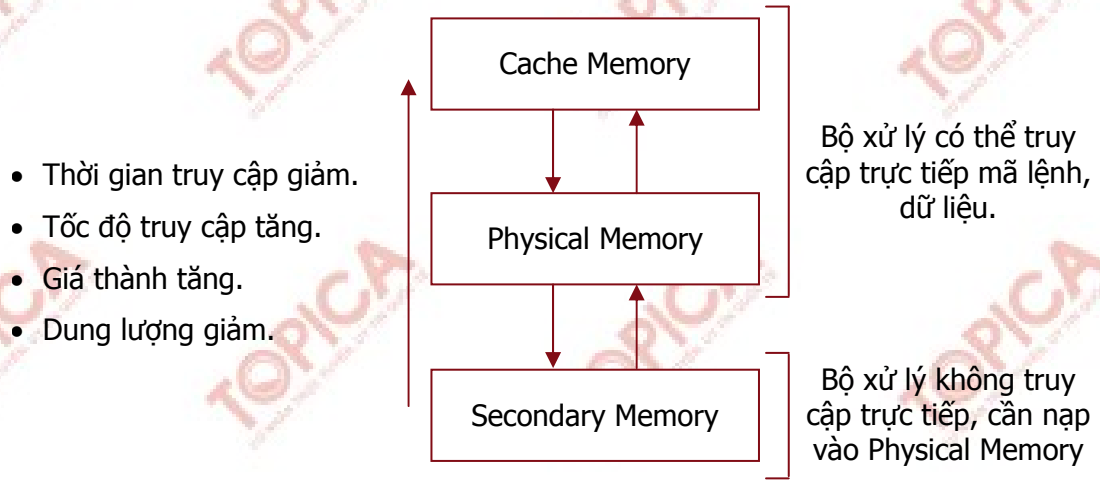
Vào những năm 1950 – 1960 bộ nhớ thực rất đắt. Do đó việc chọn lựa kích thước bộ nhớ thực cần phải tính toán trước. Khách hàng không muốn mua bộ nhớ lớn hơn anh ta có thể, mặt khác anh ta phải mua một số ít nhất nào đó để đảm bảo hoạt động của hệ điều hành và số lượng định trước các user. Vấn đề là xác định dung lượng bộ nhớ tối thiểu thoả mãn bài toán và đồng thời nằm trong khả năng tài chính cho phép.

Để có thể chạy chương trình hay truy cập dữ liệu, chúng cần phải được nạp vào bộ nhớ vật lý. Các chương trình và dữ liệu chưa cần có thể lưu trong bộ nhớ ngoài, khi cần thiết sẽ được nạp vào bộ nhớ vật lý. Bộ nhớ ngoài (đĩa cứng,...) thường rẻ hơn và có dụng lượng lớn hơn nhưng thời gian truy cập bộ nhớ thực lại nhanh hơn nhiều. Ví dụ về tốc độ truy cập, bộ nhớ vật lý: 60ns, bộ nhớ ngoài: 9ms.

Hệ thống với các lớp bộ nhớ có đặc tính tần suất trao đổi chương trình, dữ liệu giữa các lớp khác nhau tương đối lớn. Sự trao đổi đó cũng làm hao hụt tài nguyên hệ thống, ví dụ thời gian bộ xử lý,...

Vào những năm 1960 xuất hiện thêm một lớp nữa (ngoài bộ nhớ thực và bộ nhớ ngoài), đó là Cache Memory, cho phép làm tăng tốc độ và hiệu quả sử dụng bộ nhớ. Cache Memory có tốc độ truy cập nhanh hơn nhiều (15ns) so với bộ nhớ vật lý. Nhưng nó cũng đắt hơn nhiều, do đó trong hệ thống thông thường dung lượng Cache không lớn.

Cache Memory làm tăng thêm một lớp trao đổi nhưng chi phí đó được bù lại bởi tốc độ truy cập. Và do đó tốc độ của cả hệ thống được nâng lên nhiều.



Hình 4.1.1–1

Các chiến lược điều khiển bộ nhớ:

Để đảm bảo sử dụng tốt các tài nguyên giá trị, chúng cần được điều khiển một cách có hiệu quả. Các chiến lược điều khiển bộ nhớ theo hướng đảm bảo sử dụng tốt nhất bộ nhớ vật lý, và chia theo các hướng sau:

1) Chiến lược lựa chọn:

- Chiến lược lựa chọn theo yêu cầu (Demand Fetch).
- Chiến lược lựa chọn trước.

2) Chiến lược phân bố.

3) Chiến lược loại ra.

Mục đích của chiến lược lựa chọn là xác định xem khi nào phải nạp khối chương trình hay dữ liệu vào bộ nhớ vật lý. Trong nhiều năm người ta cho rằng cách tốt nhất là lựa chọn theo yêu cầu. Theo đó khối chương trình hay dữ liệu được nạp vào bộ nhớ khi chương trình đang hoạt động đòi hỏi đến. Bởi vì rằng nói chung khó mà nói trước được điều khiển sẽ được chuyển đến đâu (địa chỉ lệnh tiếp theo) và chi phí thêm gắn với việc dự đoán trước sẽ tăng đáng kể thời gian chờ. Còn ngày nay, nhiều nhà thiết kế tin rằng lựa chọn dự đoán trước hoàn toàn có thể đảm bảo tăng tốc độ của hệ thống.

Các chiến lược phân bố có mục đích xác định xem chương trình mới sẽ được nạp vào vị trí nào của bộ nhớ. Chúng ta sẽ xem xét một số chiến lược như chọn khối đầu tiên phù hợp (first suitable), chọn khối phù hợp nhất (most suitable) và khối ít phù hợp nhất (least suitable), theo kích thước các vùng trống.

Các chiến lược loại bỏ xác định xem khối chương trình hoặc dữ liệu nào sẽ bị loại ra khỏi bộ nhớ để giải phóng chỗ cho việc nạp chương trình hay dữ liệu.

Phân bố bộ nhớ liên tục và không liên tục

Trong các máy tính đầu tiên bộ nhớ được phân bố liên tục – tức là mỗi chương trình phải nằm trong một vùng nhớ. Chỉ sau khi xuất hiện khái niệm đa chương trình với phân đoạn thay đổi (Variable Partition Multi Programming) thì việc phân bố bộ nhớ không liên tục mới chứng tỏ sự hiệu quả của mình.

Trong phân bố bộ nhớ không liên tục, chương trình được chia làm nhiều phân đoạn (khối hay phân đoạn), và chúng có thể nằm tại các vùng nhớ khác nhau, không nhất thiết phải liền nhau. Về phía hệ điều hành, việc đảm bảo phân bố bộ nhớ không liên

tục là rất phức tạp, nhưng nó đem lại nhiều ưu thế: nếu bộ nhớ có nhiều vùng nhớ trống thay vì một vùng lớn thì hệ điều hành vẫn có thể nạp và thực hiện chương trình mà trong trường hợp ngược lại sẽ phải chờ.

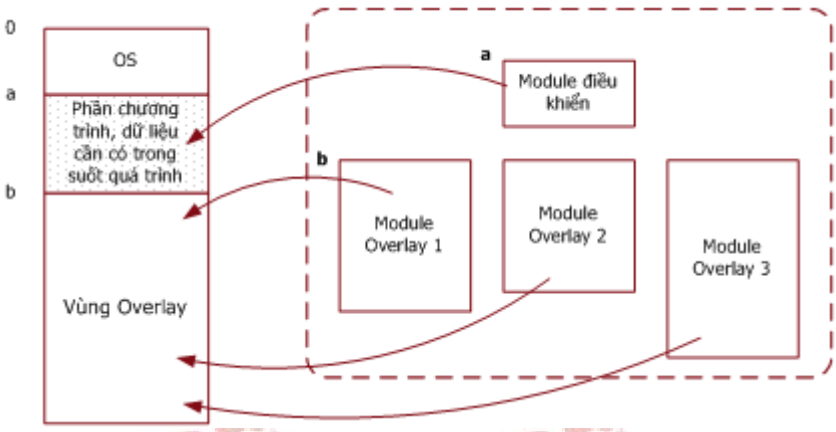
Phân bố bộ nhớ liên tục đối với một user

Trong các máy đầu tiên tại mỗi thời điểm chỉ có một user (một chương trình ứng dụng) và tất cả tài nguyên đều thuộc quyền của anh ta. Và tiền sử dụng máy được tính theo nguyên tắc đơn giản – bởi vì user toàn quyền sử dụng các tài nguyên do đó anh ta phải trả cho tất cả, không phụ thuộc việc chương trình của anh ta có sử dụng hết các tài nguyên hay không. Do đó cơ chế tính tiền thực hiện theo thời gian máy đã sử dụng. Trong các hệ làm việc trong chế độ phân chia thời gian thì việc tính toán sử dụng các thuật toán phức tạp hơn nhiều.

Đầu tiên mỗi user đều phải tự mình viết toàn bộ chương trình kể cả các hàm thực hiện vào/ra bằng ngôn ngữ máy. Sau đó thì phần lệnh thực hiện các thao tác vào/ra cơ bản được đưa vào IOCS – hệ thống điều khiển vào/ra, và người dùng không phải viết lại các hàm vào/ra mà chỉ phải gọi các hàm tương ứng của hệ thống. Điều đó làm đơn giản và tăng hiệu quả lập trình và có thể cho rằng đó là bắt đầu sự phát triển khái niệm về hệ điều hành hiện đại. Tổ chức bộ nhớ trong trường hợp phân bố liên tục đối với một người dùng được biểu diễn bằng hình 4.1.1–2

Hình 4.1.1–2

Thông thường thì kích thước chương trình phụ thuộc dung lượng bộ nhớ (hay không gian quản lý bởi hệ điều hành), nếu kích thước chương trình nhỏ hơn kích thước bộ nhớ thì không có vấn đề nảy sinh, còn khi kích thước chương trình vượt quá thì sao? Nhờ cơ chế overlay cho phép chúng ta có thể viết các chương trình lớn hơn giới hạn trên. Khái niệm của nó được thể hiện trong hình 4.1.1–3. Nếu như Module nào đó của chương trình không hoạt động trong cả quá trình thực hiện chương trình thì nó hoàn toàn có thể được lưu trong bộ nhớ ngoài, khi cần thiết mới được nạp vào bộ nhớ thực và sau khi kết thúc nó lại được ghi ra bộ nhớ ngoài, giải phóng bộ nhớ thực cho Module tiếp theo.



Hình 4.1.1–3: ***Cấu trúc Overlay***

Chế độ Overlay cho phép lập trình viên thiết kế các chương trình lớn vượt qua hạn chế bộ nhớ. Nhưng việc thiết kế overlay module đòi hỏi nhiều công sức và trí tuệ.

Các chương trình với cấu trúc overlay phức tạp rất khó thay đổi. Ngày nay chúng ta không còn phải dùng chế độ overlay để vượt qua giới hạn về bộ nhớ do đã có các hệ thống với bộ nhớ ảo – Virtual Memory.

Bảo vệ bộ nhớ trong hệ thống đơn nhiệm

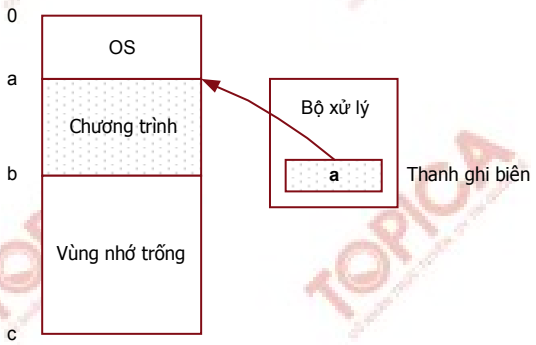
Trong hệ thống đơn nhiệm, user được cấp vùng nhớ liên tục và nói chung là có quyền điều khiển, truy cập toàn bộ bộ nhớ vật lý. Trong trường hợp này, bộ nhớ thường chia làm 3 vùng: vùng cho hệ điều hành, vùng cho chương trình ứng dụng và vùng trống. Vấn đề bảo vệ bộ nhớ trong trường hợp này cũng tương đối đơn giản. Bảo vệ bộ nhớ trong trường hợp này là bảo vệ hệ điều hành để chương trình ứng dụng không làm hỏng nó.

Nếu như chương trình hoạt động không đúng, điều đó có thể dẫn tới phá vỡ hệ điều hành, nếu như hệ điều hành bị hỏng đến mức chương trình khác ứng dụng không thể tiếp tục hoạt động thì người dùng ít nhất cũng nhận thấy, sửa lỗi và chạy lại chương trình. Trong trường hợp này thì sự cần thiết bảo vệ hệ điều hành không phải là rõ ràng.

Nếu trong trường hợp khác, chương trình ứng dụng phá hỏng hệ điều hành một cách 'tinh vi'. Ví dụ như nó 'vô tình' làm thay đổi một hàm vào/ra của hệ thống. Khi đó chương trình có thể vẫn hoạt động nhưng tất cả kết quả có thể bị mất. Còn có thể xảy ra trường hợp tồi tệ hơn là hệ thống đưa ra kết quả sai mà việc phát hiện ra không dễ dàng.

Do đó rõ ràng cần phải bảo vệ hệ điều hành. Việc bảo vệ có thể thực hiện bằng thanh ghi biên nằm trong bộ xử lý (hình 4.1.1–4)

Thanh ghi biên chứa địa chỉ thấp nhất của lệnh. Nếu như user cố gắng truy cập vào vùng của hệ điều hành thì lệnh đó bị cấm và sẽ đưa ra thông báo lỗi.



Hình 4.1.1–4

Tất nhiên, user cũng cần truy cập đến hệ điều hành, ví dụ như thực hiện các thao tác vào/ra. Để phục vụ điều này, user có thể dùng các lệnh đặc biệt, với chúng, anh ta có thể yêu cầu dịch vụ nào đó của hệ điều hành (ví dụ như ngắt SVC). Trong trường hợp này, hệ điều hành thực hiện dịch vụ được yêu cầu và sau đó trả lại điều khiển cho chương trình ứng dụng.

Với việc hệ điều hành ngày càng phức tạp, cần phải có các cơ chế tốt hơn để bảo vệ hệ điều hành đối với user cũng như giữa các user với nhau.

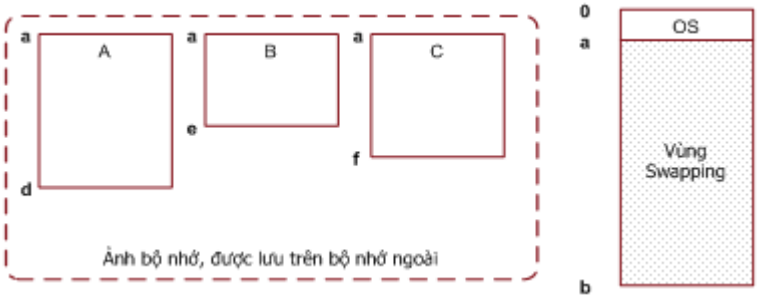
4.1.2. Hoán chuyển bộ nhớ (Swapping)

Trong các hệ đa nhiệm đã xem xét ở trên thì chúng ta đều giả sử rằng chương trình ứng dụng luôn nằm trong bộ nhớ đến khi nó kết thúc. Còn có cơ chế khác gọi là Swapping mà không cần điều kiện đó.

Trong một số hệ thống dùng Swapping (hình.4.1.2) thì mỗi thời điểm trên cả bộ nhớ chỉ có một chương trình ứng dụng. Chương trình đó thực hiện đến khi có thể sau đó nó giải phóng cả bộ xử lý và bộ nhớ để cho chương trình tiếp theo. Như thế, toàn bộ bộ nhớ được dành cho một chương trình trong khoảng thời gian ngắn, và sẽ bị đưa ra khỏi bộ nhớ, chương trình tiếp theo được nạp vào bộ nhớ. Bình thường thì một chương trình trong quá trình chạy từ đầu đến lúc kết thúc sẽ bị swap nhiều lần.

Trong các hệ thống phân chia thời gian có sử dụng Swapping, khi số chương trình khá ít thì hệ thống có thể đảm bảo thời gian trả lời chấp nhận được, còn khi số chương trình nhiều thì các nhà thiết kế hiểu rằng cần có các phương pháp và công cụ hiệu quả hơn. Trên cơ sở hệ thống với Swapping đầu những năm 1960, người ta đã xây dựng nhiều hệ thống với sự tổ chức bộ nhớ theo trang (Page) và ngày nay trở nên thông dụng.

Cũng đã có xây dựng các hệ thống phức tạp hơn với Swapping, trong đó cho phép phân bố trong bộ nhớ nhiều chương trình. Trong các hệ thống đó, chương trình bị đưa ra khỏi bộ nhớ chỉ khi vùng bộ nhớ của nó cần thiết để nạp chương trình khác. Trong trường hợp dung lượng bộ nhớ khá lớn thì các hệ thống đó hoạt động tốt hơn đáng kể do giảm được thời gian thực hiện Swapping.



Hình 4.1.2

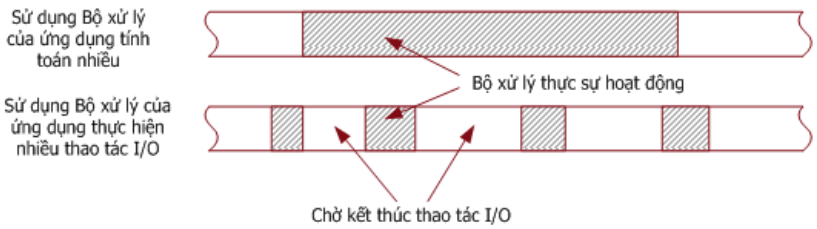
Hệ thống đa nhiệm với Swapping, tại mỗi thời điểm chỉ có một chương trình trong bộ nhớ. Chương trình đó tiếp tục đến khi có một trong các sự kiện sau xảy ra:

- Xuất hiện tín hiệu Timer.
- Nó kết thúc.

