CHƯƠNG 3. QUẢN LÝ BỘ NHỚ

3.0. Quan niệm về quản lý bộ nhớ

Một trong các phương tiện điều hành quan trọng là bộ nhớ chính. Quản lý tài nguyên bộ nhớ là một đề tại hữu ích và cấp bách, nó quyết định khả năng hiện hữu của một hệ thống máy tính. Theo đó, chúng ta phân biệt ba phạm vi, mà trong đó, các chiến lược khác nhau được sử dụng để quản lý bộ nhớ.

- *Các chương trình người sử dụng*:

Nhiệm vụ chính là bao gồm: việc quản lý một cách tối ưu không gian lưu trữ của tiến trình xung quanh các yêu cầu lưu trữ đặc biệt của chương trình. Điều này được thực hiện nhờ các bộ phận chương trình (thí dụ điều hành bộ nhớ) hay nhờ các chương trình thu gom rác.

- *Bộ nhớ chính*:

Vấn đề chủ yếu là phân bổ tối ưu không gian bộ nhớ chính trên các tiến trình riêng lẻ. Theo nguyên tắc chung, nhiệm vụ được trợ giúp nhờ các đơn vị phần cứng chuyên dụng. Đặc biệt, trong hệ thống đa vi xử lý, điều quan trọng là phải tránh các tranh chấp, nếu có nhiều tiến trình cùng muốn chiếm một không gian lưu trữ. Trong trường hợp này, các kỹ thuật viên sẽ giúp đỡ thêm, như việc truy cập bộ nhớ không đồng dạng (*NUMA*), thí dụ khi chiếm dụng các bộ vi xử lý, nó được tách chia ra trên bộ nhớ cục bộ hay bộ nhớ toàn cục.

- *Bộ nhớ quảng đại*:

Việc tách khỏi sự quản lý các files sẽ có những files chuyên dụng, mà với nó, dung lượng của file được phân chia và được quản lý. Thí dụ file swap được chuyển dịch trên các tiến trình, mà những tiến trình này không chiếm nhiều không gian bộ nhớ chính.

Đầu tiên, chúng ta nghiên cứu kỷ thuật trực tiếp che phủ bộ nhớ, mà nó tìm thấy việc ứng dụng ở hầu hết các chương trình người sử dụng hoặc ở các hệ điều hành đơn giản.

3.1. Che phủ trực tiếp bộ nhớ

Trong buổi đầu của việc sử dụng máy tính, công việc cụ thể là chế ngự chiếc máy tính với bộ nhớ của nó cho từng Job. Nhân của hệ điều hành thường bao gồm một sự tổng hợp các thủ tục xuất nhập. Đó chính là các thủ tục thư viện mà chúng được gắn thêm vào Job. Nếu người ta phải đặt Job trở lại, để đầu tiên thực hiện một Job khác, thì do đó, tất cả các dữ liệu của tiến trình được di chuyển trên bộ nhớ quảng đại và được phép di chuyển các dữ liệu của tiến trình mới từ bộ nhớ quảng đại tới bộ nhớ chính. Tuy nhiên, việc di dịch vào ra này (swapping) cần phải có thời gian. Nếu chúng ta chấp nhận thời gian truy cập trung bình của ổ đĩa cứng khoảng 10ms và nếu các dữ liệu chuyển thẳng trực tiếp (*DMA-Tranfer*) với tỷ lệ vận chuyển khoảng 500kByte/s, do đó, chúng ta cần 110 ms cho một chương trình có dung lượng 50kB để viết chương trình lên đĩa và cùng thời gian đó để lấy một chương trình tương tự từ ổ đĩa, và như vậy, hợp lại thành 220ms trôi qua cho một công việc bình thường. Việc di chuyển tráo đổi này đã hạn chế sự đổi chiều giữa các tiến trình rất mạnh.

Từ cơ sở này, điều cần thiết là, phải bảo vệ các dữ liệu của nhiều tiến trình tồn tại đồng thời ở trong bộ nhớ, tức là, ngay lập tức bộ nhớ phải tồn tại một cách đầy đủ. Tuy nhiên, nó

cũng dẫn tới một vấn đề, người ta phải phân chia bộ nhớ như thế nào đó, để nhiều tiến trình có điều kiện nhận chỗ (?). Câu hỏi này đặt ra không chỉ đối với bộ nhớ chính, mà tất nhiên cả đối với không gian lưu trữ của ổ đĩa cứng, nó còn gọi là không gian tráo đổi (*swap space*).

3.1.1. Sắp xếp bởi các bảng cố định

Cách tốt nhất là, từ hình ảnh thu nhỏ của bộ nhớ, người ta tạo lập thành những bảng che phủ bộ nhớ. Mỗi đơn vị của một bảng như vậy (thí dụ Bit) được sắp xếp cho một đơn vị lớn hơn (thí dụ một từ 32bit). Nếu một từ bị che phủ, thì do đó, Bit đạt giá trị 1 hay 0.

Hình 3.1 chỉ ra một sự sắp xếp như vậy. Khi đơn vị bộ nhớ được chọn một đoạn lơn hơn (thí dụ 4kB), do đó, việc sắp xếp các khối dữ liệu A,B,C (là những chương trình) được các phương tiện điều hành quản lý.

Một bảng che phủ như thế tự dùng 100 kByte đối với bộ nhớ 3,2 MByte. Nếu một không gian trống bị che phủ, do đó, một bảng tổng hợp sẽ được tìm kiếm trên một số lượng phù hợp các số 0.

Hình 3.1 ----------

3.1.2. Sắp xếp bởi danh sách cụ thể

Thực chất không gian trống cũng như không gian đã bị che phủ thì hầu như dài hơn sự mô tả về chúng như đã nói trên. Điều đó thì có lợi: đáng lẽ một bảng cố định thì phải làm một danh sách điền đầy việc che phủ bộ nhớ, nhưng ở đây, chúng được liên kết với nhau ở trong dãy tuần tự của các địa chỉ nhớ (nhờ bộ chỉ thị). Do đó, thí dụ minh họa ở trong hình 3.1 được làm sáng tỏ thêm như ở trong hình 3.2 dưới đây.

Hình 3.2----------

Việc điền vào danh sách bao gồm ba phần: địa chỉ bắt đầu của bộ nhớ, độ dài và chỉ số (dấu hiệu chỉ dẫn) cho việc điền thêm kế tiếp.

Một danh sách che phủ được phân thành 2 danh sách: một danh sách cho khoảng bị che phủ mà việc điền vào của nó được khai báo nhờ khối điều khiển tiến trình PCB và một danh sách chỉ dành cho khoảng chưa được che phủ. Nếu chúng ta sắp xếp danh sách này theo độ lớn của khoảng trống thì do đó, một cách bình thường, danh sách đầy đủ không cần phải tìm kiếm. Tuy nhiên, qua đó, sự hoà hợp của khoảng trống và khoảng giới hạn sẽ bị gây trở ngại. Việc khai báo hai lần sẽ tạo ra khả năng tìm kiếm danh sách theo hai hướng.

Thí dụ *về quản lý theo xấp (heap management):*

Việc quản lý bộ nhớ bằng cách dựa vào danh sách là một thí dụ có ý nghĩa cho việc quản lý bộ nhớ theo xấp; xấp được mỗi chương trình người sử dụng quản lý. Thí dụ ở máy tính MODULA-2, điều đó có ý nghĩa để mô tả việc thực thi các thủ tục ALLOCATE()vàDEALLOCATE().Qua việc ghi tên loại, một không gian trống được quản lý,

mà nó được kết nối với nhau thành một danh sách liên tục và nó được dẫn vào nhờ móc treo hay nhờ một chỉ thị có tên rootPtr:

```
TYPE        EntryPtr = POINTER TO EntryHead;
            EntryHead = RECORD
                            next  : EntryPtr;
                            size : CARDINAL;
                        END;
VAR         rootPtr : EntryPtr;
            nextStart : EntryPtr;
            FirstFree, NoOfEntryHeads : CARDINAL;
```

Đầu tiên, mỗi không gian trống được viết đè lên bởi mục nhập (*Entry*). Một không gian trống đối với danh sách thì không thể nhỏ hơn chiều dài TSIZE (*EntryHead*) của một mục nhập. Nếu một không gian trống được ghi, do đó, một mục nhập mới có dạng *EntryHead* được thiết lập và được treo vào danh sách trống;nếu không gian trống quá nhỏ, khi đó, nó được bỏ qua.

Một cách đơn giản hơn và nhanh hơn để dẫn tới một danh sách riêng lẻ, đó là danh sách của các không gian trống. Tuy vậy, nó thì bảo đảm hơn để dẫn tới một danh sách được che phủ, cũng như để có thể kiểm tra tình trạng và độ lớn của không gian được việc trả lại từ không gian đã bị che phủ và nhờ vậy, để làm sang tỏ việc lập trình thiếu hụt của chương trình người sử dụng. Đặc biệt ở giai đoạn gỡ rối, điều đó thì có lợi; sau đó, một sự kiểm tra có thể được ngắt khỏi khi tối ưu thời gian thực hiện ở các giai đoạn muộn hơn.

3.1.3. Các chiến lược che phủ

Một cách độc lập với cơ cấu của danh sách che phủ bộ nhớ, có những chiến lược khác nhau để lựa chọn một cách thích hợp nhất từ vùng nhớ chưa bị che phủ. Mục đích của các chiến lược là ở chỗ: phải giữ cho được một vùng trống dù nhỏ, nhưng nó có ý nghĩa rất lớn. Sau đây là những chiến lược quan trọng:

- *FirstFit* (đầu tiên vừa đủ):

Trước hết, vùng trống của bộ nhớ phải đủ lớn để phù hợp cho việc che phủ. Tức là, bao giờ cũng còn lại một phần thừa chưa bị che phủ.

- *NextFit* (kế cạnh vừa đủ):

Chiến lược *FirstFit* dẫn tới trong khoảng trống đầu tiên chỉ còn thừa lại một phần nhỏ, nhưng phần còn lại này luôn luôn được tìm kiếm trở lại. Để loại bỏ điều này, người ta xuất phát giống như chiến lược *FirstFit*, nhưng ở lần kế tiếp, việc tìm kiếm được tiếp tục tại một vị trí và cũng tại đó, người ta ngừng lại.

- *BestFit* (vừa đủ nhất):

Danh sách toàn thể cũng như bảng che phủ được tìm kiếm, cho tới khi người ta tìm thấy một khoảng thích hợp, mà lập tức, nó đủ để tiếp nhận khoảng che phủ.

- *WorstFit* (đáng vừa đủ):

Nếu khoảng trống đang tồn tại là khoảng lớn nhất được tìm thấy, thì do đó, phần còn lại là có khả năng nhất và đủ lớn.

- *QuickFit* (vừa đủ để thoát khỏi):

Đối với mỗi loại che phủ, một danh sách đặc biệt được nói tới. Điều đó cho phép để tìm thấy những chỗ trống thích hợp một cách nhanh hơn. Nếu ở trong một hệ thống thông tin có độ dại 1kB được gởi đi một cách đều đặn, do đó, nó thì có lợi, để dẫn tới một danh sách đầy đủ cho việc che phủ 1kB và để hài lòng khi đạt được mọi sự thăm dò một cách nhanh chóng và không có sự pha trộn.

- *Buddy-systems* (những hệ thống thân hữu):

Bây giờ, người ta có thể mở rộng các quan điểm về chiến lược *QuickFit*, rằng đối với mỗi độ lớn bị che phủ, tốt nhất, bộ nhớ được dự đoán là một danh sách gồm những độ lớn là luỹ thừa của cơ số 2 và nó chỉ trao cho mỗi khoảng bộ nhớ một độ lớn cố định. Tất cả các yêu cầu phải được thiết lập trên luỹ thừa tiếp theo của cơ số 2. Nếu có một không gian nhớ dài 280 Bytes, tạm diễn giải:

$$280 \text{ Bytes} = 256 + 16 + 8 = 2^8 + 2^4 + 2^3$$

thì nó phải được tạo lập trên không gian nhớ $512 = 2^9$. Nếu không có một khoảng nhớ trống nào của độ lớn 2^k tồn tại, do đó, một đoạn nhớ có độ lớn 2^{k-1} phải được phân thành hai đoạn. Cả hai đoạn là đối tác thân hữu, được biểu thị một cách chính xác: địa chỉ bắt đầu của chúng thì giống hệt cho đến khi kBit ở trong địa chỉ của chúng. Thí dụ …XYZ0000… và XYZ1000 là các địa chỉ bắt đầu của đối tác. Điều đó được sử dụng để kiểm tra (từng bước) rất nhanh, liệu một khoảng bộ nhớ có được trở nên trống khi có một đối tác ở trong bảng che phủ, mà với bảng này, nó có thể làm lan ra với một khoảng lớn gấp đôi.

Cả hai quá trình, nếu vừa thực hiện việc tìm kiếm một khoảng trống thích hợp (cũng tựa như việc bẻ gãy riêng lẻ cần thiết của một đại lượng lớn hơn), thì cũng phải vừa thực hiện việc kết hợp chúng lại thành những đại lượng lớn hơn, để đạt được một cách quy nạp nhiều đối tác (nhiều luỹ thừa của cơ số 2).

Đánh giá chiến lược

Sự mô hình hoá so sánh các chiến lược khác nhau dẫn tới một sự đánh giá chi ly. Điều đã nhận ra rằng, chiến lược *FirstFit* sử dụng không gian thì tốt hơn các chiến lược *NextFit* và *WorstFit*, và một cách ngẫu nhiên nào đó cũng có thể tốt hơn chiến lược *BestFit*, do vậy, điều đó còn hướng tới chỉ để lại một phần rất nhỏ chưa bị che phủ. Một sự sắp xếp các nhân tử trong danh sách theo độ lớn của các dải sẽ làm giảm đi thời gian thực hiện các chiến lược *BestFit* và *WorstFit*.

Nếu chúng ta biết được nhiều hơn về sự phân bổ các yêu cầu bộ nhớ theo thời gian hay theo độ lớn che phủ, do đó, chiến lược *QuickFit* hay các thuật toán đặc biệt khác có thể đạt được các kết quả cao hơn.

Khả năng hữu hiệu của hệ thống đối tác thân hữu được đánh giá một cách ngắn gọn có sự tính toán như sau: Nếu chúng ta chấp nhận rằng, tất cả các luỹ thừa 2^n có độ lớn các khoảng là s, chúng được kéo dài với xác xuất như nhau $1/2^n$ (tức là có một cái gì đó xảy ra không chắc chắn, nhưng mà, người ta có thể phỏng đoán với một sự chập nhận thuyết phục). Do đó, yêu cầu bộ nhớ S_a phải đạt tối thiểu:

Công thức ----------

Vì: $\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$

Đối với sự che phủ thực tế, nó phải được thiết lập bởi luỹ thừa của cơ số 2, nghĩa là, đáng lẽ sự che phủ có dạng: 1,2,3,4,5,6,7,8,...,... 2^n

Chúng ta sử dụng 1,2,4,4,8,8,8,8,... 2^n,..., 2^n

Hay 2^0,2^1,2^2,2^2, 2^3,2^3,2^3,2^3... 2^n,...,2^n

Hay ta nhân được tích: 2^1 nhân với 2^2 nhân với ... 2^{n-1}

Vì vậy, sự che phủ trên thực tế là mỗi một 2^i nhân với 2^{i+1}

Do đó sự che phủ thực tế trung bình S_b được tính:

công thức ----------

Với điều đó, tỷ lệ tương quan giữa các giá trị trung bình S_a và S_b được xác định:

Công thức--------

Kết quả tính toán phỏng đoán của chúng ta cho thấy: Một phần tư của không gian bộ nhớ được kéo dài chưa sử dụng. Qua đó, hệ thống đối tác đặc trưng cho một phương pháp che phủ; tuy nhiên, phương pháp thì có nhanh, nhưng không hiệu quả lắm. Nguyên nhân là ở chỗ sự phân đoạn thô bởi việc tăng gấp đôi không gian bộ nhớ. Nếu người sửa chữa điều đó, thì do đó, hệ số sử dụng đối với bộ nhớ được nâng cao, tất nhiên, việc quản lý sẽ trở nên phức tạp hơn.

Việc phân mảnh và mẫu cắt:

Tuy rằng, nói chung, nếu những danh sách che phủ tạo lập nên một sự sắp xếp tốt đáp ứng mong muốn che phủ tới các khoảng trống của bộ nhớ, thì mặc dù, phương pháp che phủ bộ nhớ trực tiếp có khuynh hướng tới việc chia nhỏ bộ nhớ thành nhiều khoảng nhỏ chưa bị che phủ. Điều đó xảy ra một cách độc lập, liệu nó có đặt cơ sở cho việc che phủ bộ nhớ bởi mẫu cắt hay đặt cơ sở để phân chia bộ nhớ (đối với chương trình tổng thể ở bộ nhớ chính) bởi các khoảng trống nằm giữa các chương trình và mẫu cắt bên ngoài. Từ lý do này, việc làm chảy ra các khoảng trống là một trong các chức năng của việc quản lý bộ nhớ.

Đối với vấn đề phân bổ bộ nhớ cho việc nạp các tiến trình của các bộ nhớ quảng đại, có rất nhiều chiến lược. Nếu đầu tiên chúng ta che phủ bộ nhớ với các tiến trình lớn nhất để sử dụng các tiến trình nhỏ làm đầy các chỗ trống, thì do đó, điều này được tổng hợp trong một chiến lược định thời, mà nó có ý nghĩa đối lập với chiến lược định thời *Job ngắn nhất- trước nhất* và nó cũng đảm bảo thời gian làm việc lâu bền một cách đặc biệt.