Vùng Swapping của chương trình đó sẽ được copy vào bộ nhớ ngoài, còn ảnh bộ nhớ của chương trình tiếp theo sẽ được nạp vào vùng Swapping. Chương trình thứ hai tiếp tục được thực hiện, quá trình cứ thế tiếp tục.

4.1.3. Phân đoạn cố định

Đối với các hệ thống đơn nhiệm và ngay cả hệ thống xử lý gói (Package Processing) thì vẫn bị lãng phí một phần đáng kể tài nguyên. Trên hình 4.1.3 ta thấy rằng chương trình chỉ thực sự sử dụng bộ xử lý khi mà nó không thực hiện thao tác vào/ra. Sau khi bắt đầu thao tác vào/ra chương trình không thể tiếp tục cho đến khi kết thúc thao tác. Tốc độ thực hiện thao tác vào/ra thường chậm hơn nhiều so với tốc độ của bộ xử lý. Như vậy bộ xử lý bị bỏ phí trong suốt thời gian chờ thực hiện tác vụ vào/ra, và phần này chiếm đáng kể thời gian chung.



Hình 4.1.3: ***Sử dụng bộ xử lý trong hệ thống đơn nhiệm***

Các nhà thiết kế thấy rằng hoàn toàn có thể tăng hiệu quả sử dụng bộ xử lý. Họ đã thực hiện các hệ thống đa nhiệm, trong đó các user cùng 'cạnh tranh' để có được tài nguyên. Chương trình đang chờ kết thúc thao tác vào/ra sẽ phải nhường bộ xử lý cho chương trình đã sẵn sàng hoạt động (tất nhiên nếu có chương trình như thế). Do đó

đảm bảo khả năng cùng thực hiện thao tác vào/ra và bộ xử lý – nâng cao hiệu quả sử dụng bộ xử lý và cả hệ thống.

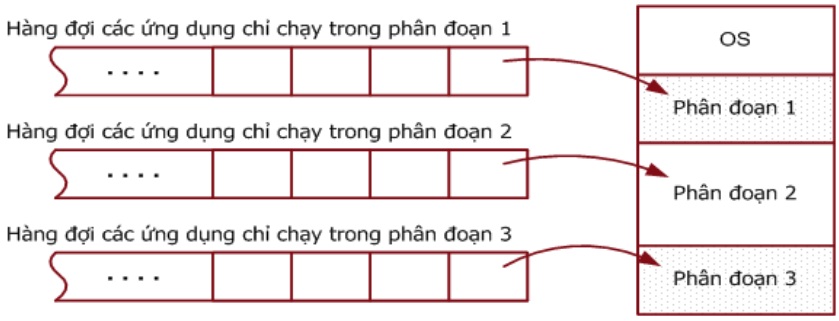
Các ưu thế của Multiprogramming chỉ có thể tận dụng hết khi trong bộ nhớ có nhiều chương trình. Nhờ đó khi một chương trình thực hiện thao tác vào/ra thì bộ xử lý có thể chuyển sang phục vụ chương trình khác và thực hiện tính toán với thời gian trễ nhỏ nhất. Khi chương trình thứ hai giải phóng bộ xử lý thí đến chương trình thứ ba khác lại có thể sẵn sàng sử dụng nó.

Multiprogramming đòi hỏi dung lượng bộ nhớ lớn hơn so với trường hợp đơn nhiệm nhưng chi phí đó được bù lại bởi việc sử dụng hiệu quả các tài nguyên khác (bộ xử lý, thiết bị vào/ra).

4.1.3.1. Đa nhiệm với phân đoạn cố định (Fixed Partition Multiprogramming) dịch và nạp theo chương trình theo địa chỉ tuyệt đối

Trong các hệ đa nhiệm đầu tiên, bộ nhớ được chia thành các phần (phân đoạn) với kích thước cố định. Trong mỗi đoạn chỉ có thể nạp một chương trình. Còn bộ xử lý nhanh chóng chuyển từ chương trình này sang chương trình khác, do đó tạo ra hiệu quả là tất cả chương trình dường như được thực hiện đồng thời.

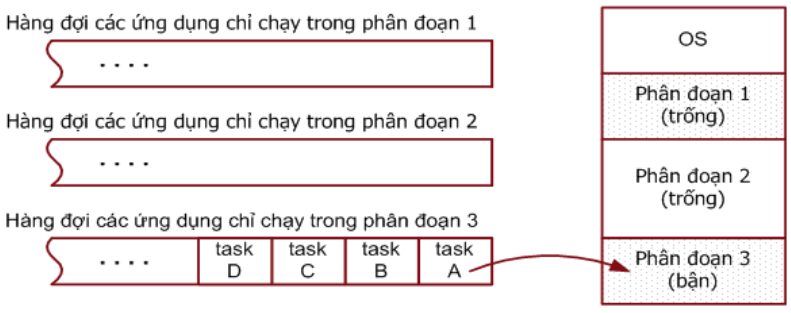
Việc biên dịch chương trình được thực hiện bởi Assembler và Compiler với địa chỉ tuyệt đối bị giới hạn bởi việc chương trình chỉ có thể chạy trong một phân đoạn cụ thể (hình 4.1.3.1–1).



Hình 4.1.3.1–1

Nếu như chương trình đã sẵn sàng nhưng phân đoạn của nó đã có chương trình khác thì nó phải đợi dù rằng các phân đoạn khác có thể trống (hình 4.1.3.2). Điều này dẫn tới sự lãng phí bộ nhớ nhưng việc thiết kế hệ điều hành lại tương đối đơn giản.

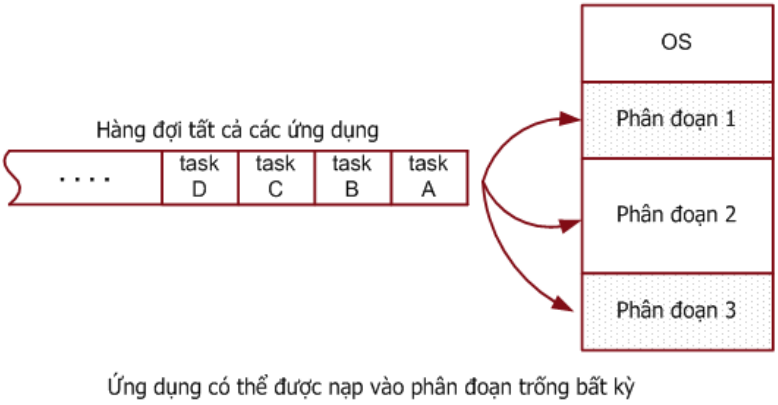
Trên hình 4.1.3.1–2 thể hiện tình huống xấu nhất của phương pháp phân bố bộ nhớ liên tục với phân đoạn cố định và chương trình được dịch và nạp vào địa chỉ tuyệt đối (phân đoạn cố định). Cả hai phân đoạn một và hai đều bỏ phí còn hàng chờ của phân đoạn ba quá nhiều trong khi chúng đã có thể được nạp vào phân đoạn 1 và 2.



Hình 4.1.3.1–2

4.1.3.2. Đa nhiệm với phân đoạn cố định: dịch và nạp chương trình các module tự do

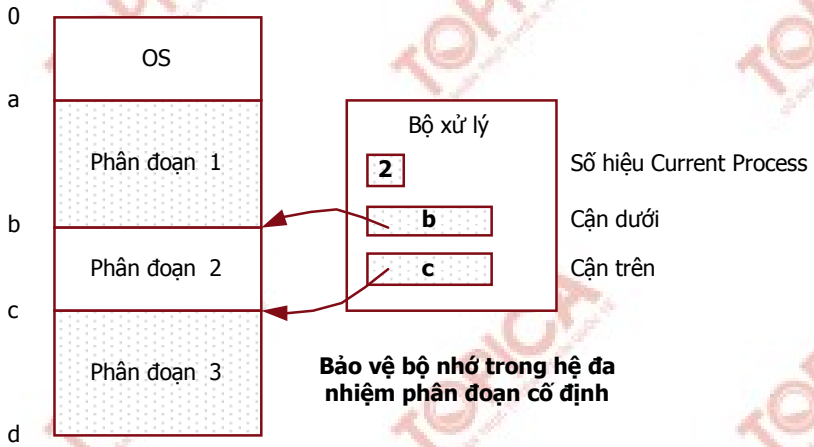
Các chương trình dịch, Assembler và Loader tự do được sử dụng để dịch và nạp các chương trình có thể thực hiện tại bất cứ phân đoạn trống nào đủ lớn (hình 4.1.3.2). Phương pháp này khắc phục được một số nhược điểm về sử dụng bộ nhớ trong trường hợp trước (đa nhiệm với phân đoạn cố định, dịch và nạp chương trình theo địa chỉ tuyệt đối). Nhưng để đạt được điều đó, việc thiết kế lại phức tạp hơn nhiều.



Hình 4.1.3.2

4.1.3.3. Bảo vệ bộ nhớ trong các hệ đa nhiệm

Trong các hệ thống đa nhiệm với phân bố bộ nhớ liên tục (Contiguous Memory Allocation), người ta thường sử dụng các thanh ghi biên để bảo vệ bộ nhớ. Hai thanh ghi chỉ ra cận dưới và cận trên của phân đoạn, hoặc một cận và kích thước của phân đoạn; chương trình có yêu cầu truy cập đến các dịch vụ của hệ điều hành phải dùng ngắt SVC để yêu cầu.



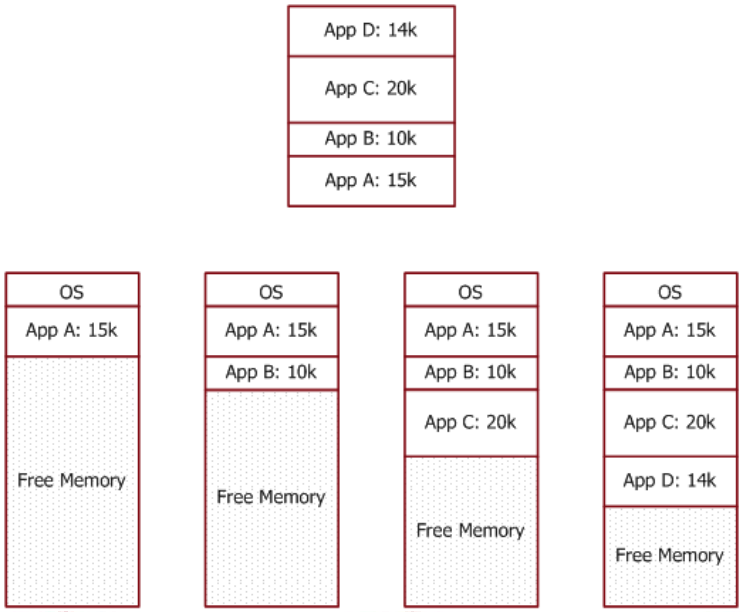
Hình 4.1.3.3

4.1.3.4. Vấn đề chia nhỏ bộ nhớ (Fragmentation) trong đa chương trình với phân đoạn cố định

Fragmentation xảy ra trong bất kỳ hệ thống nào không phụ thuộc vào tổ chức bộ nhớ. Trong các hệ đa nhiệm với các phân đoạn cố định, Fragmentation xảy ra do các chương trình ứng dụng không chiếm hết toàn bộ phân đoạn của nó hoặc là do các phân đoạn quá nhỏ để có thể nạp các chương trình đang chờ vào đó.

4.1.4. Phân đoạn thay đổi

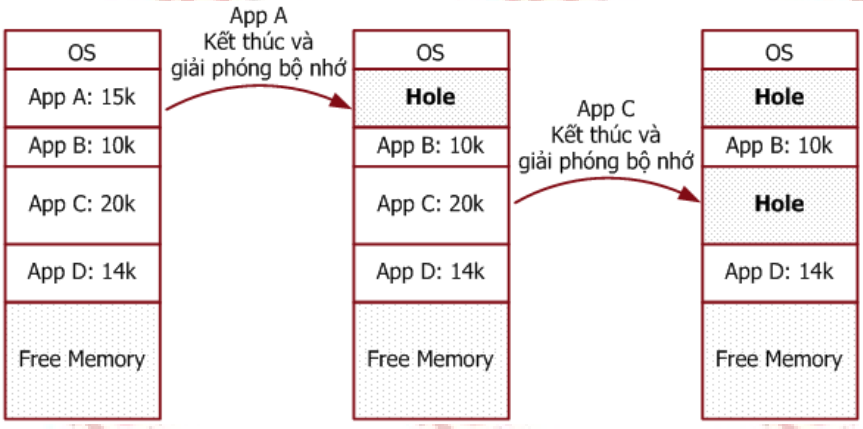
Khi xem xét các vấn đề của đa nhiệm với phân đoạn cố định, các kỹ sư thiết kế hệ điều hành thấy rằng tốt hơn là cho phép các chương trình chỉ chiếm bộ nhớ kích thước vừa đủ kích thước nó cần. Không chia bộ nhớ thành các đoạn cố định mà có kích thước thay đổi theo chương trình ứng dụng. Phương pháp đó gọi là đa nhiệm với các phân đoạn thay đổi. Sự phân bố bộ nhớ ban đầu được thể hiện trong hình 4.1.4–1.



Hình 4.1.4–1

Chúng ta chỉ xem xét phương pháp phân chia bộ nhớ liên tục tức là mỗi chương trình nạp vào vùng nhớ liên tục. Trong phương pháp đa nhiệm với các phân đoạn thay đổi chúng ta không đề ra bất cứ giả sử gì về chương trình (trừ khi kích thước của chúng không vượt quá kích thước bộ nhớ). Các chương trình khi được nạp (thực hiện) sẽ được cấp vùng nhớ chúng cần. Mỗi phân đoạn có kích thước đúng bằng kích thước chương trình nằm trong đó.

Mỗi sơ đồ tổ chức bộ nhớ đều có sự lãng phí nhất định. Trong trường hợp đa nhiệm với các phân đoạn thay đổi, sự lãng phí này xuất hiện khi các chương trình kết thúc và trong bộ nhớ xuất hiện các vùng trống (Hole) như hình 4.1.4–2. Các vùng trống này có thể sử dụng để nạp các chương trình khác nhưng dù sao vẫn sẽ còn các vùng trống vì không phải các chương trình đều có kích thước đúng bằng vùng trống. Như vậy đa nhiệm với các phân đoạn thay đổi vẫn không tránh khỏi lãng phí bộ nhớ.

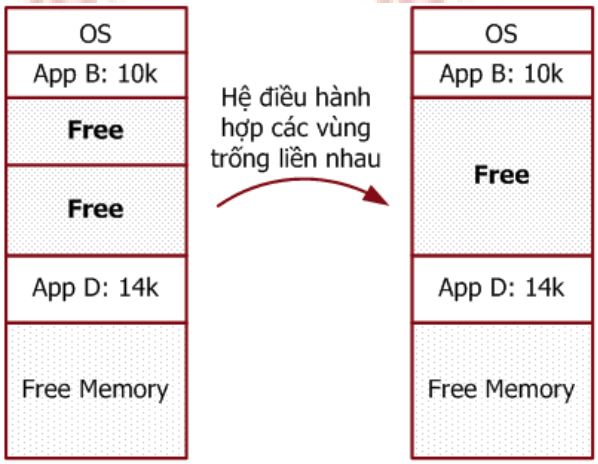


Hình 4.1.4–2

4.1.4.1. Hợp nhất các vùng trống liền nhau

Để khắc phục tình trạng bộ nhớ bị chia nhỏ chúng ta có thể dồn các vùng nhớ trống kề nhau.

Trong đa nhiệm với các phân đoạn thay đổi, khi một chương trình kết thúc chúng ta có thể kiểm tra xem vùng được giải phóng có nằm liền kề với vùng trống khác không? Nếu tồn tại thì chúng ta có thể hoặc đưa thêm vào danh sách các phân đoạn trống thêm một bản ghi nữa hoặc nối liền với vùng trống liền kề thành một vùng trống. Quá trình hợp nhất (nối liền) các vùng trống kề nhau biểu diễn trên hình 4.1.4.1. Nhờ sự hợp nhất này mà chúng ta tạo được các vùng nhớ liên tục với kích thước lớn nhất có thể.

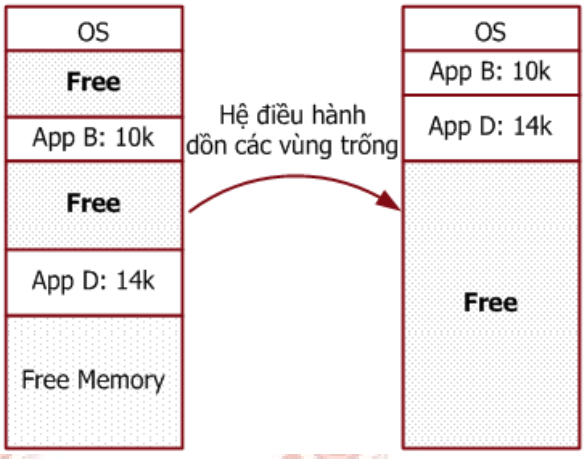


Hình 4.1.4.1

4.1.4.2. Dồn bộ nhớ

Như chúng ta thấy sau khi thực hiện hợp nhất các vùng trống (Holes) liền kề nhau nhìn tổng thể thì trong bộ nhớ vẫn tồn tại các vùng trống – cho dù ít hơn và kích thước lớn hơn. Đôi khi chương trình cần thực hiện tiếp theo lại lớn đến mức không có vùng trống nào đủ lớn để nạp cho dù tổng cộng kích thước các vùng trống vẫn lớn hơn cần thiết.

Vấn đề này được khắc phục nhờ phương pháp gọi là dồn bộ nhớ (hình 4.1.4.2) và bản chất là chuyển (dồn) tất cả các phân đoạn đang có chương trình về một phía bộ nhớ. Nhờ đó thay vì có nhiều phân đoạn trống (vùng trống) vụn vặt chúng ta có được một vùng trống lớn duy nhất trong bộ nhớ. Khi đó chương trình tiếp theo có xác suất lớn sẽ có đủ bộ nhớ cần thiết để chạy. Đôi khi phương pháp 'dồn bộ nhớ' còn gọi là 'dọn rác'.