Khác biệt với cái đó, người ta có thể đạt được một sự định thời đối với độ lớn của tiến trình, trong đó, người ta phân đoạn bộ nhớ tổng cộng thành một số lượng cố đinh các khoảng chia có độ lớn khác nhau. Mỗi độ lớn khoảng chia chứa đựng một hang đợi riên lẻ, do đó, các bảng sắp xếp bộ nhớ thì cố định và một sự phân mảnh sẽ không xuất hiện. Thí dụ, một giải pháp như vậy đã tồn tại ở trong hệ điều hành IBM (đa lập trình với sự cố định các con số của Job): OS/ MFT đối với hệ điều hành OS/360, đó là hệ thống máy tính lớn. Dĩ nhiên, một hệ

thống cố đinh không thay đổi sẽ mang lại cho nó một khả năng tải tồi các phương tiện điều hành.

3.2. Định vị logic và bộ nhớ ảo

Vì lý do khi quản lý bộ nhớ, để loại bỏ việc phải sử dụng xấu bộ nhớ, thì đã có những nỗ lực khác nhau để tìm thấy những giải pháp mới cho vấn đề này.

3.2.1. Những vấn đề về bộ nhớ và các giải pháp

Một giải pháp đối với vấn đề phân bổ bộ nhớ thì bao gồm việc làm gọn nhẹ bộ nhớ trống nhờ ghép làm đầy đủ các khoảng trống. Tuy nhiên, giải pháp này cũng rất tiện dụng và kéo dài thời gian sử dụng thực tế đối với một phần cứng.

Sự xử lý các mã chương trình:

Cho đến đây, vẫn còn một số vấn đề chưa được diễn giải một cách đầy đủ. Đó là vấn đề: *Định vị tuyệt đối*. Ở việc kết nối các phần của chương trình dịch, nhà tạo lập đã phân bổ cho các lệnh nhảy và các biến tham chiếu một địa chỉ nhớ tuyệt đối rõ ràng, mà ở đó, một địa chỉ cơ sở ở trong chương trình được đếm tăng dần, thí dụ bắt đầu từ giá trị 0. Nếu bây giờ người ta muốn sử dụng một chương trình ở một khoảng nhớ khác với các địa chỉ khác, khi địa chỉ đó đã được kết nối,thì do đó, các địa chỉ tham chiếu phải được xử lý trước đó. Về điều đó, dẫn tới những giải pháp khác nhau:

- Mã chương trình đã được tạo lập cần thiết phải chứa đựng chỉ các địa chỉ tương đối, thí dụ:

Địa chỉ = Địa chỉ tuyệt đối - Số đếm của chương trình máy tính PC.

Điều này thì không phải luôn luôn có thể đối với các kiểu vi xử lý và các lệnh, mà nó kéo dài địa chỉ số hoacj trong khoảng thời gian thực thi.

- Ở mỗi chương trình, sự trao đổi thông tin được lưu trữ trên ổ đĩa. Khi nạp ở bộ nhớ chính, mỗi địa chỉ tham chiếu phải được trù tính một lần và sau đó phải được ghi chép ở vị trí thích hợp. Điều đó được đòi hỏi cho tới khi không gian nhớ được tăng gấp đôi trên bộ nhớ quảng đại và đối với việc tráo đổi ở trong bộ nhớ, thì điều đó không thích hợp nữa. Ở các máy tính nhỏ, thì điều đó là thực thi.
- Ở một thanh ghi phần cứng chuyên dụng của CPU, sự trao đổi thông tin tồn tại với tư cách là những địa chỉ cơ sở và được sử dụng mỗi khi truy cập.

Để giải quyết vấn đề chi phí gia tăng, bây giờ đối với giải pháp của các vấn đề khác, nó phải được mở rộng trên những bình diện tiếp theo.

Che phủ bổ sung bộ nhớ:

Một thí dụ cho thấy, những yêu cầu của một tiến trình phải được giải bày như thế nào, nếu tiến trình này đã ở tại bộ nhớ chính hoặc nếu nó sử dụng bộ nhớ bổ sung (?). Cho điều này có những giải pháp như sau:

- Tiến trình được đặt tĩnh tại, dung lượng mới của nó được ghi chép và nó được nạp lưu trữ. Ở lần kế tiếp, nếu nó được sắp xếp vào hàng đợi, khi đó, người ta quan tâm ngay độ lớn mới này và tạo cho nó một không gian để thoả mãn các yêu cầu bổ sung. Chiến lược này được dùng trong hệ điều hành Unix với các ấn bản cũ.
- Các khoảng trống (phần cắt để bên ngoài) giữa các tiến trình ở trong bộ nhớ chính được cắt nhỏ cho mỗi tiến trình. Do đó, không gian trống được điền đầy từ dưới (với các địa chỉ nhỏ) nhờ việc mở rộng các ngăn xếp và lên trên nhờ việc mở rộng xấp(*heap*).

Bảo vệ bộ nhớ:

Vấn đề tiếp theo là ở chỗ, phần xác định của nhân hệ điều hành phải được lưu trữ bền vững trên không gian bộ nhớ hay các bộ đệm hệ thống. Phần này không những có thể bị phóng thích, mà còn có thể bị sử dụng một cách nhầm lẫn bởi các chương trình người sử dụng. Do đó, ở các hệ điều hành, các địa chỉ nhớ chuyên dụng (*fences, limits*) đã được quan tâm phòng ngừa, cấm không được vi phạm. Nếu ở bộ vi xử lý có một thanh ghi được lưu ý về điều đó, tức là, ở việc định vị, điều đó cũng được quan tâm, do đó các giới hạn không được cố định vào các địa chỉ logic.

Đối với các vấn đề đã được đề cập, dẫn tới những hiểu biết quan trọng: đối với người lập trình để thực hiện việc định vị trên máy tính; đối với các chương trình để đạt được sự chờ đợi thông suốt. Do vậy, tổng chi phí cho việc định vị ở trên máy tính thì cần phải được giảm thiểu một cách có lợi đáng kể cho một kiểu định vị trên bộ nhớ vừa đơn giản vừa rõ ràng, đó là kiểu định vị trên bộ nhớ ảo.

3.2.2. Bộ nhớ ảo

Hình ảnh mong muốn của người lập trình là bộ nhớ (*memory*), nó bắt đầu với địa chỉ không (zero) và tiếp tục cho đến vô tận. Nhưng đáng tiếc, trong thực tế thì không như vậy: hầu hết, hệ thống ngắt và hệ điều hành được bố trí vài phần dưới các khoảng dải của bộ nhớ. Các khoảng bộ nhớ này thì không liên tục, vì những chương trình khác đang tồn tại ở trong bộ nhớ, tức là, những chương trình này khi thì yêu cầu, khi thì trả lại không gian nhớ năng động này. Cuối cùng sự việc này còn có vai trò, rằng bộ nhớ chính thì giá thành cao, nhưng lại không thể đáp ứng việc dịch vụ đồng thời nhiều chương trình.

Từ lý do này, một bản phác thảo về bộ nhớ ảo được phát triển, ở đó, những mong muốn của người lập trình được ưu tiên như việc nhượng bộ mục đích cho hệ thống nhờ bộ vi xử lý phần cứng hay hệ điều hành. Trong hầu hết hệ điều hành ngày nay có hai khả năng dịch vụ, mà với việc quản lý bộ nhớ, nó phải được giữ vững bởi phần cứng và cả bởi hệ điều hành.

- Nhiều mảnh nhỏ (của khoảng dải bộ nhớ) phải được trình bày đối với chương trình, khi mà liệu chúng xuất phát từ một khoảng liên tục bắt đầu với giá trị không (?).
- Nếu chương trình yêu cầu nhiều bộ nhớ tồn tại, thì do đó, nó không phải là một chương trình đầy đủ, đặc biệt chúng được tráo đổi chỉ đối với khoảng dải nhớ không hoạt động ở trên bộ nhớ thứ cấp (bộ nhớ quảng đại, chẳng hạn ở đĩa từ tính) và sử dụng các dải bộ nhớ sẽ trở nên trống.

Hình 3.3 ở dưới cho thấy: bên trái là dải bộ nhớ ảo (được mong muốn), dải này phản ảnh sự quản lý bộ nhớ đối với chương trình, và bên phải là bộ nhớ vật lý thực.

Hình 3.3----------

Để tạo nên địa chỉ vật lý hay địa chỉ logic, nhiệm vụ được đặt ra là phải tiến hành và thực hiện chạy chương trình đối với mỗi tham chiếu bộ nhớ của một đơn vị phần cứng (thí dụ khối quản lý bộ nhớ MMU: *Memory Management Unit*). Trong nhiều bộ vi xử lý hiện đại, thí dụ các máy tính MC68040 của hảng MOTOROLA, các máy tính Pentium Procesor của hảng Intel, thì MMU được chứa đựng trên vĩ mạch vi xử lý (*processor- chip*).

Bây giờ có những cơ cấu nào cho hình ảnh mong muốn về bộ nhớ ảo trên bộ nhớ tồn tại thực (?). Điều đó được nghiên cứu trong mục tiếp theo dưới đây.

3.3. Quản lý trang

Một trong các cơ cấu đơn giản để thực thi không gian địa chỉ ảo thì bao gồm việc phân đoạn bộ nhớ thành *những đơn vị có độ lớn bằng nhau*: những đơn vị tách chia này *gọi là những trang* (*pages*). Các độ lớn trang tiện dụng thì khoảng 1kB, 4kB hay 8kB. Địa chỉ và trạng thái của mỗi trang được dẫn tới trong một bảng trang (*pape table*), bảng này tồn tại cho mỗi chương trình ở trong bộ nhớ chính.

3.3.1. Nguyên tắc hoán vị địa chỉ:

Để tính toán địa chỉ ảo (được sử dụng trong chương trình) ở trên địa chỉ vật lý thực của bộ nhớ chính, địa chỉ ảo được chia làm hai phần (xem hình 3.4). Phần chứa đựng các Bit ít ý nghĩa (*Least Significant Bits*: LSB) được gọi là độ dịch vị và nó cho thấy khoảng cách tương đối của địa chỉ hiện hành tới một địa chỉ cơ sở. Người ta nhận được giá trị của địa chỉ cơ sở, mà trong đó, phần thứ hai (bên trái hình 3.4) với các Bit có giá trị cao được sử dụng với tư cách một chỉ số (trong thí dụ là số 6), nó biểu thị cho việc ghi vào ở trong bảng kê các trang. Việc ghi vào này chứa đựng địa chỉ cơ sở (được tìm thấy). Người ta cũng nhận được địa chỉ vật lý đầy đủ qua sự kết nối của địa chỉ cơ sở với độ lệch offset.

Hình 3.4----------

Hình 3.4 chỉ cho thấy quá trình chuyển đổi này được mô tả thành hai giai đoạn, tại đó, đối với mỗi tiến trình, các bảng trang khác được sử dụng. Do đó, giới hạn giữa số trang PageNr và độ lệch offset thì ở trong khoảng các số nhị phân của địa chỉ ảo, nó phụ thuộc vào mỗi phần cứng được sử dụng.

Người ta lưu ý rằng, bộ nhớ (tồn tại thực) cũng có thể được phân thành các trang (gọi là sự phân đoạn bộ nhớ: *memory - partition*), mà ở đây, vị trí và độ lớn hữu hiệu của trang ảo nhờ cơ cấu định vị thì độc lập với sự phân chia đó. Khoảng phân đoạn bộ nhớ vật lý được biểu thi là khung trang (*pape frame*). Để thực hiện việc chuyển nhanh như có thể, hầu hết, địa chỉ định rõ tiến trình của bảng trang được giữ ở trong một thanh ghi chuyên dụng; tiến trình tổng hợp tồn tại ở trong khối điều khiển bộ nhớ MMU. Đối với việc nạp và điền thêm trang, hệ điều

hành có hiệu lực ở mức độ rất cao. Nếu Bit trạng thái của một trang chỉ ra rằng, trong không gian của bộ nhớ, đầu tiên trang phải được chuyển vào bộ nhớ quảng đại, do vậy khi đó, khối điều khiển bộ nhớ MMU phát sinh ra một tín hiệu lỗi trang (*pape fault*) có hình dạng của một ngắt chương trình (*program interrupt*). Ngắt này được hệ điều hành quan tâm, cụ thể: ở trong lập thức của ngắt (*interrupt-routine*), hệ điều hành chọn một trang ít được sử dụng, trang này được viết trở lại vào ổ đĩa. Cho điều đó, trang đã được dùng để viết vào và bảng được sửa chữa cho phù hợp. Tiếp theo, sau khi nhảy trở lại vào ngắt, lệnh máy được nhắc lại với sự hoán vị địa chỉ mà trước đó còn thiếu và sau đó chương trình được thực hiện tiếp tục.

3.3.2. Phương pháp dịch vị địa chỉ

Cho đến nay, chúng ta đã lưu ý đến trường hợp ,một địa chỉ bất kỳ của không gian địa chỉ ảo được phỏng theo một cách trực tiếp trên một địa chỉ giao dịch của không gian vật lý. Điều đó thì không phải luôn luôn có thể. Nếu chúng ta nhận thấy rằng, đối với mỗi trang, một sự điền vào bảng trang (*page-table*) thì rất cần thiết, do đó, nó dẫn tới một số lượng các địa chỉ được điền đầy như sau: giả sử nếu bề rộng từ là 16bit và độ lớn trang là 12 bit, thì do đó 16-12 =4, tức là ta có 2^4 =16 lần điền địa chỉ vào khác nhau. Một bảng như vậy thì thật dễ dàng để lưu trữ và điều khiển. Và do đó, ở các máy tính của bảng Digital, bảng này đã được phổ dụng một cách rộng rãi. Bấy giờ đối với từ có bề rộng 32 Bit (*word-wide*) thì điều đã trở nên vô cùng khó khăn, nếu chỉ lấy bề rộng từ 20Bit đã tương ứng với $2^{20} \sim 10^6$, tức là có tới một triệu lần điền địa chỉ. Ở cấu trúc mới nhất có bề rộng từ 64Bit, nếu chỉ lấy bề rộng từ 52 Bit thì đã phải cần tới $2^{52} \sim 4.10^{15}$ lần điền trang!

Ở đây vấn đề được nêu lên rằng, đối với không gian địa chỉ ảo, một sự biểu lộ về bộ nhớ vật lý cần thiết phải thực hiện; tuy nhiên, không phải hoàn toàn bộ nhớ nào cũng cho các địa chỉ ảo phần lớn nhất. Sự tiến thoái lưỡng nan này chỉ có thể được giải quyết trên các phương pháp khác nhau.

- *Việc giới hạn địa chỉ*:

Ý tưởng đầu tiên mà người ta đưa tới giải pháp cho vấn đề này là ở chỗ làm giới hạn không gian nhớ ảo trên một độ lớn có lợi nhất. Thí dụ, nếu chúng ta giới hạn không gian địa chỉ 30 Bit, thì một cái gì đó phù hợp với một bộ nhớ ảo khoảng 1GB trên một tiến trình, do đó, chúng ta cần dùng 4kB tương ứng 12 Bit cho sự phân đoạn trang, một trang tương ứng khoảng chừng 256 ngàn lần ghi vào cho một tiến trình, vậy một cái gì đó nằm trong khoảng có thể.

Tuy nhiên, độ lớn nhất còn phụ thuộc vào độ lớn thực của tiến trình. Đối với hầu hết các tiến trình, nó thì quá lớn; không gian của bảng không thể đem dùng: đối với một tiến trình có độ lớn khoảng 1GB thì một bảng trang có độ lớn 1MB có thể được đo đạc không đủ cho một tiến trình nhỏ với độ lớn 50kB.

Đối với sự biên dịch (*compiler*) thì việc giới hạn địa chỉ sẽ không có lợi. Đối với sự phát triển các ngăn xếp (*stacks*) thì có lợi để các ngăn xếp được mở rộng từ các địa chỉ rất cao tới các địa chỉ rất thấp và đề phòng ngừa một chỗ trống lớn giữa ngăn xếp và các địa chỉ thấp nhất được thiết lập.

- *Các bảng đa mức*:

Từ những lý do vừa nêu, có những điều kiện khác nhau được thực hiện. Một trong các ý tưởng quan trọng được thực hiện để giữ thông tin không bị chia cắt. Vấn đề được đặt ra là: liệu thông tin nói chung có sử dụng một khoảng bộ nhớ không (?), hay địa chỉ của thông tin có bị biến đổi bởi khoảng nhớ như thế nào đó không (?). Do đó, địa chỉ tổng hợp được phân đoạn thành nhiều phần như trong thí dụ dưới đây.

Thí dụ về phân đoạn địa chỉ:

Nếu ta có bề rộng của từ 32 Bit, độ lớn trang 8kB tương ứng 13 Bit thì sự phân đoạn được chỉ ra trên hình 3.5 ở dưới đây.

Hình 3.5----------

Bề rộng từ (*wordwide*) 32 Bit được phân thành 14 Bit cho chưa sử dụng, 3 Bit cho bảng 1(Tab1), 2 Bit cho bảng 2 (Tab2) và 13 Bit cho dịch vị offset. Nếu có một địa chỉ 41488_{10}~121024_{8} thì người ta phân thành một chỉ số Index =1 cho bảng 1, một chỉ số Index =1 cho bảng 2 và một độ dịch vị offset = 528.

Mỗi thành phần địa chỉ nhận được những bảng riêng lẻ, mà ở đó, bảng đối với các địa chỉ cao (*pape base table*) chỉ được lưu ý, liệu đối với địa chỉ ảo này bộ nhớ có tồn tại không (?), nếu có, thì bảng trang của việc che phủ đặt ở đâu (?). Ở trong hình 3.6, điều đó được chỉ ra một bảng hai bậc đối với địa chỉ được phân 2 đoạn, mà ở đó, sự tìm kiếm được ám chỉ bằng những bảng với các mũi tên đậm.

Mỗi một phần của địa chỉ tác dụng như là một chỉ số ở trong một bảng nào đó, mà chúng được ám chỉ bằng những mũi tên nét đứt đoạn; độ dịch vị offset là một chỉ số trên trang tồn tại thực ở bộ nhớ chính. Vì nếu mỗi địa chỉ có thể thuộc một trang, mà nó không ở trong bộ nhớ chính, thì do đó, địa chỉ của bảng đầu tiên cũng có thể thuộc một trang, mà nó không ở trong bộ nhớ chính, thì do đó, địa chỉ của bảng đầu tiên cũng có thể dẫn tới một trang lỗi. Trong trường hợp này, trang được nạp và được chứa đựng một bảng mong muốn, và sau đó, nó nhắc lại lệnh một cách mới mẻ, cho tới khi một trang ảo thực thụ được xác định, nó tồn tại ở trong bộ nhớ, địa chỉ vật lý được xác định, các tế bào nhớ được nhạy đáp.

Hình 2.6----------

Tương tự, một điều kiện như thế có thể tồn tại đối với loại bảng trang hai bậc (như ở các máy tính SPARC của hảng SUN) và đối với loại bảng 4 bậc (như ở các máy tính MC68038 của hảng MOTOROLA), mà ở đó, để tìm kiếm một địa chỉ, một sự trợ giúp của phần cứng kéo dài rất lâu, và do đó, việc thực hiện chương trình bị trì hoãn rất mạnh, thí dụ ở máy tính MC68030 khoảng 80%.

- *Bảng trang đảo ngược*:

Điều này dẫn tới những điều thực nghiệm khác nhau, để rút ngắn việc tìm kiếm lâu nhờ các bảng chưa được sử dụng. Con đường dẫn tới là, để thiết lập một bảng với tương đối ít các trang nhớ tồn tại thực của bộ nhớ vật lý. Đáng lẽ với tư cách là chìa khoá để viết tất cả địa chỉ ảo ở trên trang bên phải, do đó, người ta đã trao đổi bên phải thành trang bên trái và lập danh sách theo dãy tuần tự tăng dần từ bên trái với các trang đang tồn tại đến bên phải với các trang có địa chỉ ảo đã được sắp xếp, gọi là bảng trang đảo ngược (*inverse page table*). Để thay đổi

việc định vị với một bảng đảo ngược và rút ngắn bớt, người ta phải tìm kiếm tất cả các địa chỉ ảo ở bên phải, cho tới khi, người ta tìm thấy trang hiện hành phù hợp, và sau đó, đọc số trang vật lý ở bên trái. Hình 3.7 là một sự đảo ngược như thế đối với một thí dụ đơn giản của hai tiến trình với các bảng riêng lẻ. Chúng ta sẽ thấy thế nào, nếu địa chỉ ảo không đạt một mình, vì không gian địa chỉ thì như nhau thí dụ tại trang ảo 0 – 4 . Tuy nhiên, để có thể sắp xếp các địa chỉ ảo giống nhau một cách rõ ràng cho các trang khác nhau ở trong bộ nhớ chính, gọi là các khung trang thì chỉ số tiến trình phải lấy thêm số tiến trình.

- *Kho chứa các bảng liên kết*

Ở các bảng đa mức cũng như ở các bảng trang đảo ngược, vấn đề được đặt ra là: việc đánh giá thông tin các bảng để chuyển đổi địa chỉ thì tiêu tốn bao nhiêu thời gian (?). Từ lý do này, nó sẽ trở nên tiện dụng, rằng các việc sắp xếp cuối cùng từ các trang ảo sang các trang thực được lưu trữ trong bộ nhớ nhanh, còn gọi là bộ nhớ truy cập nhanh (*cache*), thí dụ ở máy tính MC68030. Đầu tiên bộ nhớ nhanh này được tìm kiếm trước khi các bảng được truy cập.

Bộ nhớ cache được tạo lập theo nguyên tắc truy cập định hướng nội dùng, khi đó cache còn được gọi là bộ nhớ liên kết; sau khi làm xuất hiện địa chỉ các bảng ảo thì một khoảng thời gian đã bị tiêu phí, liệu một sự chuyển chỗ các địa chỉ trang có được lưu trữ (?). Nếu có, thì địa chỉ vạt lý có ý nghĩa như thế nào (?). Hình 3.8 cho thấy điều đó bằng một sơ đồ. Các Bít của số trang thực được kê khai một cách tổng thể bằng các số thập phân.

Để đạt được sự đọc chọn bộ ký tự chỉ thị thời gian, một thiết bị điện tử được tích hợp vào nguồn pin của bộ nhớ (còn gọi là tế bào nhớ), để so sánh các Bits của địa chỉ ảo (ở đây: tiến trình Id=1 và trang ảo =0) với các giá của việc điền vào. Nếu ở việc điền vào, tất cả sự so sánh với các Bits yêu cầu là mĩ mãn, do đó, một cờ hiệu được đặt cho việc này (dấu X ở trong hình vẽ) và giá trị địa chỉ vật lý (dùng để giao dịch) được lựa chọn.

Hình 3.8----------

Một bộ nhớ liên kết kiểu như thế được biểu thị là bộ đệm dịch đổi khoảng nhìn. Bộ nhớ nhanh Cache góp phần quan trọng trong việc dịch địa chỉ. Ở một bộ Cache có dung lượng lớn, người ta cũng có thể loại bỏ các phần cứng dư thừa để chọn lọc các bảng trang. Thật vậy, người ta tiến hành việc dịch đổi địa chỉ bằng các phần mềm khi có ít biến cố, tức là khi đó việc dịch đổi không cần tới Cache. Tuy nhiên, đối với việc dịch địa chỉ theo cách đó cũng vẫn không giảm thiểu một cách mạnh mẽ hiệu dụng của bộ vi xử lý.

3.3.3. Bộ nhớ cùng nhau sử dụng (*shared memory*)

Một điều quan trọng nữa là việc cùng nhau sử dụng và điều khiển năng động các khoảng bộ nhớ nhờ các tiến trình. Điều đó đặc biệt có ý nghĩa: khi các mã được dùng trình biên soạn Text hay khi các chương trình của người sử dụng dùng nhiều thư viện (chẳng hạn thư viện của ngôn ngữ C). Kể cả các dữ liệu toàn cục cũng có ý nghĩa ở việc xác định các thông số cửa sổ của các tiến trình khác nhau ở trên màn hình.

Để có thể phản ảnh các khoảng của bộ nhớ vật lý xác định ở trong không gian địa chỉ ảo của nhiều tiến trình, nhiều biện pháp khác nhau được áp dụng. Biện pháp đầu tiên là, phải tạo

nên các gọi hệ thống để giải thích một khoảng bộ nhớ của một tiến trình mà nó đóng vai trò như một bộ nhớ cùng chia xẻ. Lúc đó, hệ điều hành dùng bộ định danh (*call-over*) để tham chiếu tới tất cả các tiến trình. Thêm vào đó, điều phải được đảm bảo rằng, các trang này thì không được dập bỏ, khi một trong các tiến trình kết thúc, tức là khi một trong các tiến trình dẫn các trang riêng lẻ vào trong không gian địa chỉ của nó. Đối với việc đồng bộ truy cập dữ liệu trên bộ nhớ Cache, các tác vụ cờ hiệu của hệ điều hành cũng cần phải được đặt sẵn sàng. Hình 3.9 chỉ ra một tình huống như thế cho ba tiến trình.

Hình 3.9----------

3.3.4. Bộ nhớ ảo ở Unix và Windows NT:

Bản phác thảo bộ nhớ ảo được thực thi một cách khác nhau ở trong các hệ điều hành Unix và Windows NT và được thany đổi từ ấn bản này tới ấn bản khác.

Không gian địa chỉ ở Unix:

Ở trong hệ điều hành Unix, không gian địa chỉ ảo là 32 Bit tương ứng với dung lượng 4 GB

Ngoài ra, để đặc trưng cho không gian địa chỉ ảo của một tiến trình có một không gian các thanh ghi (*register- space*) với chiều dài 16 Bit hay 32 Bit. Do vậy, địa chỉ 32 Bit được mở rộng thành các địa chỉ dai 48 Bit cũng như 64 Bit, mà ở đây, không gian các thanh ghi có thể được quan niệm là bộ chỉ thị (*pointer*) cho một trong các không gian địa chỉ ảo với độ lớn 2^{16}.

Tổng không gian địa chỉ ảo khoảng 4GB được phân bổ thành 4 khoảng riêng lẻ, gọi là các cung phần tư (*quadrant*), tuỳ theo loại máy tính, các cung phần tư này có ý nghĩa riêng. Trong các khoảng chia này còn tồn tại những cung nhỏ hơn, gọi là các cung logic xác định (*segment*) hay các khu vực tiến trình (*process regions*), mà các segment này được điều hành để đặt lên những địa chỉ ảo cố định. Đối với mỗi một segment cũng tồn tại một sự điền vào, mà nó dẫn tới những thông tin về quy luật truy cập (đọc / viết) và số lượng các trang. Đối với mỗi thông tin riêng lẻ của các trang với độ lớn 4 kB thì có một sự điền vào ở một cấu trúc dữ liệu trong nhân của hệ điều hành: liệu trang đã hợp thể thức chưa, trang ở trong trạng thái nào và liệu trang có được ghi chép trở lại không (?)…Tuy nhiên, việc sắp xếp các segment ở trong không gian địa chỉ ảo của người sử dụng thì ở file dữ liệu được mô tả /usr/include/sys/VA s.h và nó được chỉ ra trong hình 3.10 ở dưới đây.

Hình 3.10----------

Với các điều đã được trình bày ở trên, những segment của tiến trình người sử dụng được sắp xếp theo các địa chỉ tăng dần như sau:

- *Cung phần tư thứ I:* Với địa chỉ 0 thì các đoạn mã (*codesegment*) như mã chương trình của người sử dụng được nạp. Những mã này có thể được tạo lập một cách đồng đệ trên địa chỉ vật lý, do đó, việc cùng nhau sử dụng mã chương trình với các tiến trình khác nhau, thí dụ trình Editor, là có điều kiện.

- *Cung phần tư thứ II:*Cung này chứa các đoạn dữ liệu (*data-segment*) của các dữ liệu đã được bắt đầu cũng như chưa được bắt đầu. Tại đây, các heap- segment (các đoạn xấp) có thể được phát triển nhờ các dịch vụ hệ thống cbrk() và malloc(). Ngoài ra, cũng tại đây (user area), ngữ cảnh của người sử dụng được ghép vào với ngăn xếp nhân và ngăn xếp người sử dụng, vì vậy ngăn xếp có thể phát triển một cách năng động ở trạng thái người sử dụng trong sự khác biệt với ngăn xếp nhân.
- *Cung phần tư thứ III:* Ở đây tồn tại các địa chỉ, mà nó được tham chiếu tới các khoảng cùng nhau sử dụng của các thư viện với các segment mã và segment dữ liệu cũng như khoảng các files được tạo lập trực tiếp (*memory mapped files*).
- *Cung phần tư thứ IV:* Tại đây tồn tại các địa chỉ, mà nó chứa đựng các khoảng bộ nhớ với các tiến trình khác nhau. Ở trong Unix, đối với mỗi segment của bộ nhớ shared memory có một cơ cấu gọi, mà một segment của các tiến trình khác nhau có thể được tham chiếu tới cơ cấu này. Các gọi hệ thống *plock()* và *shmct1()* cho phép giữ cố định các trang ở trong bộ nhớ và cho phép loại bỏ một sự phí tổn. Điều đó hạn chế được khoảng 75% bộ nhớ trống.

Ở khoảng địa chỉ cao tồn tại một khoảng, mà nó được phòng ngừa cho việc truy cập đặc biệt nhanh trên các thiết bị vào ra (I/O map); các bộ đệm và các thanh ghi của nó có thể được chọn hay được mô tả dưới một tham chiếu địa chỉ, còn gọi là các thiết bị ánh xạ bộ nhớ (*memory mapped devices*). Tất nhiên, tiến trình người sử dụng thông thường không thực hiện điều đó với các quy tắc truy cập thông thường, đặc biệt nó được thực hiện dưới sự điều khiển của trạng thái nhân hệ điều hành.

Không gian địa chỉ ảo Windows NT:

Hệ điều hành Windows NT được dự định cho một không gian địa chỉ ảo 64 Bit; tuy nhiên, không gian này thường được dùng 32 Bit ứng với độ lớn 4 GB. Độ lớn này được phân thành 2 phần: 2GB cho không gian địa chỉ đối với tiến trình người sử dụng, và 2GB cho không gian của các chức năng còn lại. Sự phân chia này được chỉ ra trong hình 3.11 ở dưới.

Nửa trên của 2GB để lưu trữ nhân hệ thống và các bảng hệ thống, mà những thứ này có đặc điểm nổi bậc là thời gian truy cập rất ngắn. Vì tất cả các trang của nhân hệ thống luôn luôn tồn tại ở trong bộ nhớ chính và vì đối với hệ điều hành thì không dùng cơ cấu bảo vệ ở việc truy cập, cho nên, việc truy cập chỉ đạt được trên địa chỉ nhân. Ở trạng thái nhân, thì 2 Bit cao nhất của địa chỉ bị nén lại và phần còn lại của địa chỉ ảo được thông dịch thành các địa chỉ vật lý. Việc thực thi của hệ điều hành Windows NT phải được khẳng định rằng, nó rất hiệu quả.

Ngược lại, cấu trúc trang hay việc quản lý trang làm cho phần bộ nhớ còn lại lâm vào tình trạng sút kém, lúc đó, việc quản lý trang được thực hiện bởi sự điều hành bộ nhớ ảo (*virtual memory manager*). Ngoài ra, điều đó được trợ giúp bởi các trang (được định nghĩa bởi phần cứng) có độ lớn 4kB tới 64kB, bình thường thì dùng 4 kB.

Hình 3.11----------

Việc quản lý bằng bộ nhớ ảo được dự định là một bảng có 2 bậc. Bậc đầu tiên chứa đựng một thư mục trang, mà thư mục này chứa đựng địa chỉ của các bảng có 2 bậc. Nếu ở việc truy cập địa chỉ xảy ra một lỗi trang, do đó, trang được nạp, mà các bảng còn lỗi thì được chứa

đựng trong trang này. Ở bảng thứ 2 thì chứa đựng số trang vật lý (pape frame) và kết nối với các thông tin tiếp theo như các quy tắc truy cập…

Đối với bộ nhớ cùng nhau sử dụng (shared memory) có một quy tắc đặc biệt. Đối với các số trang vật lý, bảng thứ 2 chứa đựng một chỉ số của một bảng đặc biệt, gọi là bảng trang nguyên mẫu (*prototype page table*). Trong bảng này chứa đựng địa chỉ của các trang, mà chúng giải thích cho việc sử dụng cùng nhau. Sự khái quát ở trong sơ đồ định vị yêu cầu việc truy cập lần đầu (còn sự tham chiếu trực tiếp vào bộ nhớ Cache hợp với những lần truy cập tiếp theo) cho phép có một tổ chức đơn giản của trang. Nếu một trang được giữ trở lại ở bộ nhớ quảng đại sau khi nạp, và trước đó, được đặt một vị trí khác ở trong bộ nhớ chính, thì do đó, hệ điều hành xem xét tất cả các bảng của các tiến trình và thay đổi địa chỉ vật lý một cách phù hợp, nếu tìm thấy. Điều này loại bỏ được sự điều hành tập trung. Hệ điều hành có thể bị giới hạn trên tại các bảng trang nguyên mẫu.

Các khoảng bộ nhớ có thể được một tiến trình là sáng tỏ việc cùng nhau sử dụng bởi các tiến trình khác nhau. Do đó, một đối tượng đoạn (*section object*) được tạo ra để một file đặc biệt khảo sát và để dẫn ra các tính chất sau đây:

- *Thuộc tính đối tượng (object attribute)*: Đó là độ lớn tối đa, bảo vệ trang, file ảnh xạ hay file đánh số trang: Yes/No, địa chỉ bắt đầu giống nhau tại tất cả các tiến trình: Yes/No.
- *Các phương pháp đối tượng (objectmethode)*: Đó là các phương pháp sản sinh, mở, nới rộng, chọn mặt cắt, xác định trạng thái…

Nếu các tiến khác nhau muốn sử dụng một khoảng dữ liệu đã được làm rõ, do đó, chúng phải mở đối tượng section (đoạn) và lựa chọn một mặt cắt. Mặt cắt này chỉ cho thấy ở trong không gian địa chỉ ảo của chúng, liệu nó có phải là một mặt cắt bộ nhớ bình thường hay không (?).

Để quản lý các trang vật lý (*page frames*), nhân của hệ điều hành Windows NT có một cấu trúc dữ liệu đặc biệt, đó là cơ sở dữ liệu trang vật lý (*page frame database*). Đối với mỗi trang tồn tại, chúng chứa đựng một sự điền vào, nghĩa là, mỗi số trang ở trong các bảng của bậc thứ 2 thì phù hợp với một chỉ số ở trong cơ sở dữ liệu khung trang và ngược lại. Trong sự khác biệt với các Bit trạng thái của một trang ảo, ở trong cơ sở dữ liệu khung trang tồn tại những thông tin trạng thái của các trang thực, mà chúng chứa đựng một trong các trạng thái sau:

valid	là trạng thái hợp lệ, nó tồn tại một sự điền vào thuận lợi các bảng trang;
zeroed	là trạng thái tự do, nó được bắt đầu với giá trị 0;
free	là trạng thái trống, nhưng không có giá trị bắt đầu;
standby	là trạng thái chuyển tiếp trang thuộc tiến trình chính thức không nhiều nhưng có thể được giữ trở lại;
modified	được mô tả giống như trạng thái standby;
bab	là trạng thái chứa đựng lỗi vật lý, không thể sử dụng.

Bằng cơ cấu gọi ở mỗi lần điền vào, những lần điền vào như nhau được kết nối với nhau thành một danh sách. Bên cạnh những trang hợp lệ (valid), còn có năm danh sách còn lại. Cơ sở dữ liệu khung trang được các tiến trình sử dụng, và do vậy, đối với hệ thống đa vi xử lý, nó được đảm bảo bởi một cờ hiệu spinlock.