Hình 4.1.4.2

Tuy có nhiều ích lợi nhưng phương pháp dồn bộ nhớ vẫn có những nhược điểm nhất định:

- Nó cũng làm hao phí tài nguyên hệ thống có thể được sử dụng vào mục đích khác tốt hơn.
- Vào thời gian dồn bộ nhớ, hệ thống phải dừng tất cả các công việc khác. Kết quả dẫn tới thời gian không thể dự đoán trước trong việc phản ứng lại các sự kiện (trả lời user trong chế độ dialog chẳng hạn) và điều đó có thể không chấp nhận được trong các hệ thống thời gian thực.
- Dồn bộ nhớ dịch chuyển các task trong bộ nhớ. Điều đó có nghĩa là thông tin về sự phân bố chương trình phải được lưu lại ở dạng nào đó.
- Trong trường hợp các bài toán thực hiện liên tục thì tần số thực hiện thao tác dồn bộ nhớ có thể trở nên thường xuyên và hao phí thực hiện nó có thể vượt quá lợi ích nó mang lại.

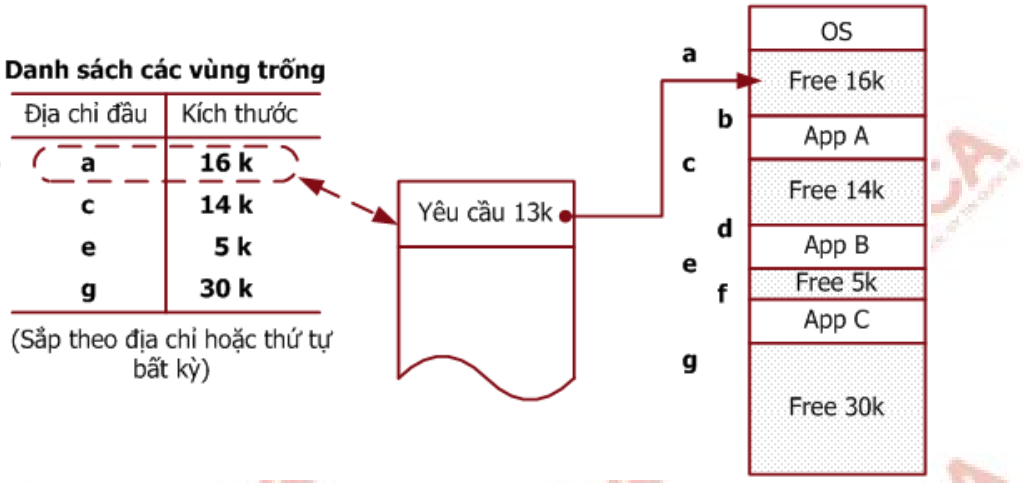
4.1.4.3. Các chiến lược phân bố thông tin trong bộ nhớ

Các chiến lược phân bố bộ nhớ được áp dụng để xác định chương trình và dữ liệu sẽ được tiếp tục nạp vào vùng nào của bộ nhớ. Chúng ta thường gặp 3 chiến lược được biểu diễn trên hình 4.1.4.3.

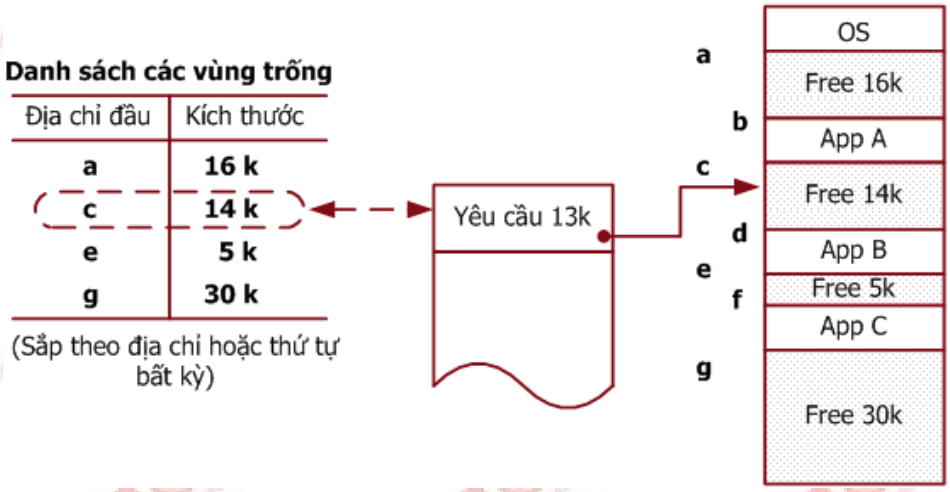
1) Chiến lược 'First Suitable': chương trình sẽ được nạp vào vùng trống gặp đầu tiên (trong danh sách các vùng trống) có kích thước đủ lớn. Chiến lược này có vẻ trực giác và thực tế vì nó cho phép tìm lời giải nhanh nhất.

2) Chiến lược 'Most Suitable': Chương trình được nạp vào vùng trống 'vừa nhất' do đó không gian lãng phí là ít nhất. Với nhiều người thì chiến lược này trực giác có vẻ là đúng nhất.

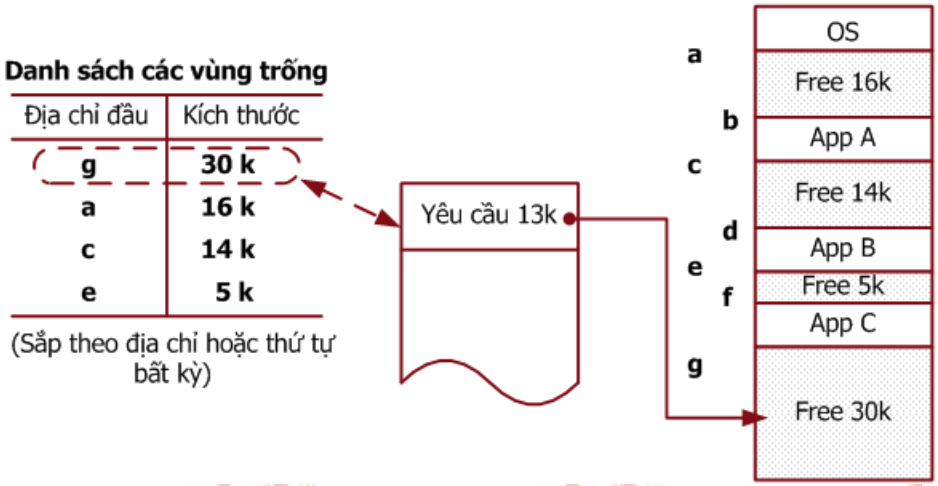
3) Chiến lược 'Least Suitable': đầu tiên thì chiến lược này có vẻ lạ lùng, nhưng xem xét kỹ thì nó cũng có các ưu điểm nhất định. Với chiến lược này chương trình nạp vào vùng trống lớn nhất. Ưu điểm là sau khi đã nạp chương trình, vẫn còn đủ không gian trống tương đối lớn để có thể nạp thêm chương trình mới khá lớn.



Hình 4.1.4.3a: ***Chiến lược FS***



Hình 4.1.4.3b: ***Chiến lược MS***



Hình 4.1.4.3c: ***Chiến lược LS***

4.2. Bộ nhớ ảo

4.2.1. Khái niệm

Thuật ngữ bộ nhớ ảo (Virtual Memory) thường gắn liền với khả năng đánh địa chỉ cho không gian nhớ lớn hơn nhiều dung lượng bộ nhớ vật lý. Virtual Memory xuất hiện (ứng dụng) lần đầu vào năm 1960 tại trường đại học Manchester và sau đó nhanh chóng được phổ biến.

Có hai phương pháp được chấp nhận một cách tự nhiên khi thực hiện (tổ chức) bộ nhớ ảo: đó là tổ chức theo trang (Page) và theo phân đoạn (segment). Trong một số hệ thống, ứng dụng một trong hai phương pháp đó còn trong một số khác thì áp dụng tổ hợp cả hai phương pháp.

Tất cả các cơ chế bộ nhớ ảo đều đặc trưng bởi tính chất rằng địa chỉ được tính bởi chương trình, không nhất thiết phải trùng với địa chỉ bộ nhớ (vật lý). Trong thực tế địa chỉ ảo thể hiện không gian lớn hơn nhiều so với thực tế bộ nhớ vật lý.

Quá trình phát triển các dạng tổ chức bộ nhớ

Hình.4.2.1a biểu diễn quá trình phát triển từ các hệ thống bộ nhớ thực đơn nhiệm (1 user) đến các hệ thống bộ nhớ ảo sử dụng tổ hợp cả hai phương pháp Page và segment.

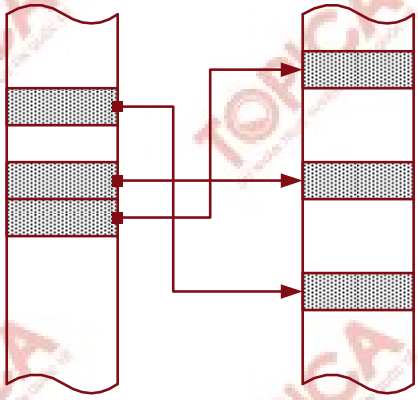
<table>
<tr><td colspan="3">Bộ nhớ vật lý</td><td colspan="3">Bộ nhớ ảo</td></tr>
<tr><td rowspan="3">Hệ thống đơn nhiệm</td><td colspan="2">Hệ đa nhiệm, bộ nhớ vật lý</td><td colspan="3">Hệ đa nhiệm, bộ nhớ ảo</td></tr>
<tr><td>Phân đoạn cố đinh</td><td>Phân đoạn thay đổi</td><td rowspan="2">Tổ chức theo trang</td><td rowspan="2">Tổ chức theo Segment</td><td rowspan="2">Tổ chức kết hợp</td></tr>
<tr><td>Địa chỉ tuyệt đối</td><td>Địa chỉ thay đổi</td></tr>
</table>

Hình 4.2.1a

Bộ nhớ ảo: các khái niệm cơ bản

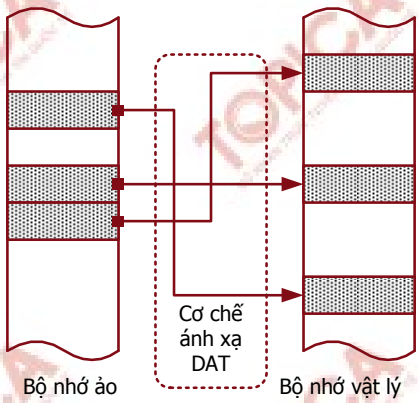
Cốt lõi các khái niệm về bộ nhớ ảo là ở chỗ địa chỉ mà tiến trình có thể truy cập gọi là không gian địa chỉ ảo V của tiến trình đó, còn vùng địa chỉ thực tồn tại trong bộ nhớ gọi là không gian địa chỉ thực R.

Dù rằng tiến trình làm việc với địa chỉ ảo thì thật sự chúng phải làm việc với bộ nhớ. Do đó vào thời gian thực hiện tiến trình, các địa chỉ ảo cần phải biến đổi thành địa chỉ thực, ngoài ra việc xử lý phải nhanh chóng vì nếu không thì hiệu quả của máy tính có thể giảm tới mức không chấp nhận được và lợi ích của tổ chức bộ nhớ ảo không thể ứng dụng được.



Hình 4.2.1b: ***Ánh xạ các ô nhớ từ bộ nhớ thực sang bộ nhớ ảo***

Để xác định ánh xạ giữa địa chỉ thực và ảo, người ta đã thiết kế các phương pháp khác nhau. Cơ chế biến đổi địa chỉ động (DAT–Dynamic Address Translation) đảm bảo sự biến đổi địa chỉ ảo sang địa chỉ thực vào thời gian thực hiện tiến trình. Tất cả các hệ thống đó đều có tính chất chung: đó là các địa chỉ ảo liền nhau chưa chắc đã tương ứng với các địa chỉ thực liền nhau (hình 4.2.1c). Như thế, user không cần phải quan tâm đến việc nạp chương trình hay dữ liệu của mình trong bộ nhớ (vật lý). Anh ta có được khả năng viết chương trình một cách tự nhiên hơn, chỉ phải quan tâm đến các vấn đề thiết kế chương trình, thuật toán mà không cần để ý đến chi tiết cụ thể của thiết bị. Khi đó máy tính (có thể) được coi như là một cài gì đó trừu tượng hơn – một công cụ logic nào đó để chạy chương trình chứ không phải như một cái máy vật lý với các chi tiết cần để ý, làm khó khăn cho quá trình thiết kế chương trình.



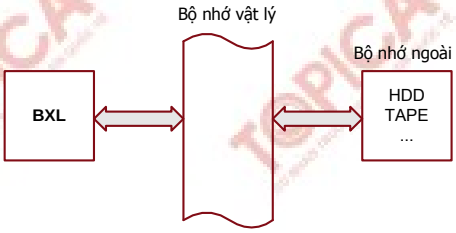
Hình 4.2.1c

Tổ chức bộ nhớ nhiều lớp

Nếu như chúng ta đề ra rằng không gian địa chỉ ảo của user sẽ lớn hơn không gian địa chỉ thực, và nếu chúng ta định hướng rằng hệ thống sẽ làm việc tốt trong chế độ đa nhiệm và tài nguyên bộ nhớ được chia sẻ (dùng chung), thì chúng ta cần có môi trường, công cụ lưu trữ chương trình, dữ liệu trong bộ nhớ thứ cấp. Thường thì người ta sử dụng mô hình bộ nhớ hai cấp (hình 4.2.1d). Lớp thứ nhất – bộ nhớ vật lý, tại đó cần nạp các chương trình được thực hiện và các dữ liệu cần truy cập. Lớp thứ hai – bộ nhớ ngoài có dung lượng lớn để lưu trữ các chương trình, dữ liệu lúc đó chưa cần nạp.

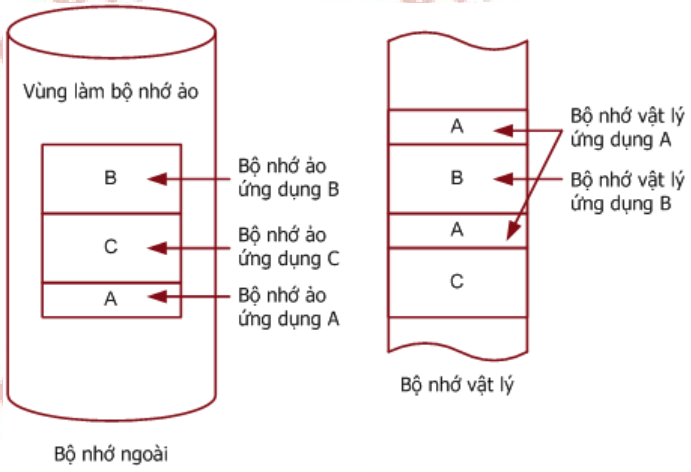
Để chạy chương trình, code và dữ liệu của nó cần được đưa vào bộ nhớ.

Bởi vì bộ nhớ thực hiện được chia sẻ giữa nhiều chương trình và mỗi chương trình (Process) có thể có không gian địa chỉ ảo lớn hơn không gian địa chỉ thực do đó vào thời gian thực hiện, chỉ có một phần code và dữ liệu của mỗi tiến trình nằm trong bộ nhớ thực.



Hình 4.2.1d

Trên hình 4.2.1e thể hiện cấu trúc bộ nhớ hai lớp, trong bộ nhớ thực lưu chỉ một phần bộ nhớ ảo của các chương trình.



Hình 4.2.1e

Blocking Mapping: ánh xạ theo khối

Cơ chế biến đổi địa chỉ động (DAT) phải sử dụng các bảng ánh xạ – chỉ rằng các ô nhớ ảo nào tại thời điểm hiện thời nằm trong bộ nhớ và ở ô nhớ thực nào. Nếu như ánh xạ được thực hiện theo từng byte hay word thì rõ ràng thông tin về ánh xạ còn lớn hơn bản thân bộ nhớ thực. Do đó để có thể áp dụng bộ nhớ ảo, cần có phương pháp cho phép giảm kích thước của bảng ánh xạ.

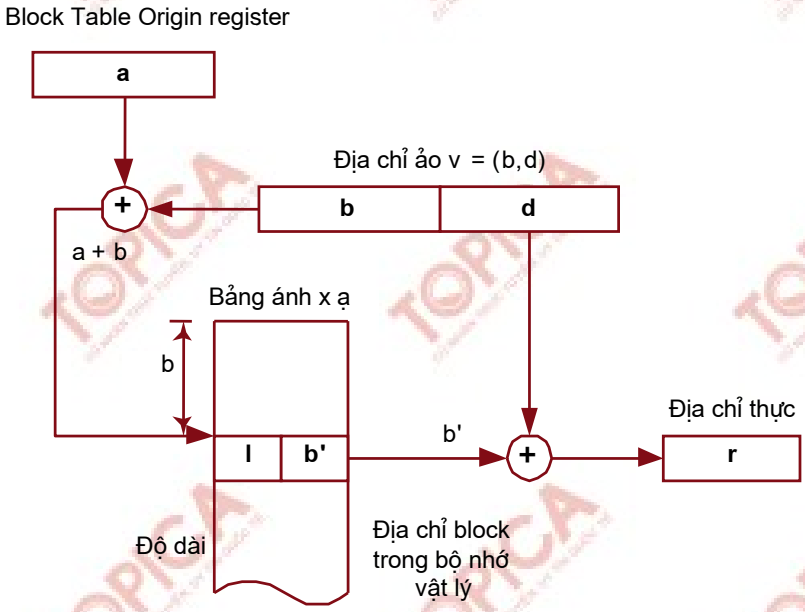
Vì ánh xạ từng ô nhớ là không thể chấp nhận, chúng ta phải gộp theo các khối – Block, và hệ thống theo dõi xem các khối của bộ nhớ ảo nằm đâu trong bộ nhớ. Kích thước khối càng lớn thì thông tin về ánh xạ càng nhỏ, nhưng các khối lớn lại cần nhiều thời gian nạp hơn và ngoài ra với xác suất lớn còn hạn chế số tiến trình có thể chia sẻ chung bộ nhớ vật lý.