Tuy vậy, vì chỉ một cờ hiệu cho cấu trúc dữ liệu được sử dụng, do đó, ở tần số đánh số trang cao, thì cơ sở dữ liệu khung trang có thể thu nhỏ hiệu suất của hệ thống. Một ấn bản song hành với nhiều mặt cắt thì ở việc điều hành song song sẽ nhanh hơn ở việc điều hành đơn điệu, tuy nhiên, nó vẫn chưa thực thi.

3.3.5. Chiến lược thay thế trang:

Sau khi dịch địa chỉ, nếu dẫn tới một trang cần dùng bị thiếu, do đó, chúng phải được bộ nhớ quảng đại đọc và được nạp trong bộ nhớ chính vật lý. Bởi vậy, câu hỏi đặt ra là: tại chỗ nào và trang tồn tại nào được viết chồng lên (?).

Nếu chúng ta cần thay tới một trang bình thường hay được dùng, thì chúng ta phải đòi trang này đến ngay, và như vậy, thời gian chạy chương trình sẽ gia tăng. Vì vậy, nhiệm vụ của một chiến lược tốt phải là: Việc tìm trang để có thể thay thế nó, *thì người ta không cần để ý tới vấn đề này*. Người ta có thể tìm thấy một chiến lược thích hợp như thế, đó là chiến lược định thời đối với một bộ vi xử lý trao đổi trang, vì bộ vi xử lý này chứa đựng những yêu cầu đối với các trang cần dùng ở trong hàng đợi. Điều đó cho thấy, chiến lược này thì tốt hơn để thoả mãn nhanh hơn những yêu cầu về trang. Do đó, công việc đến với chúng ta là, hầu hết các tham chiếu chỉ xảy ra một cách cục bộ ở trong một chương trình, do đó, đối với các trang bất kỳ thì không có kết quả.

Dãy các trang tham chiếu cần dùng còn được gọi tắt là dãy tham chiếu (*reference string*), có còn đặc trưng cho con đường của dòng dữ liệu và dòng điều khiển trang khi thực hiện chương trình. Nếu chúng ta mong muốn dãy tham chiếu này ở trong chương trình, do đó, chúng ta có thể điều chỉnh việc thay thế trang. Đó là cơ sở của chiến lược tối ưu để thay đổi trang.

- *Chiến lược tối ưu*:

L.A.Belady đã chỉ ra (1966) rằng, những sự thay thế ít nhất được đón nhận khi người ta chọn trang, mà muộn nhất, nó sẽ được dùng sau này. Ở hiình3.12 chỉ ra dãy tham chiếu 0,1,2,4,0,1,5,6,0,1,2,3,4… cho một bộ nhớ chính chỉ đối với 3 trang có điều kiện.

Hình 3.12----------

Do đó, bảng được đọc như sau: Mỗi một cột chỉ một trạng thái của hệ thống tại một thời điểm, mà ở đó, chỉ số tại thời điểm của một cột được tăng lên từ trái sang phải ở trong bảng. Mỗi cột được chia làm 2 phần: Phần trên là số trang của các trang nằm ở trong bộ nhớ chính (RAM), còn phần dưới là số trang của các trang được nạp trên bộ nhớ quảng đại (ổ đĩa cứng và ổ đĩa mềm). Trang cần dùng được ghi vào hang đầu tiên của khoảng bộ nhớ RAM; còn trang cần thay đổi được ghi vào hàng đầu tiên của khoảng bộ nhớ quảng đại. Nếu một trang dời chỗ lên hang thứ nhất của mỗi dãy, do đó, tất cả các số trang khác được di dịch xuống phía dưới còn gọi là cơ chế ngăn xếp (*stack mechanicmus*) cho đến khi các chỗ trống được điền đầy ở trong RAM. Những trang mới thay đổi được viết khoanh tròn như ở tron hình 3.12 ở trên.

Chiến lược tối ưu này (*optimal strategy*) chỉ được sử dụng ở những chương trình xác định, mà đối với các chương trình này thì các yêu cầu trang đã rõ. Tuy nhiên, những chương trình

như thế không có trong thực tế, cho nên, chiến lược chỉ được sử dụnh với tư cách là tham chiếu để so sánh với các chiến lược khác ở khả năng hiệu dụng của chúng.

Với lý do trên, các phép thống kê thời gian (được chi tiết hoá nhiều hơn hoặc ít hơn) được lập nên qua các trang cần dùng và được dùng làm cơ sở cho việc thay thế trang trên cơ sở đánh giá của việc thống kê. Một trong các phép đánh giá đơn giản được trợ giúp bởi các Bits trạng thái, mà chúng tồn tại bên cạnh thông tin địa chỉ với việc điền vào cho các bảng trang. Người ta ký hiệu: với R (*referenced*) là Bit trang được tham chiếu hay được sử dụng, còn M (*modified*) là Bit trang được điều chỉnh, tức là trang được thay đổi và được viết trở lại.

Nếu trong một khoảng thời gian đều đặn, Bit R được đặt lui nhờ bộ đếm thời gian (tức là vượt lên ngăn xếp thời gian), do đó, R=1 chỉ ra rằng, trang phù hợp được sử dụng trong chốc lát (nhưng chưa vượt lên ngăn xếp thời gian), và vì vậy, nó thì chưa cần thiết phải thay thế. Nếu trang được tham chiếu, do đó, bộ đếm thời gian được đặt trở lại giá trị không. Vì thế, đối với mỗi trang, một bộ đếm được sử dụng. Phương pháp này thì hơi đắt, do đó, người ta có thể áp dụng cách làm gần đúng là dùng một bộ đếm cho tất cả các Bits này với mỗi chu kỳ mộ lần. Vì không có khoảng thời gian tuyệt đối, mà chỉ có sự khác nhau của trang trong một khoảng là cần thiết, do vậy, việc đặt lại các Bits R ở tại các biến cố đặc biệt, thí dụ có thể dẫn tới tại một lỗi trang.

Việc sử dụng thông tin cần dùng cho các trang dẫn chúng ta tới những chiến lược sau đây để thay đổi trang.

- *Chiến lược FIFO*:

Những trang mới tới được sắp xếp vào trong một dãy tuần tự của một bảng phù hợp cho sau này của chúng. Nếu chúng ta chọn một trang ở đầu danh sách đại diện cho vieecej thay thế, do đó, chúng ta có trang cũ nhất. Nếu chúng ta phỏng đoán rằng, sự thay thế chúng được chứa đựng chương trình chính. Cho nên, đối với trang cũ nhất thì còn phải lưu ý thêm trạng thái của các Bits R. Nếu R =1, thì trang còn được dùng. Ở sự biến đổi của chiến lược FIFO thuần khiết, người ta có thể đặt trang này trở lại cuối danh sách và đặt R=0. Và lúc đó, khi trang vừa mới tới, người ta nói rằng: Trang nhận được một dịp may thứ 2, tức là khi đó trang tuân theo thuật toán dịp may thứ 2 (*second chance algorithmus*). Thật vậy, nếu R=0 thì trang thay thế, còn khi M=1 thì trang được viết trở lại trước đó.

Người ta có thể đơn giản hoá phương pháp này, mà ở đó, người ta kết nối danh sách theo một vòng xích. Vì chỉ một số lượng trang được xác định tối đa chiếm không gian bộ nhớ chính, do vậy độ lớn của vòng xích nói trên không thay đổi. Chỉ có sự đánh dấu trang cuối cùng được thay đổi mỗi khi thay đổi trang và tiếp diễn sự điền vào ở vòng xích với các trang điền vào, như là một bộ chỉ thị giờ, còn gọi là cơ cấu giờ (*clock algorithmus*). Hình 3.13 chỉ ra chiến lược FIFO đơn giản ở thí dụ mà chúng ta nêu ở trên.

Hình 3.13----------

Trong sự so sánh với chiến lược tối ưu ở trong hình 3.13 ở đây (hình 3.13) có 2 sự thay thế trang được dùng để làm đầy trang ở trong RAM.

- *Chiến lược NRU*:

Chiến lược chỉ sử dụng một ít việc thống kê trang hay hoàn toàn không sử dụng gì cả. Do đó, người ta muốn điều đó được làm một cách tốt hơn nhờ các công cụ đơn giản. Với sự trợ giúp của hai loại Bits R và M, các trang được chia làm 4 cấp.

1) R=0, M=0
2) R=0, M=1
3) R=1, M= 0
4) R=1, M=1

Điều rõ ràng là,các trang thuộc cấp 1 với R=0 và M=0 thì ít được sử dụng nhất, và do đó, chúng được thay thế đầu tiên (trước khi các trang được điều chỉnh nhưng không được tham chiếu tới cấp 2), mà chúng có thể còn được sử dụng lại. Quan trọng hơn, đó là các trang cấp 3, tức là các trang được tham chiếu thực thụ; cũng như thế, khi R=1 và M=1, các trang này được mô tả là trang cấp 4.

Do đó, một sự quan trọng hay một ưu tiên nào đó đối với trang sẽ khẳng định: Các trang có số cấp nhỏ nhất được thay thế đầu tiên. Cách thức này được gọi là chiến lược NRU, tức là chiến lược mới tới nơi không sử dụng (*Not Recently Used*-NRU). Hình 3.14 chỉ ra một thí dụ về chiến lược NRU.

Hình 3.14----------

Số lượng các thay thế thì đơn giản trong chiến lược FIFO, các trang thì khác biệt nhau ở trong RAM, ngay khi đó, thông tin được sử dụng qua sự tham chiếu trước đây.

Trong hình 3.14, việc thay thế xảy ra từ yêu cầu của trang 6: Đáng lẽ trang 0 choáng chỗ, thì trang 5 được sử dụng ít nhất lại choáng chỗ, do đó, ở các yêu cầu tiếp theo, các trang được choáng chỗ nhiều hơn, trừ các trang 0 và 1.

- *Chiến lược LRU*:

Một cách chính xác hơn là một danh sách FIFO hay một sự tồn tại định lượng thời gian thì nó đã mang một trang tới bộ nhớ một cách không cần thiết. Do đó, tại nhiều trang với R=0 thì trang già nhất được xác định và thay thế, gọi là *chiến lược gần đó là ít nhất được sử dụng* (*Least Recently Used*: LRU).

Một giải pháp phần cứng có thể được mô tả: đó là một bộ đếm chạy nhanh, thí dụ bộ đếm thời gian để chỉ giờ và chỉ ngày tháng năm. Tại mỗi tiếng tích- tắc thì một bậc thời gian được đảm nhận một cách tự động bởi việc điền vào bảng của trang đang hoạt động. Tất cả các trang không hoạt động thì duy trì một trạng thái cũ. Bấy giờ, nếu trang cũ nhất được tìm thấy, do đó, số thời gian nhỏ nhất được tìm thấy ở trong việc điền vào các bảng.

Nếu không có phần cứng nào được sử dụng, do đó, người ta có thể mô phỏng trang cũ một cách áng chừng nhờ một thanh ghi di dịch trên từng trang. Do đó, Bit R của mỗi trang với tư cách là Bit cao nhất của thanh ghi được đặt vào những khoảng thời gian đều đặn và toàn bộ sức chứa của thanh ghi được đặt vào những khoảng thời gian đều đặn và toàn bộ sức chứa của thanh ghi được dịch chuyển một Bit sang phải (*shift right*). Hình 3.15 là một thí dụ cho sự di dịch sang phải của 3 trang. Thanh ghi di dịch tác dụng như là một cửa sổ thời gian đối với sự hoạt động của trang, thí dụ với thanh ghi 8 Bit thì nó cách nhau 8 phiên thời gian. Sự di dịch sang phải làm ảnh hưởng thông tin các trang cũ. Nếu người ta quan niệm trạng thái của thanh ghi di dịch như là một con số, do đó, giá trị con số này là lớn nhất, giá trị này cũng đã ghi lại

tất cả sự hoạt động trong thời gian cuối cùng, giá trị của sự hoạt động này cũng giảm dần theo khoảng thời gian gia tăng.

Hình 3.15----------

Trang để thay thế theo chiến lược LRU chỉ ra con số nhỏ nhất ở trong thanh ghi di dịch. Hình 3.16 chỉ ra một thí dụ về chiến lược LRU trình bày ở trên.

Hình 3.16----------

Tuy nhiên, sơ đồ trong hình 3.16 chưa phản ảnh một cách đầy đủ cơ sở quyết định của vấn đề: Khi tham chiếu trang thì nó chưa rõ ràng là một sự kiện thông thường của việc thay thế trang, do đó, sau đây nghiên cứu tiếp theo chiến lược NRU.

- *Chiến lược NFU (Not Frequently Used)*:

Để thay thế trang, một chiến lược tiếp theo cho thấy, nó thường hay được sử dụng nhất. Ở đây nó không đo đạt từng thời điểm như ở chiến lược LRU, tại thời điểm đó, trang không nằm trong bộ nhớ, mà đặc biệt trang được dùng trong khoảnh khắc. Thật vậy, đối với mỗi trang, một bộ đếm được dẫn ra, mà nó được gia tăng một cách chu kỳ khi sử dụng (r=1). Sau đó, trang được đặt với giá trị nhỏ nhất.

Đối với chiến lược này, vấn đề là ở chỗ, các trang được sử dụng mạnh mẽ trước đây, bây giờ rất khó khăn choáng chỗ ở trong bộ nhớ chính, vì giá trị con số của chúng thì rất cao. Do đó, điều có ý nghĩa là, người ta dự định dùng thêm một cơ cấu làm già để phòng tránh biến cố (*olding mechanismus*).

Việc áp dụng các chiến lược khác nhau cũng rất khác nhau. Việc lựa chọn chiến lược tốt nhất và việc thiết kế theo của kỹ thuật viên sẽ trở nên hoàn hảo, nếu chúng ta nhìn nhận một cách sâu sắc hơn về các cơ cấu thay thế.

3.3.6. Mô hình hoá và việc phân tích thay thế trang

Trong mục này, chúng ta sẽ khảo sát thực nghiệm, để từ việc nghiên cứu tìm thấy mô hình đối với các chiến lược và đặc điểm của các thuật toán. Do đó, đầu tiên, chúng ta lưu ý đến câu hỏi: Thực ra một trang cần thiết phải lớn bao nhiêu (?). Sau đây những chi tiết được trình bày để giải quyết câu hỏi vừa nêu.

Chiều dài trang tối ưu:

Nếu K là dùng lượng (size) của bộ nhớ chính và s là kích cở của một trang. Ở đây, s không phải nói về kích cở phần cứng của trang, mà đặc biệt, s biểu thị kích cở hệ điều hành của một trang; vì thế, nó được áp dụng nhiều trang phần cứng (*hardware pages*) cho một trang phần mềm (*software pages*).

Chúng ta nhận thấy rằng:

+ Chiều dài của dữ liệu được phân chia bằng nhau một cách ngẫn nhiên, do đó, tất cả các giá trị của từng lát cắt (mỗi đoạn được phân chia) nằm trong khoảng [0,s]. Thông thường, lát cắt trung bình là s/2 đơn vị bộ nhớ, thí dụ các từ (word) trên một tiến trình.

+ Mỗi bảng trang bậc 1 biểu thị sự điền vào trên mỗi tiến trình là [K/s], mà ở đó, mỗi sự điền vào yêu cầu một đơn vị bộ nhớ (thí dụ một word).

Do đó, sự tổn thất (V) của các đơn vị bộ nhớ xuất hiện trên mỗi tiến trình với hệ số tổn thất f_v được biểu thị bằng biểu thức sau:

$$V = (K/s + s/2) \sim K f_v$$

Ở các trang lớn thì một mặt các bảng trang trở nên nhỏ hơn, nhưng mặt khác các lát cắt lại trở nên lớn hơn. Ngược lại, ở các trang nhỏ thì lát cắt trở nên nhỏ hơn, nhưng độ lớn của bảng lại tăng lên. Vì vậy, giữa hai thái cực, có một sự tối thiểu cục bộ. Khi s-> 0 thì chúng ta sẽ nhận được một sự tối thiểu của tổn thất, tức là xảy ra điều kiện sau đây:

Công thức ----------

với $f_v = 2/s_{opt}$

Trong đó, K và s có thứ nguyên [kByte], f_v có thứ nguyên [%].

Thí dụ: Nếu không = 5000kB, thì do đó ta xác định được s_{opt} = 100kB với hệ số tổn thất bộ nhớ chính là f_v=2%.

Độ lớn của trang được dùng thực chất ở trong hệ điều hành còn tạo ra nhiều tiêu chuẩn tiếp theo. Các trang lớn rất có ý nghĩa đối với lát cắt thời gian, nó sẽ không có lợi vì những đòi hỏi của bộ nhớ bổ sung bị thu hẹp.Nghĩa là, khi lát cắt thời gian lớn thì sẽ không thoả mãn với mô hình của một yêu cầu bộ nhớ được phân chia đồng đều trên mỗi trang. Từ lý do vừa nêu, các độ lớn trang thường được phân chia nhỏ hơn.

Đối với độ lớn trang còn có nhiều nhân tố khác, đó là thời gian được dùng để nạp một trang trên bộ nhớ quảng đại và cũng như lát cắt của bộ nhớ quảng đại ở một độ lớn file trung bình. Nếu giả sử các files có độ lớn chừng 1kB, do đó, đối với một sự áp dụng hữu hiệu của các bộ nhớ quảng đại, độ lớn trang được sử dụng phải nằm trong khoảng đó. Nếu tại một độ lớn file trung bình 1kB, chúng ta muốn khẳng định một độ lớn trang khoảng 100kB, khi đó, trung bình đối với mỗi file có khoảng 99kB chưa được sử dụng. Đó là một khả năng chịu tải quá kém.

Để đáp ứng được cả hai yêu cầu vừa có đơn vị bộ nhớ nhỏ và vừa có số lượng chuyển đổi trang lớn, có rất nhiều hệ điều hành đã thử nghiệm để quan tâm tới: nếu file có nhiều trang thì độ lớn của trang phải nhỏ.