Khi thiết kế cơ chế bộ nhớ ảo xuất hiện câu hỏi là các khối có kích thước như nhau hay khác nhau. Nếu các khối có kích thước như nhau thì chúng (khối) được gọi là các trang (Page), còn tổ chức bộ nhớ ảo tương ứng gọi là tổ chức theo trang. Nếu các khối có thể có kích thước khác nhau thì chúng được gọi là các đoạn (Segment), tổ chức bộ nhớ ảo là tổ chức theo đoạn. Trong một số hệ thống tổ hợp cả hai cách, tức là các phân đoạn được tạo thành từ các Page có kích thước như nhau.

Các địa chỉ trong các hệ thống ánh xạ theo khối là các địa chỉ gồm hai thành phần (hai chiều). Để truy cập đến một dữ liệu cụ thể, chương trình chỉ ra khối và độ lê của dữ liệu trong khối của dữ liệu trong khối đó:

Địa chỉ ảo V được chỉ ra với cặp số v = (b, d) trong đó b – số khối (số thứ tự) còn d – offset (độ lệch) tương đối so với đầu khối.

Sự biến đổi từ địa chỉ ảo v = (b, d) thành địa chỉ thực r được thực hiện như trên hình 4.2.1f. Mỗi tiến trình có bảng ánh xạ khối của mình (nằm trong bộ nhớ thực). Địa chỉ thực a của bảng này được nạp vào thanh ghi riêng của bộ xử lý gọi là thanh ghi địa chỉ (đầu) của bảng ánh xạ (Block Table Origin Register). Mỗi dòng của bảng ánh xạ chứa ánh xạ của một khối của tiến trình, ngoài ra các dòng được sắp xếp theo thứ tự số khối, từ khối 0, khối 1. Số khối b được cộng với địa chỉ cơ sở của bảng ánh xạ a, ta được địa chỉ thực của dòng chứa ánh xạ của khối b. Dòng này chứa địa chỉ thực b' của khối b. Sau đó cộng thêm vào địa chỉ b' này với offset d ta được địa chỉ thực cần tìm r = b' + d. Tất cả các phương pháp Block Mapping dùng trong các hệ thống tổ chức theo trang, phân đoạn hay tổ hợp được thể hiện trên hình 4.2.1f



Hình 4.2.1f

Cần phải để ý rằng Block Mapping được thực hiện động (Dynamic) ngay trong thời gian thực hiện tiến trình. Nếu cơ chế biến đổi địa chỉ DAT không đủ hiệu quả, thì chi phí của nó có thể làm giảm hiệu suất của hệ thống tới mức vượt quá ưu thế mà bộ nhớ ảo mang lại.

4.2.2. Phân trang

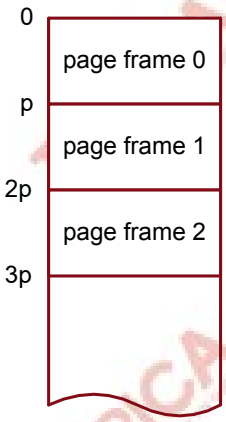
Chúng ta xem xét ánh xạ theo khối với kích thước cố định tức là tổ chức bộ nhớ theo trang. Trong phần này chúng ta chỉ xem xét tổ chức theo trang thuần tuý.

Địa chỉ ảo trong hệ thống theo trang được thể hiện bằng cặp v = (p, d) trong đó p – số thứ tự trang trong bộ nhớ ảo còn d – offset trong trang p:

p (Page No)	d (Offset)

Địa chỉ ảo v = (p, d)

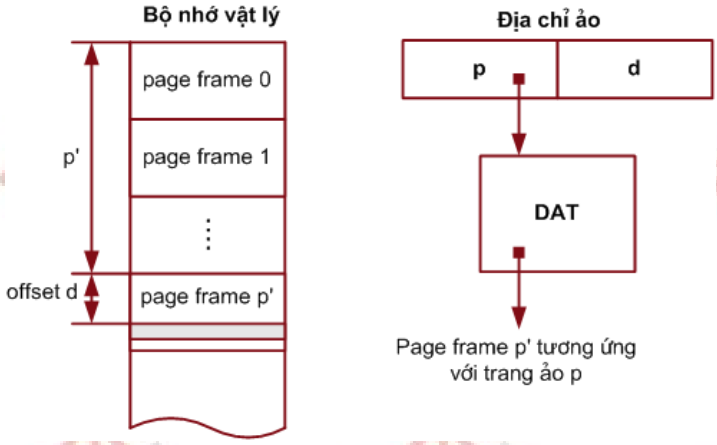
Tiến trình có thể được thực hiện tiếp nếu trang hiện thời cần đến của nó nằm trong bộ nhớ thực. Các trang được ghi từ bộ nhớ ngoài vào bộ nhớ thực, theo các khối gọi là Page frame và có kích thước đúng bằng kích thước trang. Các Page frame bắt đầu từ các địa chỉ (trong bộ nhớ thực) là bội số của kích thước trang (hình.4.2.2a). Các trang có thể được nạp vào bất kỳ Page frame trống nào.



Số trang	Kích thước	Vùng địa chỉ
0	p	0 → p-1
1	p	p → 2p-1
2	p	2p → 3p-1

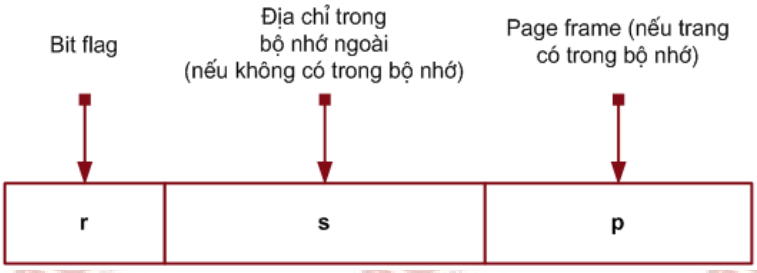
Hình 4.2.2a

Biến đổi địa chỉ động trong hệ thống tổ chức theo trang, thực hiện theo cách sau: Tiến trình truy cập đến địa chỉ ảo v = (p, d). Cơ chế ánh xạ trang thể hiện trên hình 4.2.2b, đầu tiên tìm số trang p trong bảng ánh xạ trang và xác định được Page frame p' tương ứng với trang p. Địa chỉ thực được tính bằng tổng của p' và d.



Hình 4.2.2b

Chúng ta sẽ xem xét quá trình này kỹ hơn. Trong thực tế, thường thì không phải tất cả các Page của tiến trình đều nằm trong bộ nhớ, do đó bảng ánh xạ phải chỉ ra là trang cần truy cập có nằm trong bộ nhớ không và nằm ở đâu, còn nếu không thì nó nằm đâu trong bộ nhớ ngoài. Trên hình 4.2.2c biểu diễn một dòng dữ liệu trong bảng ánh xạ trang. Bit flag r chỉ ra sự hiện diện của trang trong bộ nhớ: nếu r = 1 tức là trang nằm trong bộ nhớ thực, còn nếu r = 0 thì trang đó hiện không có trong bộ nhớ thực. Nếu như trang đó không có trong bộ nhớ thì s là địa chỉ (vị trí) của trang trong bộ nhớ ngoài, còn nếu nó nằm trong bộ nhớ thì p' là số Page frame. Chú ý rằng p' không phải là địa chỉ thực thật sự. Địa chỉ thực a – của Page frame p' được tính bằng a = (p) * (p') (số trang p có giá trị từ 0; 1; 2; 3...)



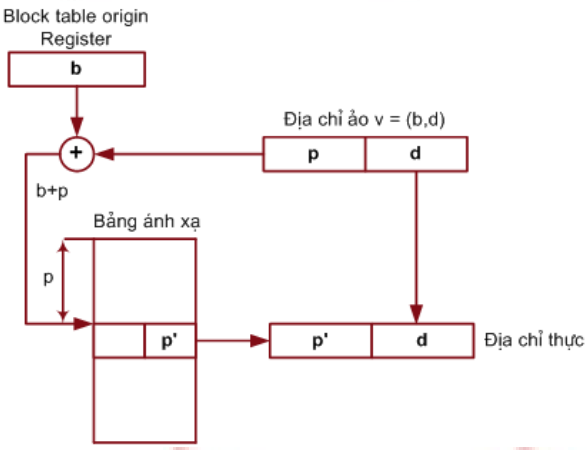
Hình 4.2.2c

Để tiết kiệm bộ nhớ (giảm kích thước của bảng ánh xạ) thì mỗi dòng có thể không cần chứa thông tin địa chỉ của trang trong bộ nhớ ngoài.

Biến đổi địa chỉ, ánh xạ trực tiếp

Chúng ta sẽ xem xét một số phương pháp biến đổi địa chỉ trang. Đầu tiên chúng ta sẽ xem xét biến đổi (ánh xạ) trực tiếp, được biểu diễn trên hình 4.2.2d.

Tiến trình đưa yêu cầu truy cập địa chỉ ảo v = (p, d). Trước khi tiến trình tiếp tục thực hiện, hệ điều hành nạp địa chỉ thực của bảng ánh xạ trang vào thanh ghi. Địa chỉ cơ sở b này được cộng với số trang p, thu được địa chỉ thực b + p của dòng chứa ánh xạ trang p. Dòng này chỉ ra Page frame p' của p. Sau đó Page frame được cộng với offset d và ta có địa chỉ thực r. Cách tính này gọi là phương pháp biến đổi trực tiếp (ánh xạ trực tiếp), bởi vì bảng ánh xạ trang chứa từng dòng cho mỗi trang bộ nhớ ảo của tiến trình. Nếu tiến trình có n trang bộ nhớ ảo thì bảng ánh xạ trong phương pháp ánh xạ trực tiếp chứa n dòng liên tiếp cho các trang từ 0 đến n –1

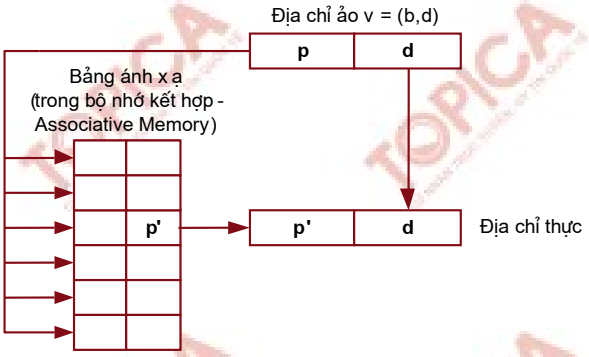


Hình 4.2.2d

Địa chỉ ảo cần tính và địa chỉ bảng ánh xạ đều chứa trong các thanh ghi của bộ xử lý do đó các thao tác với chúng có thể thực hiện rất nhanh trong vòng xử lý lệnh. Nhưng còn bảng ánh xạ, kích thước khá lớn, thường nằm trong bộ nhớ, do đó để truy cập nó cần một thao tác đọc bộ nhớ. Vì thời gian đọc dữ liệu từ bộ nhớ lớn hơn nhiều thời gian xử lý lệnh và chúng ta cần thêm một lần đọc bộ nhớ để lấy được dòng cần thiết của bảng ánh xạ, điều đó có nghĩa là phương pháp ánh xạ trực tiếp có thể làm giảm tốc độ của hệ thống khi thực hiện chương trình gần hai lần. Điều đó tất nhiên không chấp nhận được, do đó cần có các phương pháp biến đổi địa chỉ nhanh hơn. Nhưng điều đó cũng không có nghĩa là phương pháp này hoàn toàn không thể áp dụng, ví dụ như trong một số hệ thống đã sử dụng phương pháp này thành công bằng cách lưu toàn bộ bảng ánh xạ trong Cache Memory có tốc độ rất cao.

Biến đổi địa chỉ trang dùng bộ nhớ kết hợp Associative Memory

Một trong số các phương pháp cải thiện tốc độ của DAT là lưu toàn bộ bảng ánh xạ trang trong Associative Memory (bộ nhớ kết hợp) – với Associative Memory thời gian truy cập (Access Cycle) nhỏ hơn nhiều lần so với bộ nhớ thông thường. Trên hình 4.2.2e biểu diễn quá trình thực hiện biến đổi địa chỉ động chỉ dùng Associative Memory. Chương trình truy cập đến địa chỉ ảo v = (p, d). Tất cả các dòng (Record) của bảng (nằm trong Associative Memory) được so sánh đồng thời với p. Associative Memory trả lại giá trị p' – địa chỉ của Page frame của p. Sau đó cộng p' với offset d ta có địa chỉ thực r.



Hình 4.2.2e

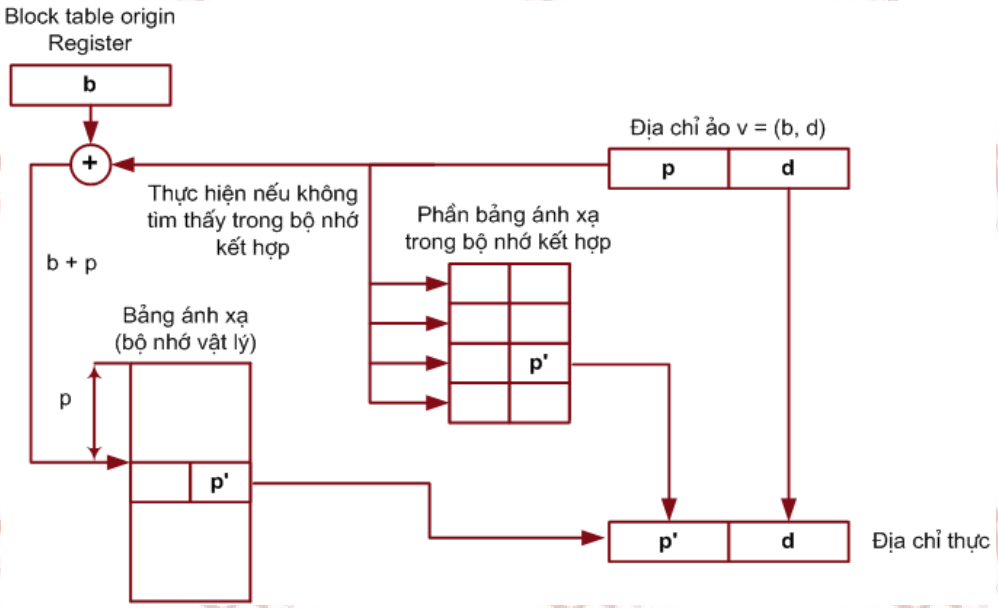
Chúng ta để ý rằng các mũi tên đi tới Associative Memory tới tất cả các bản ghi (dòng) của bảng. Điều đó có nghĩa là mỗi phần tử của Associative Memory được phân tích đồng thời với địa chỉ p. Cũng vì tính chất này mà bộ nhớ kết hợp rất đắt.

Ở đây chúng ta lại gặp vấn đề kinh tế: để có thể ứng dụng thành công bộ nhớ ảo cần có cơ chế biến đổi địa chỉ động (DAT) tốc độ cao. Nhưng dù sao sử dụng Cache Memory hay Associative Memory thì đều quá đắt. Chúng ta cần có một giải pháp trung gian nào đó vẫn đảm bảo tốc độ nhưng giá cả phải chăng.

Biến đổi địa chỉ trang sử dụng kết hợp Associative với ánh xạ trực tiếp

Đến lúc này chúng ta luôn định hướng đến các công cụ phần cứng để thực hiện bộ nhớ ảo. Nhưng chúng ta sẽ xem xét các thiết bị một cách logic (trừu tượng) hơn là vật lý. Chúng ta quan tâm không phải là một thiết bị cụ thể (vật lý) mà là tổ chức, chức năng và tốc độ của chúng.

Vì giá của Cache Memory và Associative Memory quá đắt so với RAM do đó chúng ta cần giải pháp trung gian để thực hiện ánh xạ trang. Trong giải pháp này chúng ta sử dụng Associative Memory để lưu giữ một phần của bảng ánh xạ (hình 4.2.2f). Trong bảng đó chỉ chứa các ánh xạ trang được truy cập gần nhất (về thời gian) – điều đó xuất phát từ kết quả tương đối là trang đã được truy cập trong thời gian gần nhất hoàn toàn có thể (với xác suất lớn) lại được truy cập tiếp. Trong các hệ thống hiện đại sử dụng phương pháp kết hợp này, chỉ số tốc độ đạt tới 90% hoặc hơn nữa so với khi hoàn toàn chỉ dùng Associative Memory.



Hình 4.2.2f

Sự biến đổi địa chỉ động được thực hiện theo cách sau: Chương trình cần truy cập theo địa chỉ ảo v = (p, d). Cơ chế biến đổi địa chỉ đầu tiên sẽ thử tìm ánh xạ của trang ảo p trong phần bảng ánh xạ nằm trong Associative Memory. Nếu như ánh xạ trang p có ở đó thì bảng ánh xạ trong Associative Memory trả lại kết quả số Page frame p' tương ứng với trang ảo p và giá trị p' này được cộng với offset d, ta có được địa chỉ thực r tương ứng với địa chỉ ảo v = (p, d).

Để đảm bảo chỉ số tốc độ cao, bảng ánh xạ trong Associative Memory phải không lớn quá. Thật vậy, trong thực tế, với các hệ thống sử dụng chỉ 8 hoặc 16 thanh ghi Associative Memory, có thể đạt được 90% hoặc hơn nữa tốc độ khi sử dụng hoàn toàn

Associative Memory (khi đó kích thước Associative Memory có thể lớn gấp hàng chục đến hàng trăm lần). Có được kết quả đó là nhờ tính chất đặc biệt của các tiến trình hoạt động, gọi là tính địa phương – Locality, hiện tượng này chúng ta sẽ xem xét sau.