Số lượng trang tối ưu:

Một công việc quan trọng tiếp theo lôi cuốn các nhà quản lý hệ thống, đó là câu trả lời cho câu hỏi: Nhiều tiến trình cần thiết phải được nạp vào bộ nhớ chính như thế nào (?). Điều đó cũng phù hợp với câu hỏi: Nhiều trang trên một tiến trình phải được đặt ở chỗ nào ở trong bộ nhớ chính (?).

Bây giờ chúng ta khảo sát một thí dụ. Chúng ta so sánh thuật toán FIFO đối với một hệ thống có 4 trang RAM (bảng phía trái) và một hệ thống có 5 trang RAM (bảng phía phải).

Hình 3.17----------

Chúng ta nhìn thấy một nguyên cớ ngẫu nhiên: Ở hệ thống có 4 trang RAM thì chỉ có 7 lần thay thế được sử dụng, còn ở hệ thống 5 trang RAM có 8 lần thay thế trang được sử dụng, vậy mà bộ nhớ chính còn có thể được dùng nhiều hơn. Nguyên cớ này làm xuất hiện một thuật toán mang tên nhà phát minh L.A.Balady, mà nó có tên gọi là *thuật toán các kỳ dị Balady*. Ngược lại chúng ta khảo sát thuật toán LFU ở trong hình 3.18 ở dưới đây.

Hình 3.18----------

Việc sử dụng bổ sung bộ nhớ nhờ một trang RAM vẫn không làm thay đổi các trang (khác với thuật toán FIFO) nằm ở trong bộ nhớ chính. Điều đó thì cũng logic: Ở chiến lược LFU, các trang thường được dùng thì luôn luôn đứng ở phía trên trong danh sách ưu tiên, luôn luôn độc lập với việc khi chúng được nạp hay khi chúng ở trang RAM, và đồng thời, chúng cũng độc lập với giới hạn giữa RAM và các ổ đĩa (cứng và mềm). Danh sách ưu tiên của chiến lược LFU được thực thi với sự trợ giúp của một cơ cấu ngăn xếp: Các trang có ưu tiên cao đứng hoàn toàn ở trên và các trang khác di dịch xuống phía dưới. Trang nào được di dịch ra khỏi giới hạn của RAM và các ổ đĩa thì được nạp trên bộ nhớ quảng đại. Các kiểu thuật toán tác dụng lên m trang ở trong bộ nhớ thì độc lập với việc chúng trả lại tham chiếu với m hay m+1 trang RAM. Chúng có tên là các giải thuật ngăn xếp (*stack- algorithsmen*). Điều đó chỉ ra rằng, giải thuật các kỳ dị Balady không thuộc các thuật toán ngăn xếp vừa nêu. Tập hợp các trang được tham chiếu $W(t,\Delta t)$ thì rất quan trọng (ở đây: t là một thời điểm nào đó, Δt là nhịp thời gian), nó còn chứa đựng thêm những trang đặc biệt hay được dùng ở phía dưới. Nếu không có đúng, tiến trình sẽ làm việc kém hiệu quả; do đó, $W(t,\Delta t)$ được biểu thị là *tập hợp công tác* (*working set*). Tập hợp trung bình *working set* $(W(t,\Delta t))_t$ đặc trưng cho diễn biến của một tiến trình. Người ta lưu ý rằng, định nghĩa này thì khác xa với định nghĩa gốc do P.J. Denning nêu ra (1980), mà với điều đó, người ta xác định số lượng tối thiểu các trang, mà đối với việc thực hiện một tiến trình thì các trang này rất cần thiết.

Thí dụ về Tập hợp công tác của P.J. Denning:

Nếu ở một chương trình chúng ta có các lệnh này:

```
MOVE A,B
MOVE C,D
```

Trong đó: A,B,C,D là các biến ở trong các trang khác nhau, do đó, tiến trình chỉ có thể làm việc, nếu bên cạnh các trang mã (*codepages*) tiến trình còn có 4 trang tham chiếu địa chỉ được sử dụng. Khi tiến trình cần dùng tối thiểu 5 trang, do đó theo P.J.Denning, tạp hợp công tác có giá trị w=5 trang, nó thì độc lập khi các trang tiếp theo được sử dụng.

Trong các bảng nêu ở trên, việc điền đầy các cột đầu tiên (để sử dụng các trang RAM) thì không chỉ ra một cái gì đặc biệt, vì ở đây, không có trang nào được thay thế. Tuy nhiên, điều đó chỉ có thể được xảy ra với sự trợ giúp của ngắt lỗi trang (*page fault interrupt*), nếu không còn cái gì khác thì được hệ điều hành đảm nhận phòng ngừa. Việc nạp vào hay việc thay thế các trang được biểu thị là yêu cầu thiết đặt trang (*demand paging*). Ngược lại, người ta có thể

nạp một cách ngẫu nhiên các trang của một tập hợp working set hay nạp các trang của một tiến trình đang ngủ trước khi tiến trình hoạt động trở lại (*preparing*).

Phân tích hiệu ứng thrashing (*thrashing effect*) **và tập công tác** (*working set*)

Nếu k là số trang tồn tại của bộ nhớ chính và m là số trang có điều kiện của tiến trình và nếu thoả mãn điều kiện m<k, do đó, tiến trình sẽ không bị làm chậm lại nhờ việc trao đổi trang. Và nếu khoảng thời gian xử lý của tiến trình có giá trị B_1, bấy giờ, chúng ta có thể khởi động các tiến trình này tiếp tục.

Nếu n là số trang của tiến trình theo tổng nhu cầu của bộ nhớ, thì n-m>k cũng phải lớn hơn số trang bộ nhớ cung cấp, do đó, tổng các khoảng thời gian xử lý B_G được gia tăng một cách tuyết tính, nghĩa là:

$$B_{G+} \sim nB_1$$

Với biểu thức trên, sự trao đổi các trang cần thiết cho thấy không tiêu tốn thời gian. Tại sao vậy?

Nếu chúng ta quan sát các trạng thái, mà tại đó, các trang tồn tại một thời gian duy trì trang với giá t_s, nó được gọi là trạng thái chạy (*running phase*). Còn trạng thái được gọi là trạng thái hãm (*blocked phase*), nếu tại đó trang phải chờ đợi với thời gian chờ trung bình là tw. Tổng thời gian này được gọi là thời gian tiêu phí hệ điều hành cho việc định thời (*scheduling*) và cho việc khai khẩn (*dispatching*). Hình 3.19 cho thấy một quá trình diễn biến trạng thái của 3 tiến trình.

Hình 3.19----------

Người ta sẽ thấy gì, nếu thời gian chờ đợi nhỏ hơn thời gian trao đổi trang $t_w < t_s$, đó là, một tiến trình bị chặn luôn luôn sẵn sàng được tìm thấy. Mặc dù ở mỗi tiến trình sau trạng thái duy trì trang có xuất hiện một thời gian chờ t_w nhưng điều đó cũng không có tác dụng gì cả, tuy nhiên nó sẽ xảy ra không mĩ mãn đối với các tiến trình bất kỳ nào đó. Bây giờ, nếu chúng ta nạp cho máy tính thêm nhiều tiến trình nữa, do đó, chúng ta sẽ quan sát thấy một hiệu quả kỳ dị: Với một lượng các tiến trình tin cậy nào đó, thì khoảng thời hạn xử lý các tiến trình riêng lẻ xảy ra với một nhịp độ đột ngột cao. Người ta nói: Những tiến trình này đã quấy rầy (*quallen*) việc xử lý. Do đó, hiệu ứng kỳ dị này còn được gọi là hiệu ứng thrashing. Điều đó đặt cho chúng ta một câu hỏi: Hiệu ứn thrashing do đâu và người ta có thể phòng tránh như thế nào?

Bây giờ chúng ta khảo sát tình trạng sau đây một cách chính xác hơn: Với sự chịu tải bổ sung bởi các tiến trình, về một phía nào đó, số trang có thể được sử dụng ở trong bộ nhớ nên mỗi tiến trình thì giảm xuống (trong khoảng thời gian t_w). Vì hai tiến trình trao đổi trang không thể xảy ra đồng thời, do đó, khi $t_w < t_s$, ở thời điểm *các phương tiện điều hành là bộ vi xử lý chính (CPU)* ít được sử dụng thì xuất hiện *các phương tiện điều hành là bộ vi xử lý trao đổi trang*. Hình 3.20 chỉ ra một tình huống trao đổi này phải chờ đợi một tình huống trao đổi khác như thế nào đó để tổng khoảng thời gian xử lý G lớn hơn tích số n.B (tức là G> n.B).

Hình 3.20----------

Nói một cách chính xác: Khi nào thì hiệu ứng thrashing xảy ra? Khi nào thì $t_w = t_s$? Bây giờ, chúng ta mô hình hoá hệ thống. Ta gọi: t_T là khoảng thời gian xử lý trung bình, ρ là mức độ trao đổi trang (cũng gọi là xác xuất trao đổi trang), lúc đó ta có:

$$t_w = \rho t_T \qquad (3.1)$$

Khi sắp xếp các chỉ số trang, chúng ta thấy, các trang thường được tham chiếu nhất nhận được chỉ số nhỏ nhất. Tại mỗi trang, chúng ta quan tâm tới tỷ số giữa số lượng các trang tham chiếu và tổng số các tham chiếu trang khi thực hiện chương trình tại một thời điểm xác định. Do đó, cho mỗi trang thứ i, chúng ta nhận được một xác suất tham chiếu p_i. Điều đó được minh hoạ như hình 3.21 và sẽ được phân tích kỹ hơn ở dưới đây.

Tổng các xác suất thay đổi trang thì bằng 1, nó là diện tích giữa đường cong và các trục toạ độ, nó cũng chính là tích phân của hàm xác suất p(i) với cận I nằm trong khoảng i= [1,m].

Hình 3.21----------

Đối với sự thay đổi trang thì điều quyết định là, hàm xác suất p(i) không phải là một hằng số, mà chỉ có tất cả các trang được tham chiếu với một xác suất như nhau (ngay khi $p_i=p_c$), đặc biệt, các tham chiếu địa chỉ đều xảy ra một cách cục bộ ở dạng mã. Và đến một thời điểm nào đó, hàm xác suất p(i) sẽ chỉ luôn luôn đặt tại một ít trang xác định. Nguyên tắc định vị này sẽ dẫn tới một sự suy giảm mạnh mẽ của hàm p(i) khi chỉ số i quá lớn, và do đó đã tạo nên một cơ sở thực thi của mô hình bộ nhớ ảo. Nếu tất cả các trang ảo của tiến trình được tham chiếu với xác suất như nhau, do đó, các bảng đa bậc không cần tới bộ nhớ kết hợp, khi chúng nằm trên sự dư thừa các trang không tồn tại.

Để tiến hành công việc khảo sát tiếp theo, chúng ta mô hình hoá các trang (có khả năng) của một tiến trình với hai tập hợp: tập M_1 là tập các trang với chỉ số $i< i_M$ được tham chiếu với một xác suất p_1 (nó thuộc tập working set) và tập M_2 là tập các trang bên phải với chỉ số $i_M<i<m$ có xác suất p_2

Ta có: ρ là xác suất thay đổi trang; k là số trang của bộ nhớ chính; m là số trang có thể có của tiến trình và σ là hệ số khả năng của bộ nhớ, nó được xác định bằng tỷ số: $\sigma = k/m$

Nếu tất cả các trang được tham chiếu với xác suất như nhau, do đó, chúng ta nhận được quan hệ:

P = (số trang được thay đổi)/ (tổng số trang)

Hay $\rho = (m-k) / m = 1 - \sigma$

Khi xác suất xuất hiện trang đạt $\alpha = 1-\rho$ thì một trang thứ i tồn tại trong dãy k trang của bộ nhớ, nó phụ thuộc vào xác suất tham chiếu p_i. Nếu chúng ta áp dụng một chiến lược thay thế trang, mà nó thay thế một cách ít nhất các trang thường hay được tham chiếu nhiều nhất, do đó, xác suất tham chiếu p_i và xác suất có mặt α được liên kết với nhau. Nếu một trang với chỉ số k có mặt, khi đó, xác suất hiện trang được xác định: Tất cả các trang với chỉ số i<k có mặt, đối với mô hình của hai tập hợp trang nói trên (M_1 và M_2), chúng ta nhận được:

$$\alpha_1(k) = p_i\ (i<=k) = \sum_{i=}^{k} p_1 = kp_1 \quad \text{với } 1<=k<=I_M \qquad (3.2)$$

$$\alpha_2(k) = \alpha_1(i_M) + \sum_{i=iM+1}^{k} p_2 = i_M.p_1 + (k- i_M) .p_2 \quad \text{với } i_M <= k<= m \quad (3.3)$$

Nếu có một sự tham chiếu tới tập cần quan tâm M_1, khi $1<=i<=i_M$ thì xác suất xuất hiện trạng v được gán:

$v:= p$ (với $1<=i<i_M$)

và khi $i_M<=i<=m$ thì xác suất xuất hiện trang dẫn tới:

$p(i_M <= i<=m) = 1=v.$

Kết hợp với các biểu thức (3.2) và (3.3), ta có:

$v: = \alpha_1(i_M) = i_M.p_i \quad \leftrightarrow \quad p_1 = v/i_M.$

$1-v = \sum_{i=iM+1}^{m} p_2 = (m-i_M) .p_2 \quad \leftrightarrow \quad p_2= (1-v)/ (m-i_M)$

Hệ số trao đổi trang lạ ρ được kết hợp từ biểu thức (3.1) với sự trợ giu[psp của các cơ quan hệ (3.2) và (3.3). Từ hàm ρ này, ta xác định được hàm ρ_1 và ρ_2 cho 2 tập hợp trang:

$\rho_1 = 1 - \alpha_1 = 1 – kp_1 =1 – kv/i_M$ với $1<= k<=i_M$

$\rho_2 = 1 – \alpha_2 = 1 – v – (1-v). (k- i_M)/ (m-i_M)$ với $i_M <= k <= m$

Nếu chúng ta chuyển từ hệ số khả năng bộ nhớ tuyệt đối k tới hệ số khả năng bộ nhớ tương đối trên một tiến trình σ, trong đó, chúng ta mở rộng các biểu thức với 1/m và nhân các quan hệ trên tất cả các trang với 1/m, do đó, với quan hệ $\sigma_1 = i_M/m$ dẫn tới các biểu thức sau;

$\rho_1=1 – kv/i_M = 1-k.m.v/m. I_M =1 v.\sigma/\sigma_T$ với $1/m <= \sigma=k/m <=\sigma_T$ (3.4)

$\rho_2 =(1-v) [1- (k/m- i_M/m)/(1- i_M/m)] = (1-v).(1-\sigma)/(1-\sigma_T)$ với $\sigma_T<= \sigma<= 1$ (3.5)

Khi $\sigma=\sigma_T$ thì hai hàm ở trạng thái quá độ, lúc đó dẫn tới $\rho_1(\sigma) =\rho_2\sigma) =\rho_T = \rho(\sigma_T)$. Ở đây ta xác định được:

$\rho_T = \rho_1(\sigma_T) = 1-v$ (3.6)

Đối với mô hình thô sự phân bổ trang ở trong hai tập M_1 và M_2, chúng ta tìm thấy một quan hệ giữa hệ số trao đổi trang ρ và hệ số khả năng bộ nhớ σ. Hình 3.22 chỉ ra điều đó.

Thật vậy, khi tham chiếu trang với xác suất như nhau, thì hàm $\rho =1- \sigma$ được biểu diễn là một đường thẳng. Với sự mở rộng $\sigma = 0$, giá trị của hàm này đạt $\rho =1$. Các quan hệ (3.4) và (3.5) thì cũng là các đường thẳng, và chúng gặp nhau tại một điểm. Người ta nhận thấy rằng, khi thu hẹp hệ số khả năng trang σ, cho tới khi gặp tập hợp trang *working set* sau quá độ chuyển biến của các hàm ρ_1 và ρ_2, thì hàm của sự hoạt động trao đổi trang trở thành một đường dốc đột ngột, lưu ý trường hợp này: ta thấy hiệu quả *thrashing effect* xuất hiện. Khi hàm ρ có một sự phụ thuộc thích ứng, đường thẳng này biến thành một điểm ở trên đường cong.

Hình 3.22.**********

Nếu $t_w > t_s$, do đó, thời gian chờ đợi sẽ xác định tổng thời gian xử lý B_G. Với n tiến trình như nhau thì $B_G=n. B_1$, nếu có kể tới ảnh hưởng của thời gian chờ đợi t_w và thời gian thay thế trang t_s thì $B_G=n. B_1(t_w / t_s)$. Do chúng ta chỉ quan tâm tới một sự trình bày chất lượng độc lập với khoảng thời gian xử lý thông thường B_1, do đó, người ta đưa ra khái niệm thời gian xử lý tương đối G: B_G/ B_1 thông qua quan hệ ở biểu thức (3.1), ta có:

$G=n$ với $t_w<= t_s$

$G = n(t_w / t_s) = n.p(t_w / t_s)$ với $t_w >= t_s$

Khi $t_w = t_s$ thì các tiến trình ở thời kỳ quá độ. Trong trường hợp này thì $\rho.t_T = t_s$ với hằng số $\rho = t_s / t_T = \rho_w$. Với hệ số khả năng thích hợp σ_w, thì dẫn tới $\rho(\sigma_w) = \rho_w$. Khi đó, biểu thức thứ hai của (3.7) trở thành:

$G = n\rho/\rho_w$ với t_w / t_s (3.8)

Chúng ta có thể phân biệt 2 thông số quan trọng: Hệ số chuyển biến ρ_T giữa 2 tập hợp trang M_1 và M_2, và hệ thống chuyển biến ρ_w giữa khoảng thời gian xử lý t_s, và khoảng thời gian chờ đợi t_w. Bây giờ chúng ta xem xét, tính chất của hệ thống phụ thuộc như thế nào vào hai thông số này ?

Chúng ta khảo sát trường hợp $\rho_T <= \rho_w$ với $\sigma_T >= \sigma_w$. Chúng ta phân chia khoảng tổng cộng của σ (xem hình 3.22) thành 2 khoảng (A và B) như hình 3.23 ở dưới đây.

Hình 3.23**********

Khoảng A: Với $\sigma_w <= \sigma$ thì cũng có $\rho <= \rho_w$ và $t_w <= t_s$. Tức là trong khoảng (a) thì biểu thức (3.7) trở nên:

$G_A = n$ với $\sigma_w <= \sigma$

Đây là khoảng tuyến tính

Khoảng B: Từ biểu thức (3.8) và biểu thức (3.4) dẫn tới ký hiệu:

$\sigma = k/(n.m) \sim s/n$

Với hệ số khả năng bộ nhớ tương đối với σ khi (n-m) trang yêu cầu bộ nhó qua n tiến trình thì thời gian xử lý tương đối G_B ở khoảng này được viết:

$G_B = n - \rho_1/\rho_w = (n/\rho_w)(1-\sigma.v/\sigma_T) = n/\rho_w - s.v/\rho_w\sigma_T$

Khoảng B là khoảng tuyến tính mạnh mẽ.

Câu hỏi đặt ra: Tại n_0 tiến trình nào thì xuất hiện sự quá độ chuyển biến từ khoảng A tới khoảng B? Điều kiện để xảy ra chuyển biến là:

$G_A(n_0) = G_B(n_0) \leftrightarrow n_0 = n_0(\rho_1/\rho_w) \leftrightarrow$

$\rho_w = \rho_1 = (1-(s/n_0).v/\sigma_T) \leftrightarrow \rho w.\sigma_T = \sigma_T - s.v/n_0 \leftrightarrow$

$n_0 = s.v/\sigma_T(1-\rho_w)$ (3.9)

Thí dụ về sự chịu tải phi tuyến khi sự thay thế trang thấp:

Nếu một hệ thống có đủ chỗ cho 2 tiến trình (s=2) thuộc tập *working set*. Giả sử có một nửa số các tiến trình có hệ số khả năng bộ nhớ $\sigma_T = 0,5$, mà có tới 90% số đó được người ta quan tâm, tức là $v=0,9 \leftrightarrow \rho_T = 0,1$. Vì $\rho_T <= \rho_w$, nếu chúng ta chấp nhận được $\rho_w = 0,2$ thì với sự chấp nhận này, chúng ta tính được:

$n_0 = 4,5$ và $G_B (n=6) = 6.5-2.0,9.5.2 = 12$ (xem hình 3.24)

Hình 3.24**********

Người ta thấy, tuy không gian bộ nhớ chính chỉ đủ chỗ cho 2 tiến trình, nhưng nếu không có tổn hao, nó có thể thực hiện được 4 tiến trình. Nếu người ta gia tăng số tiến trình, thí dụ n=6, do đó, một sự chịu tải gấp đôi là hiệu quả.

Bây giờ, chúng ta khảo sát một trường hợp khác, với $\rho_T <= \rho_w$ và $\sigma_T <= \sigma_w$. Chúng ta có thể phân chia khoảng σ ra làm 3 phần như ở trong hình 3.25 dưới đây.

Khoảng C: với $\sigma_w <= \sigma$ thì $\rho <= \rho_w$, do đó, suy ra $t_w <= t_s$. Trong khoảng C phương trình (3.7) trở thành:

$G_C = n$ với $\sigma_w <= \sigma$

Khoảng C là khoảng tuyến tính.

Khoảng D: từ phương trình (3.8) và (3.5) ta có:

$G_D = (n/\rho_w)/\rho_2$, với $c = 1 - \sigma_T$, người ta nhận được:

$G_D = (n.c/\rho_w)/(1-s/n) = n.c/\rho_w - c.s/\rho_w$, hay:

$G_D = (n-s).c/\rho_w$ với $\sigma_T <= \sigma <= \sigma_w$

Khoảng D còn gọi là khoảng trên tỷ lệ.

Khoảng E: Khi $0 <= \sigma <= \sigma_T$, người ta nhận được:

$G_E = n.\rho_1/\rho_w = (n/\rho_w).(1-\sigma_V/\sigma_T)$, hay:

$G_E = n/\rho_w - s.v/\rho_w.\sigma_T$

Khoảng E còn gọi là khoảng tuyến tính mạnh.

Những kết luận và các chiến lược:

Từ những việc nghiên cứu được trình bày như ở trên dẫn tới những kết luận sau đây:

Điều quan trọng là, các trang của tập công tác của tất cả các tiến trình phải có đủ chỗ ở trong bộ nhớ (khoảng G_A và G_C). Nếu điều đó không được đảm bảo, do đó, số lượng các tiến trình bị thu hẹp, cho đến khi các tiến trình có đủ không gian ở trong bộ nhớ.

Tuy nhiên, nếu điều đó không đạt được, khi đó phải dẫn tới các điều kiện $\sigma >= \sigma_w$ và $\rho(\sigma) <= \rho_w = ts / t_w$. Khả năng bộ nhớ phải được xác định trên quan hệ của thời gian duy trì trang và khoảng thời gian thay thế trang. Nếu điều đó không xảy ra, do đó, chúng ta sẽ nhận được hiệu quả *thrashing effect*; mãi cho đến khi, có đủ bộ nhớ cho tập công tác (*working set*). Bây giờ, chúng ta muốn làm cho ρ_w lớn và σ_T nhỏ để đạt được những tính chất thuận lợi nhất của hệ thống. Về điều đó, có những biện pháp phần cứng và phần mềm cần thiết sau đây được trình bày.

- *Những biện pháp phần cứng*:

Ở gốc độ những biện pháp phần cứng, điều phải đạt được là thời gian duy trì trang t_s phải đủ lớn, mà trong đó, người ta cần thiết làm cho trang đủ lớn. Điều thuận lợi là, thời gian duy trì đối với một trang thì phụ thuộc mạnh mẽ vào việc làm trễ sự truy cập ban đầu (đối với ổ đĩa cứng khoảng 10 ms) và không phụ thuộc vào thời gian truyền đạt. Tốt hơn là đừng làm trễ bộ nhớ quảng đại khi truy cập nhanh. Tuy nhiên, khi đó, bộ nhớ quảng đại đảm nhận vai trò bộ nhớ chính, do đó, vấn đề khó khăn nói chung được thu hẹp.

Ngoài ra, người ta có thể thu nhỏ thời gian quá độ chuyển đổi t_T, bằng cách: người ta phải phòng ngừa khi nhiều bộ nhớ song hành nạp trang. Phương pháp này gọi là phương pháp dùng nhiều ổ đĩa đổi tráo.

Với σ_T nhỏ hay v lớn đạt được khi hiệu chỉnh mã chương trình trên bình diện lập trình. Điều đó được thực hiện nhờ việc sao chép các thủ tục, mà chúng được sắp xếp theo bậc gọi của thủ tục. Một thí dụ về điều đó cho thấy, đó là sự thiết lập thành hàng các procedure thay

thế bằng một việc gọi một thủ tục (procedure call). Điều đó làm lưu ý tới những vòng lặp lớn. Tốt hơn, những vòng lặp này cần thiết được phân chia thành nhiều vòng lặp nhỏ hơn.

Thuật toán có thể được thay đổi một cách thích hợp, thí dụ, phép nhân các ma trận có thể được phân chia, đầu tiên, nhân theo hàng, khi đó, ma trận được lưu trữ theo hàng. Khi nhân theo cột, thì các trang được thay đổi theo cột.

Tập working set và mô hình tần suất thay đổi trang:

Hệ điều hành cần được thực hiện một cái gì đó để phòng tránh hiệu quả *thrashing effect*. Khi đó một chiến lược được nêu ra: Chúng ta có thể xác định các tập *working set* đối với mỗi tiến trình với sự trợ giúp của phép thống kê thời gian, thí dụ các Bits R và M (nói ở mục 3.3.5). Khi đó chúng ta phải quan tâm rằng, số lượng các tiến trình được thu nạp sao cho đáp ứng việc bộ nhớ chính tồn tại vừa đủ cho các tập *working set*. Lúc đó, kiểu làm này được gọi là mô hình tập công tác (*working set model*). Các bộ nhớ BS2000 (của hảng Siemens) và CP37 (của hảng IBM) thích hợp cho trường hợp này.

Về vấn đề này còn có một chiến lược khác được quan tâm: Nếu F là số đo các sự thay thế trang trên một đơn vị thời gian, được gọi là tỷ phần tráo đổi trang; cũng như thời gian giữa hai lần thay đổi trang vượt lên hay lui lại một giá trị F_0; lúc đó người ta gọi là tần suất thay đổi trang, do đó, các tiến trình được đặt yên tĩnh trong chốc lát. Nếu điều đó xảy ra thì có thể có nhiều tiến trình được hoạt động. Các mô hình này được W.W. Chu chỉ ra (1975), cho thấy, nó thì tốt hơn so với mô hình *working set modell*.

- **Mô hình bậc sử dụng:**

Một ý tưởng khác được nêu lên là cần thiết phải điều chỉnh trực tiếp khả năng của hệ thống. Về điều này, chúng ta nghiên cứu điều kiện xuất ra một cách bình thường các tiến trình. Mỗi khi chúng ta nhận được các tiến trình, lúc đó, sức chịu tải của CPU thì cao hơn, kéo dài cho tới khi, thời gian đợi t_w lớn hơn thời gian duy trì trang t_s. Trong trường hợp này, các trang bị ùn đọng khi tiến hành thiết lập sự trao đổi trang. Chúng ta có thể mô hình hóa điều đó như là một bộ vi xử lý trao đổi trang CPU và chúng ta có thể mô hình hóa sự chịu tải của hệ thống L(n,t) như là một tổng của hai tải η_{CPU} và η_{PPU}.

$L(n,t) = W_t.\eta_{CPU} + W_2.\eta_{PPU}$

Trong đó, L(n,t) còn gọi bậc sử dụng; W_1. W_2 là các trọng số

Mô hình được M. Badel đưa ra (1975), chỉ ra rằng, mô hình này chứa đựng nội dung: Đầu tiên tăng theo một đường dốc và sau đó giảm xuống nhanh tới bậc sử dụng L(n,t) khi vượt qua số lượng các tiến trình tới hạn. Khi nghiên cứu sự biến đổi các thông số tiến trình ở mục 2.2.2, ở đây chúng ta thấy L(n,t) được xác định từ các đại lượng đo đạc. Nếu L(n,t) giảm, do đó, số lượng các tiến trình phải bị thu nhỏ;ngược lại, chúng phải được gia tăng. Ngay cả một phản ứng làm trễ cũng có thể dẫn tới việc phòng tránh một sự thay đổi cực nhanh.

- *Chiến lược cục bộ và toàn cục*:

Một khả năng tiếp theo là, các tiến trình và nhu cầu bộ nhớ của chúng không phải là hai vấn đề cô lập nhau, mà không gian tồn tại giữa chúng được phân chia một cách năng động. Thuộc về điều đó, người ta không thể phân bổ cho mỗi tiến trình một không gian bộ nhớ như nhau, mà người ta phân bổ theo nguyên tắc: không gian nhớ tỷ lệ với độ lớn của mỗi tiến trình. Để thoả mãn nguyên tắc này, độ lớn ban đầu của bộ nhớ không được nhỏ quá. Như vậy, chiến lược này không chỉ lưu ý số lượng sử dụng các trang, mà còn quan tâm tới cả độ lớn thay đổi năng đông của tập *working set*.

Về việc xác định cục bộ độ lớn này cho mỗi tiến trình, người ta có thể thực nghiệm, để truy tìm một chiến lược về các trang của các tiến trình, mà nó đem lại hiệu quả hơn. Do đó, các thuật toán LRU và LFU (ở mục 3.2) làm việc trên tập các trang và xác định tập công tác của các tiến trình một cách năng động. Đối với thuật toán PFF thì có ý nghĩa hơn, hệ điều hành cần thiết phải phân bổ bộ nhớ để sao cho tần suất thay thế trang ở tất cả các tiến trình là như nhau.

Nhược điểm của chiến lược toàn cục là ở chỗ: một tiến trình có thể có tác dụng lên tất cả các tiến trình khác. Nếu một tiến trình tồn tại với yêu cầu bộ nhớ lớn, do đó, tất cả các tiến trình khác sẽ bị cưỡng bức: thường phải thay đổi trang hơn hay thường phải chờ đợi. Ở một chiến lược cục bộ thì ngược lại, chỉ có những tiến trình lớn phải chờ đợi, còn các tiến trình khác tiếp tục làm việc với không gian trống của chúng. Tuy nhiên, đối với tất cả các tiến trình, thì điều đó chưa tối ưu, nó sẽ được cảm nhận một cách thoả đáng hơn trong các hệ thống đa người sử dụng.

Chiến lược lazy evaluation:

Có ý kiến cho rằng, để thu hẹp tiêu phi công tác cho các trang, người ta thực hiện nguyên tắc *lazy evaluation* như sau: Tất cả mọi hoạt động được xê dịch cho đến chừng nào đó được thoả mãn. Sau đây là vài thí dụ mô tả nguyên tắc này:

- *Copy on write* (sao khi viết):

Nếu việc sao chép trang là cần thiết, do đó, một trang gốc chỉ được thi hành bởi một lần làm dấu. Đầu tiên, trang được truy cập để viết, tức là: trước đó bản sao gốc được đặt lên, và sau đó trang được sửa chữa và được copy.

- *Pape oout Pool* (trang ra khỏi ao):

Đáng lẽ trang được viết ra ngay, thì nó lại được nạp vào bộ nhớ quảng đại; khi đó, người ta nạp chúng vào một kho chứa (depot); vì vậy, trang ở trạng thái standby (đứng cạnh). Sau đó, có một trang được dùng ngay trở lại, mà nó không bị lấy lại bởi ổ đĩa.

Một chiến lược *pape out pool* như thế tạo nên một việc đặt chỗ cho trang, dù các trang được sử dụng khác nhau. Nếu có một ít trang tự do, khi đó, người ta dùng chúng trở lại ngay cho trang mới; và nếu còn nhiều không gian trống nữa, khi đó, người ta đặt chúng lâu hơn tí nữa và khảo sát chúng như là một nơi đặt chỗ đầy tiềm năng cho sự hoạt động của trang.

Vấn đề điều kiện biên:

Bên cạnh vấn đề chính như giới hạn không gian bộ nhớ phải được phân chia khi có tranh chấp tiến trình, cũng còn nhiều vấn đề khác, mà chúng có quan hệ chặt chẽ với việc điều hành trang.

- *Giới hạn lệnh*:

Ngắt lỗi trang, hoạt động được nhờ việc định địa chỉ phần cứng, mà nó dẫn tới sự đình chỉ ngay lệnh máy đang được thực hiện, nếu địa chỉ đích không ở trong bộ nhớ.

Nếu với cơ chế thay thế trang, trang mong muốn được tồn tại trong bộ nhớ chính, do đó, lệnh này phải được thực hiện một cách mới mẻ. Tuy nhiên, về điều này, hệ điều hành phải nhận biết: Có ở đâu một lệnh tồn tại với nhiều Bytes (?), và không chỉ vậy, do đâu lệnh đã tham chiếu một địa chỉ không thuật lời (?). Từ lý do này, lệnh cần thiết được dẫn tới một nhân tử, nếu lệnh bị bẻ gãy thì nó không còn tác dụng nào cả, và nếu không, lệnh cần thiết được dẫn tới một nhân tử khác, khi đó, địa chỉ ban đầu của nó được nạp vào thanh ghi. Nếu trường hợp nói trên không xảy ra, hệ điều hành phải khai khẩn tận lực các dữ liệu cần thiết từ một ngăn xếp mã microcode hay các biến khác, do đó, một sự thay thế trang được làm trễ là không cần thiết.

- *Trang xuất nhập và trang chia xẻ (I/O pages and shared pages*)

Vấn đề tiếp theo, đó là việc xử lý các trang đặc biệt. Nếu đối với một tiến trình, một sự trao đổi dữ liệu I/O được đụng chạm tới và một tiến trình khác nhận được bộ vi xử lý, do đó, điều có thể là, trang I/O của một tiến trình chờ đợi được nạp vào không gian của một sự phân bổ bộ nhớ toàn cục và trang vật lý được trao cho một cách mới mẻ. Nếu sau khi thay đổi trang, bộ vi xử lý I/O nhận được sự điều khiển, do đó, trang sai (page fault) được sử dụng cho I/O; nếu rủi ro, nó được sử dụng cho cả hai tiến trình. Trong trường hợp này, việc khắc phục tạo nên một dấu hiệu của các trang I/O và nó thực hiện việc loại bỏ trang đánh dấu khi thay thế trang.

Một vấn đề nữa về các trang, đó là các trang, mà chúng được sử dụng trong nhiều tiến trình (thí dụ: *shared pages*), hoặc trang mã của một thư viện chung (thí dụ thư viện ngôn ngữ C). Những trang này có ý nghĩa đáng kể, chúng không cho phép nạp cùng với một tiến trình hay tại thời điểm kết thúc chúng được phép đan nhau, khi chúng được một tiến trình khác tham chiếu. Ngay tại các thuật toán thay thế trang, điều đó phải được quan tâm tới.

- *Paging demon* (lập trang ma):

Công việc của các thuật toán thay thế trang có thể trở nên hiệu suất hơn, nếu người ta xem xét mã (code) như là một tiến trình lập trang ma (paging demon) và người ta để cho nó xảy ra ở trạng thái không tải một cách đều đặn. Theo đó, những yêu cầu không chỉ được làm thích hợp, mà còn có thể được dọn dẹp cho một không gian trống; khi đó những thông tin thống kê được trở nên bức bách và dịch vụ hệ thống được thực hiện, thí dụ như việc điều hành chiến lược page out pool được nói trước đó.