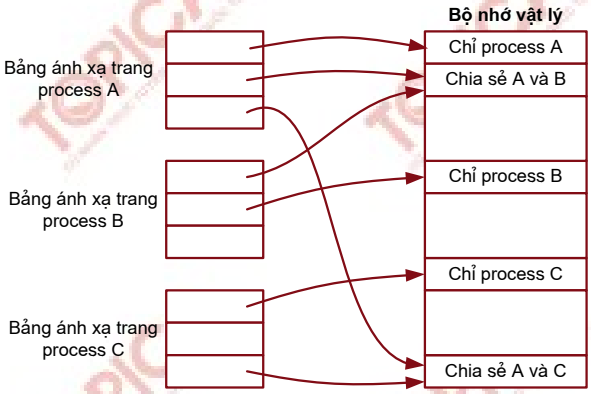
Sử dụng cơ chế kết hợp đó là lời giải kỹ thuật dựa trên các thông số kinh tế của các thiết bị hiện có.

Chia sẻ chương trình và dữ liệu trong hệ thống tổ chức trang

Trong các hệ đa nhiệm, đặc biệt các hệ phân chia thời gian, thường xảy ra trường hợp nhiều user cùng chạy các chương trình giống nhau. Nếu mỗi user đều dùng riêng một bản copy của chương trình thì một phần đáng kể bộ nhớ bị lãng phí. Rõ ràng có một giải pháp – đó là sử dụng chung (chia sẻ) các trang có thể chia sẻ.

Sử dụng chung cần phải xử lý, theo dõi cẩn thận để ngăn chặn tình huống khi một tiến trình cố thay đổi dữ liệu mà cùng lúc đó tiến trình khác đang đọc. Trong nhiều hệ thống tiên tiến, có sử dụng chung bộ nhớ, thường các chương trình cấu tạo từ hai phần riêng biệt – vùng chương trình (Procedure) và vùng dữ liệu (Data). Các procedure cố định (không thay đổi) hay còn gọi là Reentrant Procedure, có thể dùng chung. Các dữ liệu tĩnh (ví dụ bảng thông tin cố định nào đó) – có thể dùng chung. Các procedure động (thay đổi) không thể dùng chung.

Tất cả điều đó nói lên rằng mỗi trang bộ nhớ cần xác định xem có thể dùng chung hoặc không. Sau khi các trang của mỗi tiến trình được chia ra hai loại, trong hệ thống tổ chức theo trang thuần tuý việc chia sẻ bộ nhớ được biểu diễn trên hình 4.2.2g. Việc chia sẻ bộ nhớ làm giảm dung lượng bộ nhớ sử dụng và hệ thống có thể phục vụ số lượng user lớn hơn.

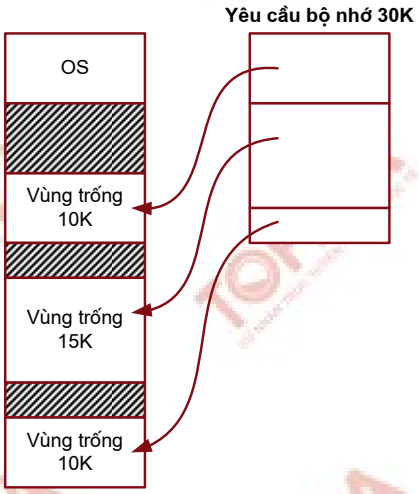


Hình 4.2.2g

4.2.3. Phân đoạn

Trong chương trước nói về bộ nhớ thực, chúng ta đã thấy rằng trong các hệ thống đa nhiệm với phân đoạn thay đổi, việc phân bố bộ nhớ thường thực hiện theo các thuật toán 'First Suitable', 'Most Suitable' hay 'Least Suitable' theo tương quan về kích thước các vùng trống. Nhưng vẫn có hạn chế là chương trình phải nạp vào một vùng liên tục.

Trong các hệ thống tổ chức theo phân đoạn, sự hạn chế đó có thể vượt qua, và cho phép chương trình (và dữ liệu) có thể nằm trong nhiều khối nhớ không liên tục (hình 4.2.3a); Tuy rằng bản thân các khối nhớ là một vùng nhớ liên tục nhưng chúng có thể có kích thước khác nhau.

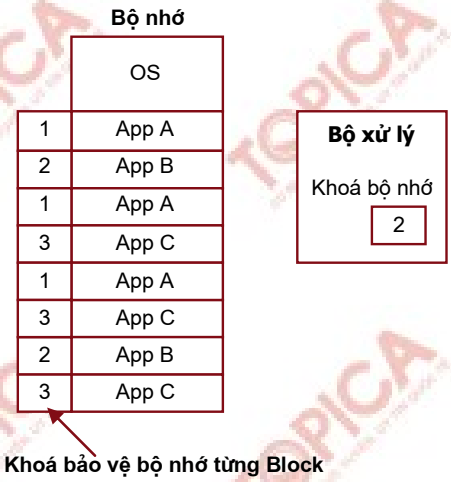


Hình 4.2.3a

Trong giải pháp này xuất hiện một số vấn đề thú vị. Ví dụ như vấn đề bảo vệ chương trình, bộ nhớ. Sử dụng cặp thanh ghi biên không còn tác dụng nữa. Vấn đề tương tự như thế với việc kiểm soát việc truy cập bộ nhớ của một chương trình nào đó.

Một trong những phương pháp thực hiện bảo vệ bộ nhớ trong các hệ thống tổ chức theo phân đoạn đó là sử dụng khoá bảo vệ, như trên hình 4.2.3b

Hình vẽ: bảo vệ bộ nhớ dùng khoá trong các hệ đa nhiệm với phân bố bộ nhớ thành các khối. Khi giá trị khoá trong bộ xử lý được đặt bằng hai – tương ứng với user B, chương trình của user B chỉ có thể truy cập đến các khối bộ nhớ có cùng khoá (2) Việc điều khiển khoá bảo vệ được thực hiện bởi hệ điều hành.

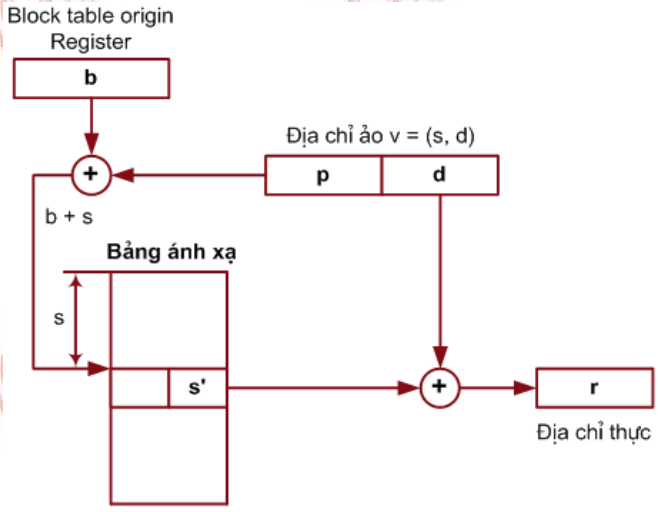


Hình 4.2.3b

Địa chỉ ảo trong các hệ thống phân đoạn – là cặp số v = (s,d) trong đó s – số phân đoạn của bộ nhớ ảo, còn d – offset trong đoạn đó.

Tiến trình có thể tiếp tục chỉ trong trường hợp phân đoạn hiện tại nằm trong bộ nhớ. Phân đoạn được nạp từ bộ nhớ ngoài vào bộ nhớ thực theo cả phân đoạn và phân đoạn phải được nạp vào một vùng nhớ liên tục. Phân đoạn có thể được nạp vào bất kỳ vùng trống liên tục nào đủ lớn. Các chiến lược phân bố phân đoạn cũng giống như trong các hệ thống đa nhiệm với phân đoạn thay đổi, thường thì các thuật toán 'First Suitable' và 'Most Suitable' hay được áp dụng.

Quá trình biến đổi địa chỉ động được thực hiện như hình 4.2.3c. Tiến trình truy cập theo địa chỉ ảo v = (s, d). Cơ chế ánh xạ phân đoạn tìm và (nếu có) thì xác định địa chỉ đầu phân đoạn s'. Sau đó địa chỉ thực r được tính bằng tổng s' và d. Các chi tiết của quá trình biến đổi địa chỉ chúng ta sẽ xem xét sau.



Hình 4.3.2c

Điều khiển truy cập trong hệ thống tổ chức phân đoạn

Cho phép mỗi tiến trình truy cập không hạn chế đến phân đoạn bất kỳ là không thực tế. một trong những ưu thế của hệ thống tổ chức theo phân đoạn là khả năng kiểm soát truy cập chặt chẽ. Mỗi tiến trình được trao một số quyền truy cập xác định đối với một số phân đoạn nào đó còn phần lớn phân đoạn khác là hoàn toàn không được truy cập.

Trong bảng có liệt kê những dạng quyền truy cập phổ thông nhất được áp dụng trong các hệ thống tiên tiến. Nếu tiến trình có quyền read thì nó có thể đọc (truy cập) đến bất kỳ ô nhớ nào trong phân đoạn và khi cần nó có thể copy toàn bộ cả phân đoạn.

Nếu như tiến trình có quyền write thì nó có thể thay đổi nội dung của bất kỳ ô nhớ nào trong phân đoạn, và có thể thêm thông tin bổ sung vào phân đoạn. Khi cần nó cũng có thể xoá toàn bộ phân đoạn. Tiến trình có quyền execute thì nó có thể làm việc với phân đoạn như là với chương trình. Còn quyền truy cập đến các phân đoạn dữ liệu thường là bị cấm.

Khi tiến trình có quyền append thì nó có thể thêm thông tin vào cuối phân đoạn, nhưng không được phép thay đổi thông tin đã có.

Bảng 1

Quyền	Ký hiệu	Giải thích
Read	R	Có quyền đọc thông tin.
Write	W	Có quyền thay đổi nội dung.
Execute	E	Có quyền thực hiện (chương trình).
Append	A	Có quyền thêm thông tin vào cuối khối.

Trong bảng hệ thống có phân biệt 4 loại quyền truy cập như trên, có thể có 16 chế độ điều khiển truy cập – cho phép hay cấm mỗi loại. Tuy nhiên, một số trong chúng là không có ý nghĩa, còn một số khác thì có. Để đơn giản chúng ta xem xét 8 tổ hợp quyền truy cập đối với các quyền read, write và execute – như trong bảng 2:

Bảng 2

	E	W	R	Giải thích
0	0	0	0	Cấm tất cả các quyền
4	0	0	1	
3	0	1	0	
6	0	1	1	
1	1	0	0	
5	1	0	1	
2	1	1	0	
7	1	1	1	Có toàn quyền

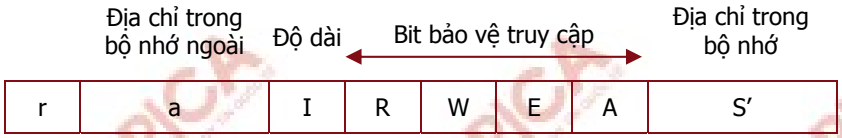
- Chế độ 0 tất cả các hình thức truy cập đều bị cấm. Chế độ này ngăn chặn sự truy cập bất hợp pháp, tiến trình sẽ không thể truy cập đến phân đoạn.
- Chế độ 1 chỉ cho phép thực hiện. Chế độ này cần thiết khi tiến trình chỉ cần được phép sử dụng chương trình trong phân đoạn nhưng không được thay đổi nó hay copy.
- Chế độ 2 và 3 thực tế không áp dụng – không có logic khi cho phép tiến trình thay đổi nội dung của phân đoạn mà không cho phép đọc.
- Chế độ 4 chỉ cho phép đọc. Chế độ này cần cho các quá trình tìm kiếm thông tin, nhưng không cho phép thay đổi chúng.
- Chế độ 5 cho phép đọc và thực hiện. Chế độ này áp dụng trong các trường hợp tiến trình được phép thực hiện chương trình trong phân đoạn nhưng không được thay đổi nó. Tuy nhiên tiến trình có thể copy phân đoạn và sau đó có thể thay đổi bản copy của mình.

- Chế độ 6 cho phép đọc và ghi. Chế độ này thường áp dụng với phân đoạn chứa dữ liệu mà tiến trình có thể đọc hoặc ghi và dữ liệu cần tránh trường hợp vô tình cố thực hiện (vì phân đoạn đó không chứa chương trình)
- Chế độ 7 không hạn chế quyền truy cập. Chế độ này cần để cho phép tiến trình có toàn quyền làm việc với các phân đoạn của mình, hoặc khi user khác tin tưởng hoàn toàn – cho phép toàn quyền truy cập phân đoạn của anh ta.
- Các chế độ kiểm soát quyền truy cập là cơ sở của việc bảo vệ phân đoạn và được áp dụng trong nhiều hệ thống.

Biến đổi địa chỉ phân đoạn – ánh xạ trực tiếp

Cũng như trong tổ chức theo trang, trong các hệ tổ chức phân đoạn có nhiều chiến lược thực hiện biến đổi địa chỉ của phân đoạn. Việc biến đổi địa chỉ có thể là ánh xạ trực tiếp, dùng Associative Memory hay kết hợp. Chúng ta sẽ xem xét biến đổi trực tiếp và toàn bộ bảng ánh xạ nằm trong Cache Memory.

Dạng của bản ghi trong bảng ánh xạ thường có như hình 4.2.3d.



Hình 4.2.3d

Đầu tiên chúng ta xét trường hợp khi quá trình biến đổi diễn ra bình thường và sau đó là một số trường hợp có thể khác.

Tiến trình truy cập theo địa chỉ ảo v = (s, d) trong đó s là số phân đoạn. Số phân đoạn s được cộng với địa chỉ cơ sở b – địa chỉ đầu của bảng ánh xạ – nằm trong thanh ghi. Ta có được địa chỉ thực của bảng ghi ánh xạ của phân đoạn s. Từ bảng ánh xạ ta có địa chỉ thực s', cộng s' với Offset d ta được địa chỉ thực r = s' + d là địa chỉ thực tương ứng với địa chỉ ảo v = (s, d).

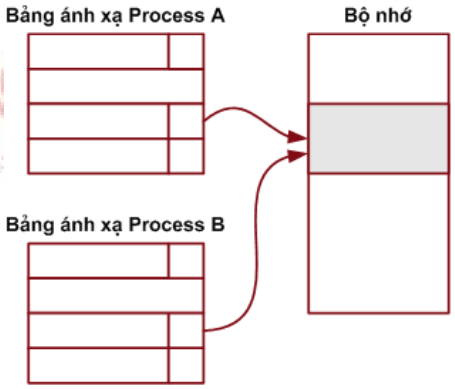
Trên hình 4.2.3d chúng ta thấy dạng của một ánh xạ.

Bit dấu hiệu tồn tại r cho biết thời điểm đó phân đoạn có nằm trong bộ nhớ hay không. Nếu như có, thì s' là địa chỉ đầu của nó, còn nếu không có trong bộ nhớ thì a là địa chỉ bộ nhớ ngoài theo đó hệ điều hành nạp phân đoạn để tiến trình có thể tiếp tục. Tất cả các thao tác truy cập phân đoạn được kiểm soát theo độ dài phân đoạn *l* để luôn chắc chắn rằng chúng không vượt ra ngoài phân đoạn. Mỗi thao tác truy cập phân đoạn còn được kiểm soát theo các bit bảo vệ để xác định xem thao tác đo có được phép hay không. Như thế, khi thực hiện biến đổi địa chỉ động, sau khi tìm đến dòng ánh xạ của một phân đoạn cụ thể s đầu tiên sẽ kiểm tra bit r để xác định xem lúc đó phân đoạn có nằm trong bộ nhớ hay không. Nếu không nằm trong bộ nhớ thì sinh ra một ngắt Missing Segment Fault, theo đó hệ điều hành nhận quyền điều khiển và nạp phân đoạn đó từ bộ nhớ ngoài (theo địa chỉ a). Sau khi nạp xong quá trình xử lý địa chỉ tiếp tục, offset d sẽ được so sánh với độ dài *l* của phân đoạn. Nếu d > *l* thì sinh ra ngắt Segment Over Flow Fault, và hệ điều hành không cho phép thực hiện. Nếu d nằm trong giới hạn thì tiếp tục kiểm tra theo các bit bảo vệ để xem tác vụ truy cập có được phép hay không. Nếu được phép thì mới cộng địa chỉ thực s' của phân đoạn với offset d và có được địa chỉ thực r = s' + d tương ứng với địa chỉ ảo v = (s, d). Còn nếu thao tác không được phép thì sẽ sinh ra ngắt segment protection fault và hệ điều hành không cho thực hiện.

Chia sẻ chương trình và dữ liệu trong các hệ thống tổ chức theo phân đoạn

Một trong những ưu điểm cơ bản của tổ chức theo phân đoạn so với tổ chức theo trang là tổ chức theo phân đoạn mang tính logic hơn là tính vật lý. Tổng quát thì các phân đoạn không bị giới hạn bởi kích thước cố định nào. Chúng có thể có kích thước tuỳ trường hợp cần thiết khác nhau. Ví dụ phân đoạn để lưu một bảng nào đó sẽ có kích thước tương ứng với bảng đó. Kích thước của phân đoạn chứa cấu trúc dữ liệu thay đổi, có thể thay đổi theo dữ liệu....

Chia sẻ chương trình, dữ liệu trong hệ thống tổ chức theo phân đoạn cũng đơn giản hơn hệ thống tổ chức trang. Ví dụ trong hệ thống tổ chức trang, có một bảng dữ liệu dùng chung nào đó có kích thước 3,5 trang, thì thay vì một con trỏ nói rằng 'bảng này dùng chung' chúng ta cần các con trỏ cho mỗi trang. Mặt khác làm việc với một phần của trang khó hơn nhiều. Tình trạng còn phức tạp hơn trong trường hợp cấu trúc dữ liệu động. Nếu dữ liệu mở rộng thêm một trang thì dấu hiệu sử dụng chung trang cần thay đổi ngay trong thời gian hoạt động của tiến trình.