3.3.7. Chiến lược thay thế trang ở hệ điều hành Unix:

Ở trong Unix, có hai cơ chế được trợ giúp. Đó là *swapping và paging* (xếp chồng và lập trang). Việc thu xếp bộ nhớ trực tiếp của cơ chế *swapping* luôn luôn được sử dụng, nhất là, nó được dùng khi thời gian truy cập nhanh. Vì vậy, ở việc sản sinh một tiến trình, thì một sự đặt

chỗ về không gian bộ nhớ ở khoảng *swap* cho việc nạp một tiến trình được dự đoán. Khoảng *swap* có thể tự tổ chức một cách khác nhau, nó thì có thể là một đơn vị vật lý hay là một đơn vị logic, nó có thể chỉ là một phần ổ đĩa hay là một phần của hệ thống files. Những nguồn gốc sinh sản này được thực hiện ở trong dãy tuần tự của khoảng thời gian truy cập.

Đối với việc thay thế trang, ở hệ điều hành Unix có một sự cạnh tranh giữa hai cơ chế vừa nói, cũng vì thế chúng được thực hiện bởi 2 tiến trình nền (background processes). Tiến trình nền để nạp trang sẽ đặt trở lại các Bits R tham chiếu trang với những khoảng thời gian đều đặn. Sauk hi đặt trở lại, tiến trình chờ đợi thêm một khoảng thời gian Δt và tiếp tục nạp tất cả các trang, cho đến khi Bits R đạt giá trị 0.

Tuy nhiên, cơ chế này chỉ có hiệu lực khi nhỏ hơn 25% không gian trống của bộ nhớ chính được sử dụng (điều đó có nghĩa là: dung lượng bộ nhớ chính giảm vì nó phải bao gồm dung lượng của nhân hệ điều hành, các tiến trình nền, bộ kích tạo các thiết bị, giao diện người sử dụng…). Vì việc chạy lướt qua tất cả các trang xảy ra quá lâu đối với một bộ chỉ thị, do vậy, ở trong Unix có thêm bộ chỉ thị thứ hai để thực hiện việc kiểm tra lại các Bits R và việc nạp trang, bộ chỉ thị này cũng chạy trong khoảng thời gian Δt nói trên. Thuật toán chỉ thị giờ này được đảm nhiệm ở trong hệ thống Unix V. Để chinh phục hoạt động nạp trang mà không có điều gì xảy ra, thì trang được nạp chỉ khi chúng ở lại trong n lối đi qua liên tiếp chưa được dùng.

Nếu chỉ còn ít không gian bộ nhớ có thể được dùng, do đó, hoạt động nạp trang được bắt đầu từ một giá trị ngưỡng cửa xác định của tiến trình nền swap, nó được tác động bởi các tiến trình dài và nó được nạp trên bộ nhớ quảng đại, cho đến khi không gian trống bộ nhớ đạt nhỏ nhất.Nếu các hoạt động nạp trang đè lên nhau, do đó, các tiến trình được hoạt động trở lại. Trong hệ thống Unix V, cho cái đó, có hai giá trị max và min được thực hiện để tránh một sự lưỡng lự ở khoảng giới hạn. Nếu chúng ta khảo sát một tiến trình lớn mà nó đang được nạp, do đó, không gian trống bộ nhớ được gia tăng rõ rệt. Bây giờ, nếu tiến trình được nhận trở lại, thì không gian trống bộ nhớ giảm đi một cách nhịp nhàng cho tới dưới giới hạn can thiệp (chiếm đủ) và tiến trình được nạp trở lại. Sự dao động khi trang được xuất đi hay được nạp vào có thể được hạn chế khi không gian trống bộ nhớ lớn hơn giá trị min (khi tiến trình được nạp), nhưng mà không lớn hơn giá trị max (khi không có tiến trình nào xuất đi).

Nếu không có phương tiện nào để tránh hiệu quả thrashing effect, do đó, phải được một bộ điều hành hệ thống can thiệp. Những biện pháp như dùng thiết bị phân cách (swap device) để tách chia hệ thống xấp và hệ thống files, như việc phân bổ khi xếp chồng hay lập trang trên nhiều ổ đĩa, hay việc tạo lập các bộ nhớ chính phụ đều có thể mang lại những giảm nhẹ mong muốn.

3.3.8. Chiến lược thay thế trang ở trong Windows NT:

Ở trong các chương trình trước đã nghiên cứu các chiến lược toàn cục và nghiên cứu nhiều thuật toán thay thế trang. Chúng rất thuận tiện trong sử dụng để quản lý bộ nhớ, và chúng cũng mang lại nhiều kết quả khả quan. Ngược lại, trong hệ điều hành Windows NT, chiến lược thay thế trang bình thường còn liên quan đến chiến lược thay thế trang cục bộ FIFO. Những lý do thật đơn giản: Vì chiến lược thay thế trang toàn cục có tác dụng từ một tiến trình này tới một tiến trình khác, do đó, với một tiến trình nào đó, các cơ chế thực thi sẽ chọn một chiến

lược, mà nó chỉ giới hạn cục bộ trên một tiến trình riêng lẻ, để xác định một cách tích cực sự phê phán của người sử dụng về hệ điều hành. Với một tiến trình khác, thuật toán đơn giản để thực thi cơ chế thay thế trang và thuật toán FIFO có thể được thực hiện với tiêu phí thời gian ít nhất, do đó, trong trường hợp bình thường, thời gian thực hiện các chiến lược sẽ giảm đáng kể. Nhờ thế, lỗi cũng được hạn chế; lỗi thường xuất hiện khi nạp trang; do đó, đầu tiên, các trang được chuyển đi bởi chiến lược page out pool ở trong bộ nhớ chính; từ đó, chúng dễ dàng được nhận trở lại. Tuy nhiên, khi có ít bộ nhớ ở trong hệ thống; do đó, một cơ chế thứ hai có hiệu lực gọi là cơ chế xén tỉa tự động tập công tác. Với cơ chế này, tất cả các tiến trình được dẫn qua và được kiểm tra, liệu chúng có sử dụng nhiều trang không, khi mã các trang này là một số tối thiểu cố định. Số các trang tối thiểu này có thể được điều chỉnh trong khoảng giới hạn xác định của người quản lý hệ thống. Con số hiện hành của các trang được sử dụng cho một tiến trình thì được biểu thị là tập công tác ở trong Windows NT (ngược lại với định nghĩa nêu ở trên). Nếu tiến trình sử dụng quá nhiều trang trong tập công tác tối thiểu; do đó, các trang sẽ được nạp và tập công tác bị thu hẹp. Bây giờ, nếu một tiến trình chỉ có một số tối thiểu các trang, nó sẽ phát sinh lỗi trang và không gian bộ nhớ chính lại được tồn tại trở lại (được giải phóng), do đó, độ lớn tập working set lại được gia tăng và tạo cho tiến trình có nhiều không gian bộ nhớ.

Ở việc sản sinh tiến trình, một phương pháp có tên clustering methode được áp dụng: một số trang được dẫn đi trước đó và sau đó bổ sung cho các trang còn thiếu, với mục đích tiếp tục làm giảm xác suất lỗi trang. Điều đó phù hợp với một chiều dài trang có kích cở hiệu quả cao.

Một biện pháp tiếp theo là tạo lập việc áp dụng phương pháp *copy on write*, mà phương pháp này thì đặc biệt hiệu nghiệm khi sản sinh ra một tiến trình mới ở hệ thống con POSIX với hàm *fork()*. Vì các trang của một tiến trình cha được làm dấu chỉ với Bit theo phương pháp *copy on write* và sau đó tiến trình con thực hiện nhờ một hàm gọi hệ thống *exec()* và chịu tải bởi một mã chương trình khác, do đó, các trang của tiến trình cha không cần thiết phải sao chép lại và vì thế, tiết kiệm được thời gian.

Ở các files thư viện kết nối động (Dynamic Link Library- DLL), một cơ chế tương tự cũng được sử dụng, khi đó các files này chỉ được đưa tới một lần. Các dữ liệu tĩnh (static data) của chúng được bảo về nhờ trạng thái *copy on write*. Bây giờ, nếu tiến trình khác nhau được truy cập và các dữ liệu được thay đổi; do đó, các dữ liệu được sao chép trên cung dữ liệu riêng lẻ (*private data segment*).

3.4. Sự phân cung (*segmention*)

Cho tới bây giờ, sự nhìn nhậ của chúng ta về nộ nhớ ảo được mô hình hóa là một trường liện tục đồng đều. Thực chất, hình dạng bộ nhớ này không được người lập trình sử dụng. Vì lẽ, có những đoạn cung dữ liệu (datasegments) luôn luôn quay trở lại dạng các ngăn xếp hay các xấp (*stacks or heaps*), nghĩa là chúng thay đổi một cách năng động, và do đó, chúng cũng cần dùng với nhau một đoạn trong không gian địa chỉ. Trong hình 3.28 (a) chỉ ra một cách bao quát cấu trúc logic sự dẫn giải bộ nhớ của một chương trình ở trong Unix. Trong hình 3.28 (b) chỉ ra một sự phân chia bộ nhớ đối với một trình biên dịch. Ở đây, vấn đề dẫn tới là, mỗi đoạn cung của các dữ liệu công tác có thể thay đổi và thu hẹp ở trong phạm vi,mà khi đó đang thực

hiện chương trình. Ví dụ: bảng tượng trưng có thể thay đổi trong khoảng mã nguồn. Do đó, nó dẫn tới sự chồng chéo các không gian địa chỉ.

Hình 3.28**********

Từ lý do này, chúng ta hoàn thiện mô hình đơn giản của bộ nhớ ảo với sự trợ giúp của việc phân đoạn logic tới một bộ nhớ ảo được phân đoạn. Để thực thi việc quản lý bộ nhớ kiểu mô hình, ở vị trí bảng trang, mô hình này sử dụng một bảng các địa chỉ cung đoạn. Bảng cung đoạn thuận tiện cho một chương trình hay một modun chương trình, bảng này cũng được giữ ở trong các thanh ghi thích hợp; các thanh ghi cung đoạn với tư cách là bản copy nhằm tăng tốc độ tính toán giữa các địa chỉ ảo và địa chỉ vật lý. Ở việc chuyển đổi giữa các chương trình hay các modun chương trình, chúng đồng thời nằm ở trong bộ nhớ chính và chỉ có nhóm các thanh ghi cung đoạn được nạp mới.

Trong hình 3.29 sau đây chỉ ra một thí dụ đơn giản về việc định vị cung đoạn ở bộ vi xử lý 80286 (của hảng Intel), ở đó chỉ ra hai đoạn cung mã và hai đoạn cung dữ liệu toàn cục. Nếu người ta sử dụng các đoạn cung này trong nhiều chương trình, do đó, người ta cần dùng một bảng cung đoạn và khoảng dữ liệu toàn cục hay khoảng ngăn xếp hệ thống, thí dụ áp dụng điều này để trao đổi thông tin giữa các chương trình. Điều thuận tiện là để cố định các bộ chỉ thị (pointer) tới các segment ở trong thanh ghi. Ở bộ vi xử lý 80286, nó bao gồm thanh ghi cung đoạn mã, thanh ghi cung đoạn dữ liệu, thanh ghi cung đoạn ngăn xếp và thanh ghi cung đoạn dữ liệu nâng cao.

Hình 3.29**********

Vì các cung đoạn chỉ rõ những đơn vị bộ nhớ có độ lớn không đều đặn, do đó, bộ nhớ có thể được điền đầy khi giải phóng một cung đoạn lớn hay khi nạp một cung đoạn nhỏ. Từ lý do này, khi thường xuyên tráo đổi trang, nó có thể dẫn tới việc tách nhỏ bộ nhớ; những phần nhỏ còn lại thì có thể không được chỉ dẫn gi thêm và có thể được gom lại. Do đó, việc lập trang và việc phân đoạn được liên hợp với nhau để dành lại những ưu điểm của phương pháp: Mỗi cung đoạn được phân nhỏ thành nhiều trang đều đặn.

Hình 3.30. chỉ ra một cách tổng quát về bộ vi xử lý 80486 của hảng Intel, các cấu trúc dữ liệu được phân đoạn thành hai phần: ở cung đoạn cục bộ được mô tả bằng một bảng miêu tả cục bộ (local description table), nó tồn tại một cách riêng lẻ đối với mỗi tiến trình và nó thực hiện sự biên dịch các địa chỉ cung đoạn ảo ở trong khuôn khổ trang vật lý, cũng như ở trong cung đoạn toàn cục bảng miêu tả toàn cục (*global description table*), nó tạo khả năng truy cập các dữ liệu được thu gom trên cung đoạn toàn cục đối với tất cả các tiến trình. Trước hết, đó là mã hệ điều hành mà nó được tiến trình trả lại ở trạng thái nhân, cũng như các khoảng nhớ (shared memory) của hệ thống.

Trong hình 3.30 một vài cung đoạn của tiến trình hoạt động B cho thấy: Tiến trình chưa hoạt động A chờ đợi phân bổ của bộ vi xử lý ở trong bộ nhớ. Các cung đoạn được biểu thị thành những khối, mà ở đó, chúng được thâu tóm các bảng mô tả trang với các khung trang. Người ta thấy rằng, khi chuyển đổi tiến trình sự quá đọ trên không gian địa chỉ ảo của tiến

trình A thì rất đơn giản: Ở tại thanh ghi LDT, người ta phải nạp bộ chỉ thị thời gian khác tới bảng LDT của tiến trình A, đó là tất cả.

Ưu điểm của bộ nhớ ảo phân đoạn là ở chỗ có một sự định vị địa chỉ cũng như việc quản lý bộ nhớ rất hiệu nghiệm. Các địa chỉ cơ sở của cung đoạn thì cố định, và với cái đó, nếu các thanh ghi tồn tại ở trong CPU, thì bậc đầu tiên của việc chuyển đổi địa chỉ xảy ra rất nhanh. Nếu một cung đoạn được cấu trúc đồng đều (tức là không có lỗ hổng địa chỉ), do đó, mô hình khảo sát được cải tiến: Mỗi khi một cung đoạn bộ nhớ (memory segment) tăng hay giảm, thì ở tại các bảng nhiều bậc, việc sản sinh hay loại bỏ các bảng bổ sung xảy ra rất năng động.

Tuy nhiên, phương cách kết hợp giữa các cung đoạn và các trang thực ra rất cần thiết, vì cả hai thông tin quản lý (về bảng cung đoạn và bảng trang) luôn luôn được xuyên suốt, được cải tiến và được sử dụng. Từ lý do này, ở các bộ vi xử lý ngày nay, các thanh ghi địa chỉ và các bộ MMU đã tiếp nhận một vị trí quan trọng bên cạnh khả năng tính toán thuần khiết: Chúng quyết định tốc độ xử lý các chương trình.

3.5. Bộ nhớ truy cập nhanh (Cache)

Ưu điểm của cấu trúc một CPU tốc độ nhanh dẫn tới một cách đúng đắn cho sự sinh lợi, khi bộ nhớ chính không chỉ có dung lượng lớn mà còn phải có tốc độ nhanh. Vì thế ở đây, các bộ nhớ truy cập năng động DRAM (dynamic random acess memory) không thuộc loại bộ nhớ ta muốn đề cập trong trường hợp này. Tuy nhiên, để thay thế những bộ nhớ đắt tiền, người ta có thể sử dụng bộ nhớ giá cả tương đối lẻ với lưu ý: mã chương trình (programcode) thường chỉ ra quan hệ cục bộ với các dữ liệu và với các chiều rộng bước nhảy. Tính chất cục bộ này của các chương trình tạo điều kiện để những đoạn chương trình nhỏ được sao chép vào bộ nhớ truy cập nhanh (cache) và do đó đạt được tốc độ xử lý cao.

Để làm đầy cache, khả năng đơn giản nhất là, thay vì lựa chọn nội dung của một địa chỉ, người ta có thể lựa chọn nội dùng của nhiều địa chỉ tương tự nhau ở trong bộ nhớ cache, trước khi các nội dung được yêu cầu hiển thị. Việc vận chuyển dữ liệu bằng bộ nhớ DRAM thì tương đối chậm và chỉ được thực hiện qua một lần. Ở bộ nhớ cache, việc vận chuyển một khối dữ liệu (chẳng hạn 4 Bytes) thì nhanh hơn nhờ một bus, gọi là *pipline cache bus*. Loại bộ nhớ cache này làm gia tăng khả năng của bộ vi xử lý lên khoảng 20%. Hình 3.31 chỉ ra sơ đồ vị trí sử dụng bộ nhớ cache.

Hình 3.31.**********

Nếu bộ nhớ cache chứa đựng nhiều dữ liệu, do đó điều dễ nhận thấy, lúc đó các dữ liệu không hiện hành hay chưa cần dùng được cho trôi qua. Một dị bản tổng hợp của cache chứa đựng các cơ chế bổ sung để thực hiện nội dung của cache. Vì thế, tất cả việc dò hỏi địa chỉ của CPU được cache xử lý đầu tiên; khi đó, đối với các lệnh và các dữ liệu, bộ nhớ cache bị tách chia (thành cache lệnh và cache dữ liệu) được sử dụng. Nếu nội dung của địa chỉ này được tồn tại ở trong cache, do đó, cache được chọn một cách trực tiếp và nhanh chóng, mà không phải chọn bộ nhớ chính. Nếu địa chỉ không tồn tại, do đó, cache phải được nạp lại từ một bộ nhớ chính. Hình 3.32 chỉ ra sơ đồ nguyên lý kết nối cache ở trong máy tính.