Hình 4.2.3e: ***Chia sẻ bộ nhớ trong tổ chức theo segment***

Còn trong hệ thống tổ chức phân đoạn, sau khi phân đoạn đã được đánh dấu dùng chung, thì dữ liệu có thể thay đổi kích thước tuỳ ý mà không làm thay đổi dấu hiệu logic là phân đoạn đó được dùng chung.

Trên hình 4.2.3e biểu diễn sự chia sẻ chương trình, dữ liệu trong hệ thống tổ chức phân đoạn, hai tiến trình dùng chung phân đoạn nếu trong các bảng ánh xạ phân đoạn có các bản ghi trỏ tới cùng một phân đoạn trong bộ nhớ.

4.2.4. Kết hợp trang – phân đoạn

Hệ thống tổ chức trang hay phân đoạn đều có những ưu điểm của mình trong hệ thống với bộ nhớ ảo. Từ giữa những năm 1960, cụ thể hơn là bắt đầu từ hệ thống Multics của trường MIT, hệ TSS của IBM và nhiều hệ thống khác người ta đã áp dụng kết hợp tổ chức theo trang – phân đoạn. Nhờ đó có được các ưu điểm của cả hai loại mô hình tổ chức bộ nhớ ảo. Các phân đoạn thường chứa nhiều trang, ngoài ra không nhất thiết là tất cả các trang của một phân đoạn nằm trong bộ nhớ cùng lúc và các trang liền nhau trong bộ nhớ ảo chưa chắc đã ánh xạ lên các trang liền nhau trong bộ nhớ thực.

Các hệ thống tổ chức trang – phân đoạn sử dụng đánh địa chỉ gồm ba thành phần, tức là địa chỉ ảo v gồm 3 số v = (s, p, d) trong đó s – số phân đoạn, p – số trang trong phân đoạn và d – offset trong trang:

Số segment **s**	Số trang **p**	Offset **d**

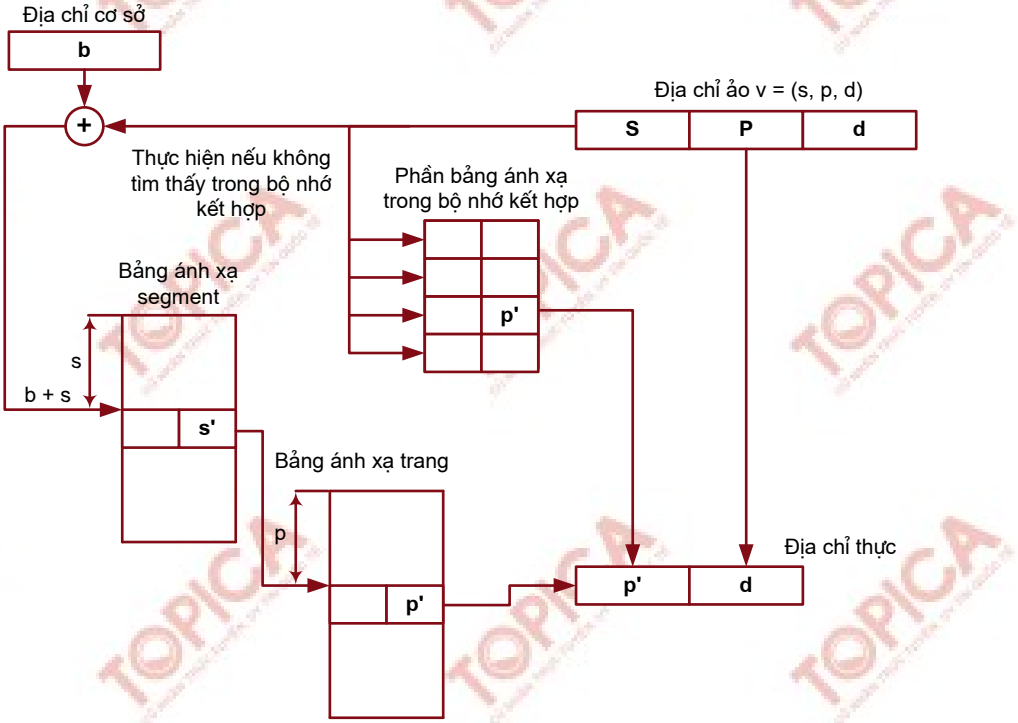
Biến đổi địa chỉ động trong hệ thống kết hợp trang – phân đoạn

Bây giờ chúng ta xem xét sự biến đổi địa chỉ động trong các hệ thống tổ chức trang–phân đoạn sử dụng kết hợp ánh xạ trực tiếp và Associative Memory (hình 4.2.4a)

Tiến trình truy cập theo địa chỉ ảo v = (s, p, d). Phần bảng ánh xạ trong Associative Memory chứa ánh xạ các trang truy cập gần nhất. Đầu tiên hệ thống tìm kiếm bản ghi với tham số (s, p) trong Associative Memory. Nếu như kết quả là dương thì ta có địa

chỉ Page frame p' theo đó trang p nằm trong bộ nhớ thực. Cộng thêm offset d ta có địa chỉ thực r tương ứng với địa chỉ ảo v.

Thông thường thì phần lớn số yêu cầu biến đổi địa chỉ đều đạt được khi tìm trong Associative Memory. Nếu như không có địa chỉ cần tìm trong bộ nhớ kết hợp thì dùng phương pháp ánh xạ trực tiếp như sau: địa chỉ cơ sở b của bảng ánh xạ phân đoạn được cộng với số phân đoạn s, ta có được địa chỉ b + s của bản ghi ánh xạ cho phân đoạn s trong bảng ánh xạ (trong RAM). Trong bản ghi này có chứa địa chỉ cơ sở s' của bảng ánh xạ trang của phân đoạn s. Sau đó cộng s' với số trang p, tạo thành địa chỉ p + s' của bản ghi ánh xạ trang đối với trang p của phân đoạn s. Bảng này cho phép xác định trang ảo p tương ứng với số Page frame p'. Cuối cùng, cộng p' với offset d ta có địa chỉ thực r tương ứng với địa chỉ ảo v = (s, p, d).



Hình 4.2.4a

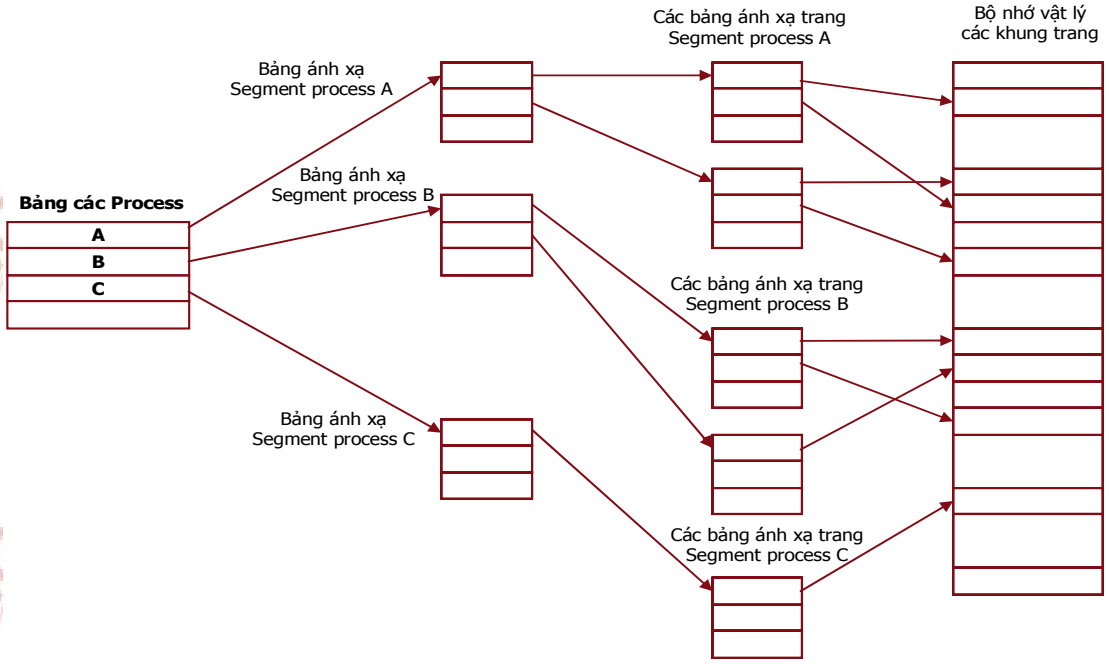
Thuật toán trên thực hiện được tất nhiên là với điều kiện rằng tất cả các thông tin cần thiết phải nằm đúng chỗ của chúng. Nhưng tất nhiên có nhiều lúc không phải như vậy. Việc tìm kiếm trong bảng ánh xạ phân đoạn có thể chỉ ra rằng phân đoạn s không có trong bộ nhớ, lúc đó sinh ra ngắt Missing Segment Fault; hệ điều hành tìm phân đoạn trong bộ nhớ ngoài, tạo cho nó bảng ánh xạ trang và nạp trang tương ứng vào bộ nhớ vào chỗ có thể là của một trang của tiến trình khác hay của tiến trình đó. Ngay cả khi phân đoạn nằm trong bộ nhớ, việc tìm kiếm trong bảng ánh xạ trang có thể chỉ ra rằng trang cần thiết không có trong bộ nhớ, lúc đó sinh ra ngắt Missing Page fault, hệ điều hành tìm trang cần thiết trong bộ nhớ ngoài và nạp nó vào bộ nhớ (hoàn toàn có thể vào chỗ của một trang khác – trang thứ hai sẽ bị đưa ra khỏi bộ nhớ)

Cũng như trong trường hợp tổ chức phân đoạn thuần tuý, địa chỉ ảo có thể vượt ra ngoài giới hạn phân đoạn, khi đó sinh ra ngắt segment overflow fault. Hoặc khi kiểm tra các bit bảo vệ thì thấy rằng thao tác truy cập đó bị cấm, lúc đó sinh ra ngắt segment protection fault và hệ điều hành phải xử lý tất cả các tình huống này.

Bộ nhớ kết hợp (Associative Memory) hoặc hình thức bộ nhớ tốc độ cao khác (ví dụ Cache) đóng vai trò rất quan trọng trong việc đảm bảo hiệu quả (về tốc độ) của cơ chế biến đổi địa chỉ động (DAT). Nếu như chỉ sử dụng ánh xạ trực tiếp tức là chỉ có bảng ánh xạ nằm trong bộ nhớ thì để truy cập được đến ô nhớ cần thiết, ta phải thực hiện 3 lần thao tác đọc bộ nhớ:

- Lần 1 để truy cập đến bảng ánh xạ phân đoạn.
- Lần 2 để đọc ánh xạ trong bảng ánh xạ trang.
- Lần 3 mới là truy cập ô nhớ mong muốn.

Như vậy để truy cập bộ nhớ cần 3 thao tác đọc bộ nhớ tức là tốc độ giảm gần 3 lần trong khi chỉ cần 8–16 thanh ghi bộ nhớ kết hợp để chứa các ánh xạ mới nhất là ta đã có thể đạt 90% hoặc hơn về mặt tốc độ.



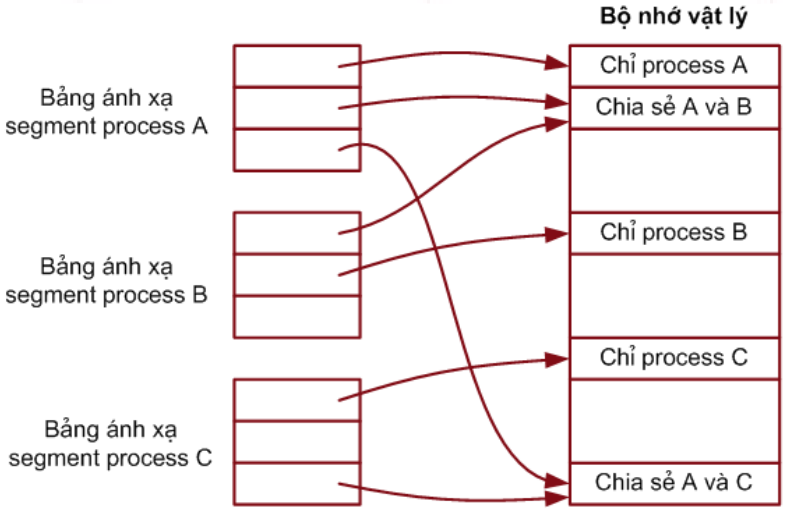
Hình 4.2.4b

Trên hình vẽ biểu diễn cấu trúc các bảng cần trong hệ thống kết hợp Page – segment. Ở mức trên cùng là bảng các tiến trình, nó chứa các bản ghi thông tin địa chỉ bảng ánh xạ phân đoạn của tiến trình tương ứng, mỗi tiến trình (đang được hệ thống quản lý) có một bản ghi trong bảng tiến trình. Các bảng ánh xạ phân đoạn của các tiến trình chứa các bản ghi có số phân đoạn và địa chỉ bảng ánh xạ trang của phân đoạn đó, còn mỗi bản ghi của bảng ánh xạ trang chỉ tới Page frame của bộ nhớ thực – tại đó có trang bộ nhớ ảo tương ứng, hoặc địa chỉ bộ nhớ ngoài nơi có thể tìm đến trang đó.

Trong hệ thống có nhiều tiến trình, phân đoạn và trang, thì cấu trúc thông tin ánh xạ bộ nhớ có thể chiếm phần đáng kể bộ nhớ. Chúng ta lưu ý rằng quá trình biến đổi địa chỉ động thực hiện ngay trong quá trình hoạt động của tiến trình, do đó các cấu trúc thông tin ánh xạ đều nằm trong bộ nhớ. Mặt khác, các cấu trúc thông tin ánh xạ càng lớn thì bộ nhớ còn lại càng nhỏ và do đó số tiến trình hệ thống có thể phục vụ giảm, điều đó có thể dẫn tới giảm hiệu quả của hệ thống (giảm băng thông). Vì vậy người ta thiết kế hệ điều hành cần phải chú ý đến rất nhiều nhân tố ảnh hưởng trái ngược nhau, phân tích để có được giải pháp cân bằng, có thể đảm bảo hiệu quả của hệ thống.

Chia sẻ chương trình, dữ liệu trong hệ thống tổ chức trang – phân đoạn

Trong các hệ thống tổ chức trang–phân đoạn, các ưu điểm của việc chia sẻ chương trình và dữ liệu dễ hơn. Việc chia sẻ được thực hiện bằng cách trong các bảng ánh xạ phân đoạn của các tiến trình khác có các bản ghi chứa các con trỏ tới cùng một bảng ánh xạ trang (hình 4.2.4c)



Hình 4.2.4c

4.3. Thay thế trang bộ nhớ

Mở đầu

Trong mục trước chúng ta đã xem xét các hình thức tổ chức bộ nhớ ảo: tổ chức trang, tổ chức phân đoạn, và kết hợp trang – phân đoạn: các cơ chế phần cứng cũng như chương trình, thực hiện bộ nhớ ảo. Mục này chúng ta sẽ xem xét các chiến lược điều khiển bộ nhớ ảo.

Các chiến lược điều khiển bộ nhớ ảo

Trong mục trước về bộ nhớ thực, chúng ta đã xem xét các chiến lược lựa chọn, phân bố và loại bỏ thông tin (trang). Chương này chúng ta sẽ xem xét các chiến lược đó được ứng dụng trong bộ nhớ ảo thế nào:

1) Chiến lược lựa chọn: chúng dùng để xác định thời điểm nạp trang hay phân đoạn từ bộ nhớ ngoài vào bộ nhớ. Chúng ta đã nói có hai chiến lược: Chiến lược lựa chọn theo yêu cầu – hệ thống yêu cầu (truy cập) trang hay phân đoạn của tiến trình, chỉ sau khi đã xuất hiện yêu cầu thì trang hay phân đoạn mới được nạp vào bộ nhớ; Chiến lược dự đoán trước – hệ thống sẽ cố gắng xác định xem trang/phân đoạn nào sẽ có yêu cầu truy cập và nếu như xác suất là lớn và có chỗ trống thì trang/phân đoạn đó sẽ được nạp trước vào bộ nhớ.

2) Chiến lược phân bố – mục đích của chúng là xác định trang/phân đoạn sẽ được nạp vào nơi nào của bộ nhớ. Trong các hệ thống tổ chức trang thì sự phân bố tương đối rõ ràng và tự nhiên, trang có thể được nạp vào bất cứ Page frame nào trống. Còn trong các hệ thống tổ chức phân đoạn thì cần các chiến lược tương tự như chúng ta đã phân tích khi xem xét cách hệ thống đa nhiệm với phân đoạn thay đổi.

3) Chiến lược loại bỏ – xác định xem trang/phân đoạn nào sẽ bị loại ra khỏi bộ nhớ để lấy chỗ trống nạp trang/phân đoạn cần thiết nếu như bộ nhớ đã hết chỗ trống.

4.3.1. Các chiến lược thay thế trang

Trong các hệ thống tổ chức trang, thường tất cả các Page frame đều bận. Trong trường hợp đó, chương trình điều khiển bộ nhớ (nằm trong hệ điều hành) phải xác định trang nào sẽ phải bị loại ra khỏi bộ nhớ để có chỗ nạp trang cần thiết. Có nhiều chiến lược, thuật toán loại bỏ trang, chúng ta sẽ xem xét các chiến lược sau:

- Nguyên tắc tối ưu (Principle Of Optimality).
- Loại bỏ ngẫu nhiên (Random Page Replacement).
- Loại bỏ theo nguyên tắc FIFO (First In First Out).
- Loại bỏ theo nguyên tắc LRU (Least Recently Used).
- Loại bỏ theo nguyên tắc LFU (Least Frequency Used).
- Loại bỏ theo nguyên tắc NUR (Not Used Recently).
- Nguyên tắc Working Set of Pages.

Nguyên tắc tối ưu (Principle Of Optimality)

Theo nguyên tắc này, để đảm bảo các đặc tính tốc độ tốt nhất và sử dụng hiệu quả nhất tài nguyên thì phải loại bỏ trang mà trong tương lai sẽ không có yêu cầu truy cập đến nó (trong khoảng thời gian lâu nhất). Tất nhiên có thể thấy rằng chiến lược này thật sự đưa lại kết quả tối ưu, nhưng thực hiện nó thì có thể nói là không thể bởi vì đơn giản là chúng ta không thể biết trước tương lai.

Vì thế chúng ta cố gắng tìm các chiến lược dễ dàng hơn và đạt kết quả chấp nhận được, gần với kết quả tối ưu.

Loại bỏ theo chiến lược ngẫu nhiên

Nếu như chúng ta cần chiến lược có thời gian trễ nhỏ và không quá thiên về một phía (trong mối tương quan với 1 user cụ thể nào đó) thì có thể theo cách đơn giản – chọn trang bất kỳ ngẫu nhiên. Trong trường hợp này tất cả các trang đang nằm trong bộ nhớ đều có thể bị chọn và với xác suất như nhau, và tất nhiên là hoàn toàn có thể trang thường xuyên phải truy cập sẽ bị loại ra khỏi bộ nhớ. Vì tính ngẫu nhiên nên chiến lược này ít được sử dụng.

Chiến lược FIFO

Trong chiến lược này chúng ta gán cho mỗi trang vào lúc nó được nạp vào bộ nhớ một chỉ số. Khi cần phải loại bỏ trang nào đó khỏi bộ nhớ thì sẽ chọn trang nằm trong bộ nhớ lâu nhất (theo chỉ số đã gán). Một cách tự nhiên, ưu thế của chiến lược này có vẻ như rõ ràng: trang đó đã có thể 'sử dụng cơ hội của mình (đã được truy cập) và đã đến lúc nhường cho người khác. Tiếc rằng không phải luôn như vậy và theo chiến lược FIFO, với xác suất lớn, trang thường được sử dụng (truy cập) sẽ bị loại ra, bởi vì sự kiện rằng 1 trang nằm trong bộ nhớ trong khoảng thời gian dài hoàn toàn có thể có nghĩa rằng nó luôn được sử dụng (truy cập).

FIFO anomaly (dị thường FIFO)

Thoạt tiên có vẻ như rõ ràng rằng khi tăng số lượng Page Frame cho một tiến trình, tiến trình đó sẽ hoạt động với số lần ngắt do Missing Page Fault sẽ giảm. Nhưng thống kê theo Belady, Nelson và Shedler thì chiến lược FIFO với các thứ tự truy xuất trang xác định, khi tăng số lượng Page frame dẫn tới tăng số lần ngắt do missing Page Fault. Hiện tượng này gọi là FIFO Anomaly.

Bảng 1

Trang				Xảy ra ngắt
A	A			*
B	B	A		*
C	C	B	A	*
D	D	C	B	*
A	A	D	C	*
B	B	A	D	*
E	E	B	A	*
A	E	B	A	
B	E	B	A	
C	C	E	B	*
D	D	C	E	*
E	D	C	E	

Bảng 2

Trang					Xảy ra ngắt
A	A				*
B	B	A			*
C	C	B	A		*
D	D	C	B	A	*
A	D	C	B	A	
B	D	C	B	A	
E	E	D	C	B	*
A	A	E	D	C	*
B	B	A	E	D	*
C	C	B	A	E	*
D	D	C	B	A	*
E	E	D	C	B	*

Bảng 1 biểu diễn quá trình phân bố trang theo chiến lược FIFO trong trường hợp tiến trình có 3 Page frame. Bảng 2 biểu diễn cùng quá trình theo chiến lược FIFO trong trường hợp có 4 Page frame. Thực tế chúng ta thấy rằng khi số Page frame tăng thì số lần ngắt missing Page fault thực sự cũng tăng.

Chiến lược LRU

Theo chiến lược này, để loại bỏ trang chúng ta sẽ chọn trang nào không được sử dụng (truy xuất) lâu nhất. Ở đây chúng ta xuất phát từ một quy luật rằng quá khứ gần – là định hướng tốt để dự đoán tương lai. Chiến lược này đòi hỏi mỗi lần truy xuất trang, chỉ số của nó (thời gian truy xuất) được làm mới (Update). Điều đó có thể kéo theo thời gian trễ đáng kể, và cũng vì thế thuật toán LRU thuần tuý ít được áp dụng trong các hệ thống hiện đại mà người ta thường sử dụng các biến thể gần với LRU đảm bảo thời gian trễ nhỏ hơn.

Khi áp dụng các qui luật có tính trực giác cần phải phân tích kỹ. Ví dụ, theo chiến lược LRU có thể xảy ra trường hợp trang không sử dụng lâu nhất có thể là trang sẽ được truy cập ngay sau đó, chẳng hạn do đến lúc đó chương trình hoàn thành vòng lặp

mà thân của nó bao một vài trang. Tức là khi loại trang không sử dụng lâu nhất chúng ta có thể lại phải nạp nó vào ngay sau đó.

Loại bỏ theo chiến lược LFU

Một trong những chiến lược gần với thuật toán LRU là chiến lược LFU, theo đó trang ít được sử dụng nhất (theo tần số) sẽ bị loại. Ở đây chúng ta kiểm soát tần số truy cập (sử dụng) mỗi trang, và trong trường hợp cần, sẽ loại trang nào có tần số thấp nhất. Chiến lược này có vẻ đúng (theo trực giác) nhưng có xác suất lớn là trang bị loại được lựa chọn không thực. Ví dụ như trang có tần số sử dụng thấp nhất có thể là trang vừa mới được nạp vào bộ nhớ và chỉ mới truy cập 1 lần, còn các trang khác có tần số truy cập lớn hơn nhưng hoàn toàn có thể là trang mới nạp vào đó lại được sử dụng nhiều ngay sau đó.

Như vậy thực tế bất cứ chiến lược nào cũng có thể dẫn tới sự lựa chọn sai. Điều đó cũng rất đơn giản bởi vì chúng ta không thể dự đoán chính xác tương lai. Do đó chúng ta cần tìm chiến lược cho phép lựa chọn đúng với xác suất cao nhất và có chi phí thấp.

Chiến lược NUR

Một trong những thuật toán thông dụng, gần với chiến lược LRU và có thời gian trễ thấp là chiến lược NUR. Chúng ta cũng thấy rằng có thể xảy ra trường hợp trang không được sử dụng trong thời gian gần nhất lại có thể là trang được sử dụng trong tương lai gần và ta lại phải nạp trang đó vào bộ nhớ.

Vì muốn loại bỏ trang trong thời gian nằm trong bộ nhớ không bị thay đổi, trong thuật toán NUR sử dụng hai bit phần cứng cho mỗi trang:

```
1* Bit flag đọc (read)
2* bit flag thay đổi (write)
```

Chiến lược NUR thực hiện như sau: Đầu tiên các bit flag truy cập (đọc/ghi) của tất cả các trang đều đặt bằng 0. Khi đọc dữ liệu trang nào đó thì bit flag đọc của nó bằng 1, còn trường hợp nội dung nó thay đổi thì bit flag ghi bằng 1. Khi cần loại bỏ trang: đầu tiên chúng ta thử tìm trang mà không xảy ra thao tác đọc (bit flag đọc bằng 0). Trong trường hợp không tìm được chúng ta không còn cách nào khác là phải chọn trang đã bị đọc, khi đó cần kiểm tra xem trang đó có bị thay đổi hay không (bit flag ghi) và thử tìm trang không bị thay đổi, bởi vì với trang đã bị thay đổi chúng ta buộc phải ghi vật lý ra bộ nhớ ngoài. Nếu tất cả đều bị thay đổi thì chúng ta không còn lựa chọn nào khác.

Trong các hệ đa nhiệm, bộ nhớ (tất nhiên) hoạt động với cường độ cao và do đó sớm hay muộn phần lớn các trang đều có bit flag đọc bằng 1 và chúng ta không thể lựa chọn theo bit flag đọc một cách đúng đắn. Một trong các giải pháp thông dụng là theo chu kỳ tất cả các bit flag đọc đều reset về 0. Tất nhiên trong trường hợp này hoàn toàn có nguy cơ là các trang được sử dụng nhiều có thể bị loại vì tất cả các bit flag đọc đều bị reset về 0, nhưng vì chúng được sử dụng nhiều nên các bit flag đọc của chúng sẽ nhanh chóng bằng 1.

Việc phân loại trên dẫn tới có 4 nhóm:

	Giá trị bit đọc	Giá trị bit ghi
Nhóm 1	0	0
Nhóm 2	0	1
Nhóm 3	1	0
Nhóm 4	1	1

Sự lựa chọn sẽ diễn ra với các nhóm số nhỏ trước và chỉ khi không có kết quả mới đến lượt các nhóm có số lớn hơn. Để ý rằng nhóm hai có vẻ như không thể có vì là nhóm các trang có ghi mà không truy cập, nhưng cần chú ý là các bit flag đọc bị reset về 0 theo chu kỳ, do đó tình huống trên có thể xảy ra.

4.3.2. Tính địa phương

Phần lớn các chiến lược điều khiển bộ nhớ đều dựa trên khái niệm tính địa phương – locality, bản chất của nó là phân bố yêu cầu truy cập bộ nhớ của tiến trình là không đều mà tập trung tại một vùng.

Tính địa phương thể hiện cả theo thời gian và không gian. Địa phương theo thời gian đó là tính tập trung (Concentrate) theo thời gian. Ví dụ như vào lúc 12 giờ trưa trời nắng, khi đó với xác suất lớn (tất nhiên là không tuyệt đối) có thể nói rằng trời cũng nắng vào lúc 11giờ 30 phút (quá khứ) và lúc 12 giờ 30 phút (tương lai). Tính địa phương theo không gian thể hiện rằng các đối tượng lân cận cũng có cùng tính chất. Ví dụ cũng về thời tiết, nếu hôm nay tại Hà Nội trời nắng thì có thể (không phải tuyệt đối) nói rằng ở Hà Tây cũng nắng.

Tính chất địa phương cũng xuất hiện cả trong công việc của hệ điều hành, nói riêng là điều khiển bộ nhớ. Tính chất này mang tính kinh nghiệm hơn là lý thuyết. Địa phương không đảm bảo hoàn toàn nhưng xác suất theo nó là lớn. Ví dụ trong các hệ thống tổ chức theo trang, người ta theo dõi thấy rằng các tiến trình thường xuyên truy cập đến một số trang nhất định và các trang đó nằm liền nhau. Tuy nhiên điều đó không có nghĩa là tiến trình chỉ sử dụng các trang đó và không truy cập các trang khác. Điều đó chỉ đơn giản chứng tỏ rằng tiến trình thường truy cập đến một số trang nhất định nào đó trong khoảng thời gian nào đó.

Tính địa phương đối với các hệ thống tính toán có thể giải thích nếu để ý đến cách thiết kế chương trình và tổ chức dữ liệu:

1. Địa phương theo thời gian có nghĩa là các ô nhớ vừa được truy cập đến sẽ có xác suất lớn lại được truy cập trong tương lai gần. Điều đó do các lý do sau:

- Các vòng lặp chương trình.
- Các thủ tục của chương trình.
- Cơ chế stack.
- Các biến đếm.

2. Địa phương theo không gian có nghĩa là trong việc truy cập bộ nhớ, khi đã đọc một ô nhớ nào đó thì với xác suất lớn các ô nhớ bên cạnh cũng sẽ được truy cập. Xuất phát từ các lý do sau:

- Tổ chức dữ liệu theo các mảng.
- Chương trình thực hiện tuần tự từng lệnh một.
- Xu hướng người lập trình đưa các biến có liên quan cạnh nhau.

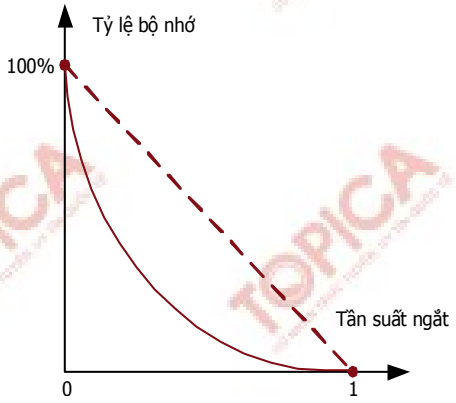
Có lẽ hệ quả quan trọng nhất của địa phương trong việc truy cập bộ nhớ là chương trình có thể làm việc hiệu quả khi trong bộ nhớ có các trang active (thường xuyên truy cập nhất).

Đã có nhiều nghiên cứu về tính địa phương. Trên hình 4.3.2a biểu diễn phân bố truy cập bộ nhớ đến các trang của tiến trình. Từ đồ thị ta thấy rằng trong một khoảng thời gian nhất định, tiến trình truy cập phần lớn đến chỉ một số trang nào đó.

Hình 4.3.2a

Hình 4.3.2b cũng thể hiện tính locality. Trên hình vẽ này biểu diễn sự phụ thuộc giữa tần số xuất hiện ngắt Missing Page Fault với kích thước bộ nhớ dành cho tiến trình để chứa các trang của nó.

Đường thẳng biểu diễn sự phụ thuộc đó trong trường hợp tiến trình truy cập bộ nhớ một cách ngẫu nhiên tức là xác suất truy cập đến từng trang là như nhau. Đường cong biểu diễn sự phụ thuộc trong thực tế phần lớn các tiến trình. Nếu như số Page Frame dành cho tiến trình giảm thì tồn tại một khoảng nào đó mà trong khoảng đó tần số xuất hiện ngắt Missing Page Fault tăng không đáng kể. Nhưng từ một điểm nhất định thì từ đó trở đi số lần xuất hiện ngắt Missing Page Fault tăng đột ngột.



Hình 4.3.2b

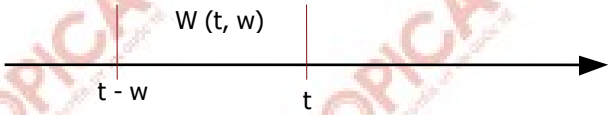
Chúng ta nhận thấy rằng khi mà các trang active còn ở trong bộ nhớ thì số lần xuất hiện ngắt thay đổi không đáng kể. Còn khi một hay nhiều trang active không có trong bộ nhớ thì số lần ngắt tăng đột ngột (vì tiến trình thường xuyên truy cập tới các trang Active).

4.3.3. Tập hợp các trang làm việc

Denning đã đề xuất đánh giá tần số nạp trang cho chương trình theo khái niệm gọi là 'Working Set Theory Of Program Behavior'. Nói một cách đơn giản thì Working Set là tập hợp các trang mà tiến trình thường xuyên truy cập tới. Denning cho rằng để đảm bảo sự làm việc hiệu quả của chương trình thì cần thiết phải nạp tất cả các trang active của nó vào bộ nhớ. Trong trường hợp ngược lại thì có thể làm tăng số ngắt Missing Page Fault.

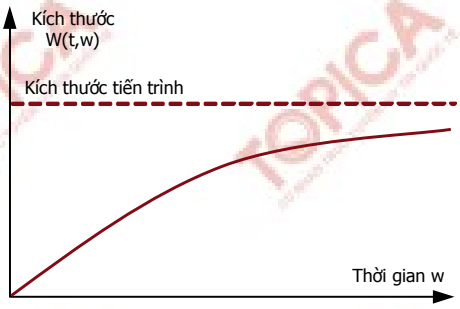
Chiến lược điều khiển bộ nhớ theo Working Set cố gắng để tất cả các trang Active (Working Set) nằm trong bộ nhớ. Giải quyết trường hợp có đưa thêm một tiến trình mới vào hoạt động (tăng hệ số đa nhiệm) hay không, cần phải dựa trên điều kiện là bộ nhớ trống có đủ nạp Working Set – tất cả các trang Active của tiến trình đó hay không. Giải pháp này, đặc biệt đối với các tiến trình đầu tiên, thường áp dụng một cách cảm tính, vì hệ thống không thể biết trước được tập hợp các trang active của tiến trình.

Tập hợp làm việc của tiến trình W(t, w) vào thời điểm t là tập hợp các trang mà tiến trình truy cập trong khoảng thời gian từ t – w đến w (hình 4.3.3a).



Hình 4.3.3a

Biến w gọi là kích thước cửa sổ của tập hợp Working Set, việc lựa chọn giá trị w đóng vai trò quyết định trong hoạt động của cơ chế điều khiển bộ nhớ theo Working Set. Hình 4.3.3b biểu diễn mối quan hệ giữa kích thước cửa sổ w với kích thước của Working Set. Đồ thị này xuất phát từ định nghĩa toán học của Working Set chứ không phải từ kết quả theo dõi thực tế. Một tập hợp làm việc Working Set thực sự là tập hợp các trang phải nằm trong bộ nhớ để tiến trình có thể hoạt động tốt.

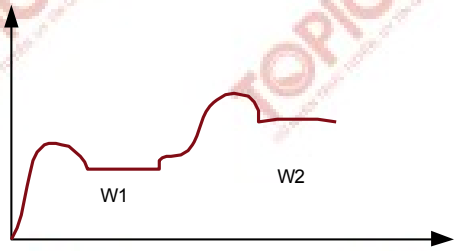


Hình 4.3.3b

Vào thời gian hoạt động của tiến trình, Working Set của nó luôn biến đổi. Đôi khi có thêm một số trang và cũng đôi khi bớt đi. Thỉnh thoảng xảy ra sự biến đổi đột ngột, ví dụ như khi tiến trình chuyển giai đoạn, đòi hỏi một tập hợp Working Set hoàn toàn mới. Do đó bất cứ điều giả sử nào về kích thước và nội dung ban đầu của Working Set sẽ không luôn đúng đắn với quá trình hoạt động về sau của tiến trình. Điều này làm phức tạp thêm việc thực hiện điều khiển bộ nhớ theo chiến lược Working Set.

Trên hình 4.3.3c biểu diễn việc sử dụng bộ nhớ trong chiến lược điều khiển bộ nhớ theo Working Set.

Đầu tiên, vì tiến trình không yêu cầu các trang active toàn bộ ngay mà theo từng trang, nên nó dần dần nhận được đủ bộ nhớ để chứa tất cả các trang active (Working Set). Sau đó trong một khoảng thời gian nào đó, kích thước bộ nhớ được sử dụng này ổn định không đổi, vì tiến trình thường truy cập đến các trang active của Working Set đầu tiên này. Theo thời gian tiến trình chuyển sang tập hợp làm việc (Working Set) tiếp theo, được thể hiện bằng đường cong đi từ tập hợp thứ 1 đến tập hợp thứ 2. Đầu tiên đường cong này là cao hơn tập hợp 1 bởi các trang của tập hợp làm việc thứ 2 nhanh chóng được nạp vào bộ nhớ theo yêu cầu truy cập của tiến trình. Lúc đó hệ thống không thể ngay lập tức nhận ra là tiến trình chỉ đơn giản mở rộng Working Set hay nó đang chuyển sang Working Set tiếp theo. Sau khi tiến trình bắt đầu ổn định truy cập các trang của Working Set tiếp theo, hệ thống thấy rằng trong cửa sổ Working Set, tiến trình chỉ truy cập đến số trang ít hơn trước kia và hệ thống thu gọn kích thước bộ nhớ cấp cho tiến trình đến mức bằng với kích thước của Working Set mới. Mỗi lần khi chuyển từ Working Set này sang Working Set mới, đường cong đầu tiên nâng lên và sau đó hạ xuống thể hiện cách mà hệ thống nhận ra điều đó.



Hình 4.3.3c

Hình 4.3.3c thể hiện một trong những khó khăn trong việc thực hiện điều khiển bộ nhớ theo chiến lược working–set, vấn đề ở chỗ working–set thay đổi theo thời gian, ngoài ra workingset tiếp theo của tiến trình có thể khác rất nhiều với Working Set trước đó. Chiến lược Working Set phải chú ý điều này để khỏi xảy ra tình trạng quá tải bộ nhớ.

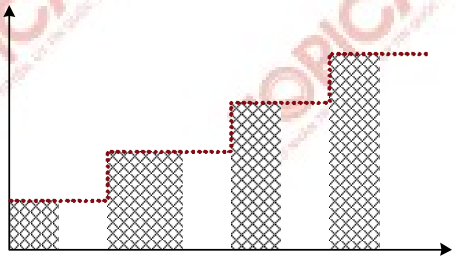
4.3.4. Các chiến lược nạp trang

Nạp trang theo yêu cầu (Demand Paging):

Trước kia người ta cho rằng tốt nhất là các trang cần thiết của tiến trình được nạp vào bộ nhớ theo yêu cầu. Tức là không có trang nào được nạp vào bộ nhớ từ bộ nhớ ngoài khi mà tiến trình chưa thực sự có yêu cầu truy cập đến nó. Chiến lược này có mặt mạnh là vì:

- Lý thuyết tính toán nói rằng con đường thực hiện của chương trình không thể dự đoán chính xác (do các lệnh điều kiện...) do đó bất cứ cố gắng nào nạp trước các trang vào bộ nhớ vì cho rằng chúng sẽ cần đến đều có thể không đúng.
- Nạp trang theo yêu cầu đảm bảo rằng trong bộ nhớ sẽ chỉ có các trang thực sự cần cho hoạt động của tiến trình.
- Chi phí cho việc xác định trang nào cần nạp vào bộ nhớ là nhỏ nhất. Còn trong trường hợp nạp trang theo dự đoán có thể cần chi phí tính toán đáng kể.

Tuy vậy chiến lược nạp trang theo yêu cầu (hình 4.3.4a) có những vấn đề của mình. Tiến trình phải tích cóp các trang cần thiết cho nó trong bộ nhớ theo từng trang một. Khi có yêu cầu đến một trang mới bất kỳ, tiến trình bắt buộc phải chờ khi mà trang chưa được nạp xong vào bộ nhớ. Phụ thuộc vào số lượng trang của tiến trình nằm trong bộ nhớ, các khoảng thời gian chờ này sẽ ngày càng đắt vì tiến trình chiếm ngày càng nhiều bộ nhớ. Hình 4.3.4a biểu diễn sự phụ thuộc 'không gian – thời gian' sự phụ thuộc thường được áp dụng trong hệ điều hành để đánh giá sự sử dụng bộ nhớ của tiến trình.



Hình 4.3.4a

Tích giữa tỷ số 'không gian' và 'thời gian' tương ứng với phần gạch chéo trong sơ đồ. Nó là chỉ số thể hiện sự sử dụng bộ nhớ cả về thời gian và không gian (dung lượng) của tiến trình.

Nạp trang dự đoán trước (Anticipatory Paging):

Một chiến lược quan trọng trong việc điều khiển tài nguyên hiệu quả có thể phát biểu như sau: 'mức độ sử dụng của mỗi tài nguyên phải xác định tương đối với giá trị của nó'. Như thế, ngày nay giá phần cứng ngày càng giảm vì vậy giá tương đối của thời gian máy ngày càng giảm so với giá trả cho người. Ngày nay, các nhà thiết kế hệ điều hành tích cực tìm kiếm theo con đường giảm thời gian chờ đợi của user. Một trong những phương pháp tốt theo hướng này là nạp trang theo dự đoán trước.

Trong cách này, hệ điều hành cố gắng dự đoán các trang mà tiến trình sẽ cần đến, và khi bộ nhớ có đủ chỗ trống thì nạp các trang đó vào. Do đó khi tiến trình cần truy cập đến trang mới thì nó đã có trong bộ nhớ. Nếu như sự chọn lựa là đúng thì chúng ta đã rút ngắn đáng kể thời gian thực hiện tiến trình.

Phương pháp nạp trang dự đoán trước có các ưu điểm sau:

- Nếu xác suất chọn đúng là tương đối lớn thì chúng ta làm giảm đáng kể thời gian thực hiện tiến trình. Do đó hoàn toàn có lợi khi thực hiện các cơ chế dự đoán trước dù đáp số không phải đúng tuyệt đối.
- Trong nhiều trường hợp hoàn toàn có thể tìm được lời giải đúng nếu việc đó có thể thực hiện với chi phí tương đối thấp để tiến trình hoạt động nhanh hơn, trong khi đó không ảnh hưởng xấu đến các tiến trình khác.
- Bởi phần cứng ngày càng rẻ do vậy việc giải pháp không tối ưu trở nên ít quan trọng hơn, chúng ta có thể cho phép mình thêm các trang mà cơ chế dự đoán xác định vào bộ nhớ.

Giải phóng các trang:

Trong chiến lược điều khiển bộ nhớ theo Working Set, các chương trình thông báo cho hệ điều hành các trang mà chúng cần sử dụng. Các trang không cần nữa phải loại ra khỏi Working Set bằng cách nào đó. Thường tồn tại khoảng thời gian trong đó các trang không cần vẫn còn ở trong bộ nhớ.

Khi nhận thấy rằng trang nào đó không cần nữa, user có thể tự gửi tín hiệu và giải phóng trang đó. Điều đó có thể làm giảm thời gian trễ nếu để cho hệ thống xác định.

Việc tự giải phóng bộ nhớ có thể loại bỏ lãng phí bộ nhớ nhưng phần lớn các user không thể làm điều đó vì thậm chí anh ta còn không biết về các trang bộ nhớ. Ngoài ra, thêm các lệnh giải phóng trang trong chương trình ứng dụng làm phức tạp thêm việc lập trình.

Có thể hy vọng rằng vấn đề này được giải quyết với sự giúp đỡ của compiler và hệ điều hành – chúng sẽ tự nhận biết tình huống khi cần loại bỏ trang và làm một cách nhanh chóng hơn là khi sử dụng Working Set.

Kích thước trang:

Trong các hệ thống tổ chức theo trang, bộ nhớ thường chia ra thành các Page frame có kích thước cố định. Vấn đề là kích thước các trang và Page frame phải lựa chọn thế nào.

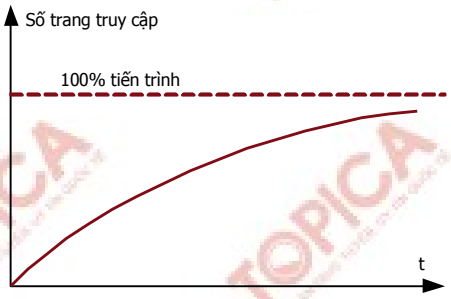
Để có lựa chọn đúng cần phải phân tích cẩn thận và hiểu biết toàn diện về các tính năng của thiết bị phần cứng, phần mềm và cả phạm vi ứng dụng của hệ thống. Chúng ta không có câu trả lời tổng quát cho mọi trường hợp. Tuy nhiên có một số vấn đề cần chú ý khi lựa chọn kích thước trang:

- Kích thước trang càng nhỏ thì số lượng trang và Page frame càng lớn và bảng ánh xạ trang cũng càng lớn. Đối với các hệ thống mà bảng ánh xạ trang nằm trong bộ nhớ thì chúng ta cần thiết phải tăng kích thước trang. Việc sử dụng không hiệu quả bộ nhớ do kích thước bảng quá lớn gọi là Page Fragmentation. Chúng ta cần nhận thấy rằng ngày nay vấn đề đó cũng không nghiêm trọng quá, do giá thành bộ nhớ giảm trong khi dung lượng lại tăng.
- Khi các trang có kích thước lớn thì lúc nạp trang vào bộ nhớ lại phải nạp khối lượng thông tin lớn hơn, và nói chung không phải bao giờ cũng bắt buộc. Điều đó làm xuất hiện sự cần thiết giảm kích thước trang.
- Vì sự trao đổi giữa bộ nhớ với bộ nhớ ngoài (HDD,...) chiếm khá nhiều thời gian do đó cần phải giảm tối thiểu số lần trao đổi xảy ra trong quá trình thực hiện chương trình và chúng ta cần giảm kích thước trang.
- Chương trình thường có tính chất địa phương, ngoài ra vùng mà tiến trình thường truy cập có kích thước không lớn. Do đó, sự giảm kích thước trang cần làm cho chương trình có số lượng trang active gọn hơn.

Chương trình và sự nạp trang:

Tổ chức theo trang là một cơ chế, mô hình hiệu quả. Đã có nhiều công trình nghiên cứu, phân tích hoạt động của tiến trình với tổ chức bộ nhớ theo trang.

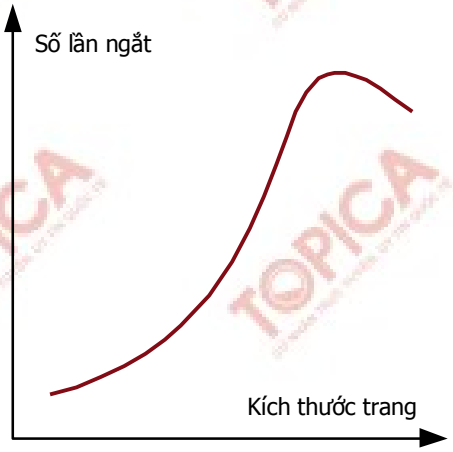
Trên hình 4.3.4b biểu diễn sự phụ thuộc giữa phần trăm số trang được tiến trình truy cập và thời gian (bắt đầu từ lúc thực hiện tiến trình).



Hình 4.3.4b

Đoạn đầu đường cong dốc đứng, thể hiện rằng ngay sau khi bắt đầu, tiến trình truy cập đến phần đáng kể các trang của mình. Theo thời gian đường cong ngang dần và nó dần tới 100%. Tất nhiên một số tiến trình sử dụng toàn bộ 100% số trang, nhưng cũng có nhiều tiến trình có thể trong thời gian dài không truy cập đến tất cả các trang của mình.

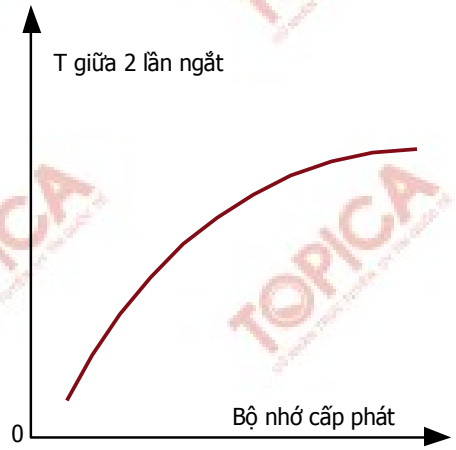
Hình 4.3.4c thể hiện hậu quả của việc thay đổi kích thước trang trong khi kích thước bộ nhớ không đổi. Từ đồ thị ta thấy rằng đối với tiến trình đang hoạt động, số lần ngắt Missing Page fault tăng lên cùng với sự tăng kích thước trang. Điều đó là do khi kích thước trang tăng trong khoảng bộ nhớ cố định thì số đoạn code và dữ liệu không cần truy cập cũng tăng, tức là tỷ lệ của code và dữ liệu hữu ích (có truy cập) giảm. Vì thứ tự truy cập bộ nhớ của tiến trình không phụ thuộc lượng bộ nhớ cấp cho nó, kết quả là số lần ngắt Missing Page Fault nhiều hơn.



Hình 4.3.4c

Hình 4.3.4d biểu diễn sự thay đổi thời gian trung bình giữa các lần ngắt mising Page fault khi tăng số Page Frame cấp cho tiến trình. Đường cong đơn điệu tăng–tức là số Page Frame cấp cho tiến trình càng lớn thì khoảng thời gian giữa các lần ngắt Missing Page Fault càng lâu.

Chúng ta thấy trên đồ thị có điểm uốn, mà từ đó trở đi độ lệch của đường cong giảm hẳn đi, Điểm này tương ứng với lúc mà tất cả Working Set của tiến trình nằm trong bộ nhớ. Đầu tiên khoảng thời gian giữa hai lần ngắt tăng nhanh tương ứng với sự tăng của phần Working Set được nằm trong bộ nhớ. Sau khi bộ nhớ đã đủ lớn để nạp toàn bộ Working Set thì xuất hiện điểm uốn, chỉ ra rằng từ đó, khi cấp thêm Page frame cũng không làm tăng đáng kể thời gian giữa các lần ngắt.



Hình 4.3.4d

Tóm lại tất cả những yếu tố chúng ta đã xem xét đã chứng tỏ sự ảnh hưởng của khái niệm Working Set và sự cần thiết phải giảm kích thước trang. Theo sự phát triển của tin học, các kết quả này cũng có thể phải phân tích lại.

TÓM LƯỢC CUỐI BÀI

Các bạn đã được học về quản lý bộ nhớ trong hệ thống.

Ta cần ghi nhớ các vấn đề sau:

- Cách tổ chức, điều khiển bộ nhớ (bộ nhớ vật lý).
- Các chế độ bảo vệ bộ nhớ trong hệ thống đơn và đa nhiệm.
- Các cách tổ chức bộ nhớ ảo: tổ chức theo trang, theo đoạn – phân đoạn
- Các cách biến đổi địa chỉ theo trang và theo đoạn trong bộ nhớ ảo.
- Các chiến lược điều khiển bộ nhớ ảo.
- Các chiến lược nạp trang.
- Giải bài toán về biến đổi địa chỉ, và điều khiển bộ nhớ ảo.

CÂU HỎI TỰ LUẬN

Câu 1. Tìm số trang bị lỗi khi sử dụng thuật toán thay thế trang LRU trên chuỗi truy xuất trang 1, 2, 3, 0, 1, 2, 4, 1, 2, 3, 4, 5, 1, 3, 2, 0, 1, 2 với tổng khung trang là 3 và 5?

Câu 2. Cho ví dụ để loại bỏ cùng 1 trang dùng hai cơ chế FIFO và LRU. Hãy giải thích.

Câu 3. Nêu cơ chế biến đổi địa chỉ động trong hệ thống kết hợp trang và đoạn?

Câu 4. Vẽ sơ đồ biến đổi địa chỉ logic (địa chỉ ảo) thành địa chỉ vật lý trong kỹ thuật phân trang. Cho kích thước trang và kích thước khung trang là 100K, địa chỉ bắt đầu cấp phát trong bộ nhớ là 0K. Cho bảng trang của P như sau:

P	P'
0	7
1	2
2	5
3	4

Tính địa chỉ vật lý tương ứng với các địa chỉ logic sau: (0,30K); (2,70K); (3,25K).

Câu 5. Nêu chiến lược loại bỏ FIFO,FIFO dị thường, cho ví dụ.

BÀI TẬP TRẮC NGHIỆM

1. Các thuật toán nào sau đây thuộc loại thuật toán thống kê?

a) LFU b) LRU c) NRU d) Câu a và c.

2. Với thuật toán thay thế trang FIFO sử dụng 3 khung trang, số hiệu các trang đi vào lần lượt là: 1, 2, 3, 4, 1, 2 như bảng sau thì có bao nhiêu trang bị lỗi?

1	2	3	4	1	2
1	12	123	423	413	412

a) 6 lỗi. b) 5 lỗi. c) 4 lỗi. d) Không có câu nào đúng.

3. Kỹ thuật cấp phát nào sau đây loại bỏ được hiện tượng phân mảnh ngoại vi?

a) Phân đoạn. b) Phân trang. c) Cấp phát liên tục. d) Đáp án a, b.

4. Có mấy cách biến đổi địa chỉ trong hệ thống tổ chức theo trang

a) 2. b) 3. c) 4. d) Cả 3 đáp án trên đều sai.

5. Thanh ghi biên dùng để làm gì?

a) Lưu trữ dữ liệu. b) Lưu trữ các bit cờ. c) Bảo vệ bộ nhớ. d) Điều khiển bộ nhớ.

6. Thuật toán thay thế trang dùng thời điểm cuối cùng trang được truy xuất là thuật toán :

a) FIFO. b) LRU. c) NRU. d) LFU.

7. Cho trình tự các trang như sau: P1→ P2 → P3 → P2 → P1. Nếu loại bỏ trang P3 thì loại bỏ này thuộc cơ chế nào?

a) FIFO b) LRU c) LFU d) Đáp án b và c.