Hình 3.32**********

Tuy nhiên, bộ nhớ cache chuẩn bị cho các nhà phát triển hệ điều hành nhiều vấn đề nhức óc. Không chỉ ở trong phạm vi đa vi xử lý, mà cả ở trong một máy tính đơn vi xử lý, có những đơn vị (units) làm việc song song và độc lập với CPU, thí dụ các thiết bị xuất nhập (đĩa từ…), xem hình 3.33 ở dưới. Nếu các thiết bị này truy cập trực tiếp bộ nhớ để đọc hay viết các khối dữ liệu, do đó, việc ngắt đoạn làm ảnh hưởng việc xử lý chương trình. Bởi vậy điều đó cho thấy, cache và bộ nhớ chính phải được thực hiện độc lập với nhau.

Hình 3.33**********

Điều dẫn tới sự kết nối không bền vững của các dữ liệu với cache và bộ nhớ chính được giải quyết bằng các chiến lược khác nhau. Trong trường hợp này, người ta quyết định chọn bộ vi xử lý thứ hai làm bộ nhớ cho việc đọc và viết. Trong trường hợp thứ nhất, đáng lẽ các dữ liệu hiện hành của cache được đọc, thì lại đọc các dữ liệu cũ từ bộ nhớ chính của bộ vi xử lý thứ hai. Đối với vấn đề này tồn tại hai giải pháp:

- *Viết qua (write through)*: Bộ nhớ cache được đảm nhiệm như là bộ đệm đọc (read buffer); còn bộ vi xử lý dùng để viết nhờ bus hệ thống ở trong bộ nhớ chính. Lúc đó, các dữ liệu ở trong bộ nhớ chính luôn luôn là hiện hành. Tuy nhiên, nó thì hơi chậm, vì nó làm trễ bộ vi xử lý.
- *Viết lùi (write back)*: Bộ nhớ cache tự giải quyết việc tiến lùi lại mà không cần tới bộ vi xử lý. Đối với bộ vi xử lý, điều đó xảy ra nhanh hơn, nhưng nó đòi hỏi một chi phí phụ ở bộ nhớ cache.

Cả hai chiến lược vừa nêu vẫn chưa loại trừ được tình huống thứ hai, khi một thiết bị khác thay đổi bộ nhớ nhờ việc truy cập bộ nhớ trực tiếp, mà không cần điều đó được phản ảnh ở trong bộ nhớ cache. Để gói gọn vấn đề không ổn định dữ liệu, bộ nhớ cache phải được trang bị thêm bằng một phần bổ sung, mà nó ghi lại, liệu một địa chỉ của phần này được tham chiếu trên bus địa chỉ của bộ nhớ chính để đọc và viết hay không (!); trong trường hợp này có giải pháp thứ ba như sau:

- *Sao chép lùi (copy back)*: Nếu bộ nhớ được viết, do đó, đối với tế bào nhớ thích hợp ở trong cache thì một Biit đặc biệt được thiết lập. Nếu CPU đọc tế bào nhớ này, do đó, Bit được thiết lập này có cơ hội truy cập cache ở trên bộ nhớ chính và vì thế, một sự xác định được tạo ra khi cần thiết có nhu cầu.

Đối với việc đọc nhờ đơn vị truy cập bộ nhớ trực tiếp từ bộ nhớ chính, người ta có thể áp dụng một trong hai chiến lược nói trên, mà chúng có giá trị ở giải pháp thứ nhất trong thời gian làm việc của bộ nhớ vi xử lý, hay chúng dẫn tới sự bất định của dữ liệu ở giải pháp thứ hai, nếu việc chuyển tới truy cập bộ nhớ trực tiếp xảy ra trước khi sao chép trở lại bộ nhớ chính. Đối với sự chịu tải của bộ xử lý, một chi phí gia tăng là thích đáng, khi đó, không chỉ việc viết mà cả việc đọc của bộ nhớ chính đều được các thiết bị phụ giám sát. Nếu một địa chỉ được cảm nhận và số liệu của nó tồn tại trong cache, do đó, cache đáp lại và thông báo giá trị đúng tới bộ vi xử lý truy cập trực tiếp.

Trong hệ điều hành, các chiến lược phần cứng được thực hiện khác nhau, thí dụ người ta có thể tiến hành nhờ việc truy cập bộ nhớ không đồng dạng có kết nối cache, phương pháp này có tên là cc-NUMA (cache coherent non uniform memory access). Về điều này, hệ điều hành

phải tự điều chỉnh. Nghĩa là, muốn nâng cao hiệu dụng của cache phải được lưu ý tới việc dự kiến các hoạt động của cache; vì vậy, ta thấy, các hiệu dụng này không phải là không có điều gì đối với các phần mềm. Ví dụ, trạng thái của bộ nhớ chia sẻ ở trong hệ thống đa vi xử lý không phải các thiết bị viết qua hay sao chép lùi luôn luôn tồn tại. Bấy giờ sự trao đổi thông tin (thí dụ giữa một trình gỡ rối và một tiến trình cảnh giới) nhờ bộ nhớ chia sẻ có quan tâm tới điều đó và sau khi hoạt động, ở trong mỗi lập trình, các dữ liệu của tiến trình phải tìm kiếm và được điều chỉnh mới. Ở đây, các cơ chế được thực thi bằng các phần cứng đã tạo nên sự trợ giúp quan trọng.

Thí dụ về chiến lược MESI của hảng Intel:

Ở các hệ thống đa vi xử lý đối xứng, đối với việc truy cập đồng thời các bộ vi xử lý, tỷ phần của bus hệ thống rất đắt, vì vậy nó không được dùng tới. Ở đây, hạn chế này được bù đắp bằng cách: ở trong bộ nhớ cache có xử lý đặc biệt, các dữ liệu của người sử dụng được giữ gìn và được xử lý cục bộ. Đối với các bộ nhớ cache có tồn tại các dữ liệu toàn cục, chúng được giải quyết theo chiến lược MESI của hảng Intel như sau: Ở trong cache, mỗi đơn vị bộ nhớ có một từ trạng thái (statusword), từ này biểu thị cho 4 bậc thuận tiện M (modified), E (execlusive), S (shared) và I (invalid). Mỗi cache chứa đựng một bộ cảnh giới (snooper) dùng để kiểm tra số liệu ở trong cache, các số liệu sẽ được bộ vi xử lý khác làm thay đổi ở trong bộ nhớ chia sẻ. Đầu tiên, các yêu cầu về số liệu được kiểm tra, liệu số liệu đã điều chỉnh đúng chưa (?). Nếu chưa đúng, nó phải được bộ nhớ toàn cục tu chỉnh mới. Nếu đúng rồi, thì thôi. Điều quan trọng là, trước hết, phần mềm không nhận theo cái đó, do đó, đối với hệ điều hành cũng như đối với các chương trình của người sử dụng, các bộ nhớ cache toàn cục hoạt động một cách tốt đẹp.

Ngoài ra, còn có một số chiến lược về bộ nhớ cache nữa mà ở đây chưa cần thiết phải đề cập.

Vấn đề bộ nhớ cache rất tức thời không chỉ đối với các hệ thống đa vi xử lý, mà còn cấp bách cả đối với hệ thống mạng máy tính. Ở đó có rất nhiều bộ đệm (buffer), chúng tác dụng như những bộ nhớ cache. Vì vậy, nếu không được quan tâm bởi hệ thống vận chuyển thông tin hay bởi hệ điều hành, chúng sẽ dẫn tới sự không bện vững của dữ liệu một cách dễ dàng.

3.6. Cơ chế bảo vệ bộ nhớ

Một trong các nhiệm vụ quan trọng là làm cô lập các chương trình với nhau để loại trừ các lỗi, các tính chất sai trái và các xung đột cố ý giữa các người sử dụng cũng như giữa các tiến trình của chúng.

Ở mô hình một bộ nhớ các cung đoạn ảo, chúng ta có trong tay nhiều phương tiện bảo vệ khác nhau. Ưu điểm của việc định vị địa chỉ bộ nhớ ảo cũng như của việc định vị địa chỉ bộ nhớ vật lý đem lại cho chúng ta nhiều khả năng để bảo vệ bộ nhớ:

- Phải đạt được một sự cách lý hoàn toàn của không gian địa chỉ của các tiến trình với nhau. Vì mỗi tiến trình có không gian địa chỉ như nhau nhưng bộ nhớ vật lý thì khác nhau, tức là các tiến trình không chồng chéo lên nhau và cũng không ảnh hưởng lẫn nhau.
- Phải sắp xếp luật truy cập xác định mỗi khoảng bộ nhớ (có thể đọc, có thể viết và có thể thực hiện) một cách kiên cố và điều đó phải được bảo đảm bởi các phần cứng. Thí dụ, nếu

chương trình được thực nghiệm để tự viết đè lên mã chương trình (xảy ra do lỗi bộ nhớ, lỗi lập trình, lỗi biên dịch hay virus), do đó, nó sẽ dẫn tới một sự tổn thất các luật lệ bảo vệ các segment, và với cái đó, dẫn tới một sự vi phạm việc phân đoạn (segmentation- violation). Ngay cả việc sử dụng các bộ chỉ thị (pointer) ở trong giới hạn của nó thì có thể kiểm tra được. Một sự gây ảnh hưởng có chủ ý hay không có chủ ý tới các chương trình khác và tới hệ điều hành đều phải được ngăn cản lại.

- Phải định nghĩa một cách khác nhau những luật truy cập đơn giản đối với các người sử dụng khác nhau, và do đó, phải thực hiện việc điều khiển lối dẫn vào một cách hoàn thiện.

3.6.1. Bảo vệ bộ nhớ ở trong Unix

Ở hệ điều hành Unix có rất nhiều cơ chế bảo vệ bộ nhớ được mô tả và rất tiện dụng. Do đó ở đây không cần phải nêu thêm những bản phác thảo về bảo vệ bộ nhớ ở trong Unix.

3.6.2. Bảo vệ bộ nhớ trong Windows NT

Các cơ cấu bảo vệ bộ nhớ ở trong hệ điều hành Windows NT được điền đầy bởi nhiều tính chất thích hợp cho bộ nhớ ảo, chẳng hạn việc cách ly các tiến trình một cách khác nhau nhờ sự tách chia các không gian địa chỉ, nhờ việc dẫn tới trạng thái người sử dụng, mà ở đó, cho phép người ta truy cập lên tất cả các khoảng có thể của bộ nhớ; cũng như cho phép chuyên môn hoá các luật truy cập (Read/Write/No) cho trạng thái người sử dụng và trạng thái nhân hệ điều hành.

Thêm vào đó, công việc kiểm tra cơ cấu bảo vệ cũng được nêu ra khi một tiến trình tìm kiếm một lối vào bộ nhớ dùng riêng hay dùng chung. Vấn đề khó khăn là việc thực thi cơ cấu bảo vệ bằng tay, các cơ cấu này có sự khác biệt nhau rất lớn, mà sự khác biệt này tồn tại bởi hình dạng ổ đĩa cứng khác nhau (thí dụ bộ vi xử lý R4000 khác với bộ vi xử lý Intel Pentium, vì nó được Windows NT version 4 trợ giúp): cơ cấu bảo vệ bộ nhớ ở trong Windows NT đã được trợ giúp bởi phần cứng. Với bộ vi xử lý R4000 và ở trạng thái người sử dụng, người ta có thể truy cập trên dưới 2GB, nếu không sẽ xảy ra lỗi truy cập (access violation).

Bấy giờ, nhà điều hành bộ nhớ ảo làm việc trên các cơ cấu phần cứng và tìm lỗi trang. Đối với các luật truy cập đơn giản (Read only, Read/Write), anh ta chờ đợi các thông tin execute only (chỉ thực thi), guard pape (phòng vệ trang), no access (không truy cập) và copy on write (copy khi viết); những thông tin này có thể được đặt vào thủ tục với lệnh VirtualProtect(). Sau đây, những thông tin này được giải thích cặn kẻ như sau:

- *Execute only* (chỉ thực thi):

Để tránh những thay đổi và không cho phép việc copy của các chương trình nguồn, ở mã chương trình được sử dụng, việc viết và đọc dữ liệu thì rất quan trọng. Vì điều đó không được phần cứng trợ giúp, nó giống hệt trường hợp Readonly.

- *Guard Page* (phòng vệ trang):

Khi tiến hành truy cập trang, nếu có một trang được đánh dấu, do đó, tiến trình người sử dụng sẽ phát sinh một lỗi phân đoạn, còn gọi là ngoại lệ cảnh giới trang (guard page

exception). Điều đó tạo cho hệ thống con hay cho một tiến trình một trường năng động, mà sau khi trang được đánh dấu thì các phần tử của trường được truy cập. Nghĩa là, sau khi lỗi phân đoạn qua đi, tiến trình truy cập trang trở lại một cách bình thường.

Một thí dụ hay về điều đó là cơ cấu để điều chỉnh độ lớn của stack. Độ lớn này được gia tăng với sự trợ giúp của một ngoại lệ cảnh giới trang một cách năng động khi có nhu cầu.

- *No access* (không truy cập):

Việc đánh dấu trang sẽ được ngăn ngừa, khi người ta truy cập các trang bị cấm hay các trang tồn tại.Tình trạng này được áp dụng hầu hết để nhận được sự trợ giúp hay để thu gom lỗi khi gỡ rối (debugging).

- *Copy on write* (copy khi viết):

Cơ cấu để giải quyết vấn đề copy on write được giải thích ở cuối mục 3.3.6; cơ cấu này có thể được sử dụng để bảo vệ các khoảng bộ nhớ. Nếu một khoảng bộ nhớ dùng chung của các tiến trình được trở thành khoảng bộ nhớ dùng riêng, khi đó, một sự sắp xếp ở trong gian địa chỉ ảo (xem hình 3.9) được tiến hành; do vậy, các trang ảo của tiến trình được coi là copy on write. Bây giờ, một tiến trình tiến hành viết một trang như thế, do đó, đầu tiên việc copy được tạo lập (với các luật đọc/ viết mà không cần luật copy on write ở trên trang) và sau đó, tác vụ được thực hiện. Bấy giờ, tất cả việc tiếp theo đối với một trang diễn ra trên bản copy riêng lẻ và không cần trên bản gốc.

Một cơ chế bảo vệ bộ nhớ quan trọng tiếp theo là việc dập bỏ nội dung trang, phương pháp này đòi hỏi cấp bảo vệ đúng yêu cầu, trước khi một trang được sử dụng cho một tiến trình người sử dụng. Ở sự quá độ của một trang từ trạng thái free (trống) đến trạng thái zeroed (điền đầy), điều nói trên được nhà quản lý bộ nhớ ảo thực hiện.

3.6.3 Các cấp bảo vệ

Một bản phác thảo đưa ra một phương pháp bảo vệ bộ nhớ bổ sung để dẫn tới các bước bảo vệ đối với người sử dụng. Những trạng thái được nói trong chương 1 như trạng thái người sử dụng, trạng thái nhân hệ điều hành là một sự phân chia như thế. Chúng bao gồm 2 bậc chính:

- Bậc đặc quyền: Bậc này dành cho những tiến trình làm việc ở trạng thái nhân hệ điều hành (kernel mode). Trong đó, mã bảo vệ được phép tất cả, thí dụ được phép truy cập tất cả địa chỉ của bộ nhớ.
- Bậc không đặc quyền: Bậc này dành cho những tiến trình làm việc ở trạng thái người sử dụng (used mode). Trong đó, một tiến trình không thể truy cập trên hệ điều hành và trên tất cả các tiến trình khác.

Mỗi bậc bảo vệ bộ nhớ này còn được tiếp tục chia nhỏ nữa, tại đó, các bậc nhỏ này còn được sắp xếp theo độ tin cậy, nó được người ta chỉ ra ở trong từng đại diện mỗi bậc nhỏ vừa nói. Tác dụng của nó phụ thuộc mạnh mẽ vào sự sắp xếp này khi chúng được sự trợ giúp của phần cứng.

Thí dụ về các bậc bảo vệ ở bộ vi xử lý Intel 80386:

Dãy mở rộng các bộ vi xử lý 80x86 chế ngự nhiều bậc bảo vệ ở các phiên bản ngày càng hiện đại. Nếu bộ vi xử lý được đặt trở lại nhờ tín hiệu điện, do đó, nó tồn tại ở trạng thái thực (read mode) và với điều đó, bộ vi xử lý 8086 hay 80186 thì tương thích. Không gian địa chỉ hữu ích thì đồng nhất với không gian địa chỉ vật lý, do đó, các tiến trình có thể quấy nhiễu lẫn nhau. Nói chung khi đó, hệ điều hành được điều chỉnh theo phương pháp reset (đặt lại). Hệ điều hành tạo ra những bảng theo kiểu bảo vệ bộ nhớ này. Sau đó, nó đặt với các Bit trạng thái ảo (vitual mode-Bit), nếu các Bits này không thể được đặt trở lại, do đó, cần tới sự tác động của bộ nhớ MMU. Tất cả việc truy cập bộ nhớ tiếp theo được sử dụng địa chỉ ảo, bấy giờ, đầu tiên chúng phải được biên dịch, tức là, tất cả các tiến trình đã được dự kiến được cách biệt với nhau ở trong mỗi không gian địa chỉ ảo.

Với kiểu địa chỉ ảo, *bộ vi xử lý 80386 có 4 bậc bảo vệ*: bậc 3 cho chương trình người sử dụng, bậc 2 cho thư viện dùng chung, bậc 1 cho gọi hệ thống và bậc 0 cho kiểu nhân hệ điều hành. Hai Bit để biểu diễn bậc được thực hiện với tư cách là thông tin truy cập ở việc mô tả các trang và các segments, và chúng còn quyết định về luật lệ truy cập. Nếu việc truy cập bị từ chối, do đó, dẫn tới việc bẻ gãy lỗi.

Các bước nhãy chương trình và việc gọi thủ tục ở mỗi mã của một bậc khác nhau được điều chỉnh một cách mạnh mẽ. Để gọi thủ tục của một bậc khác nhau thì một lệnh đặc biệt CALL được sử dụng, lệnh này kiểm tra việc truy cập về cấu trúc dữ liệu có dạng call gate và sau đó sử dụng các địa chỉ bước nhãy được tạo ra trước đó ở trong thủ tục. Do đó, những bước nhãy chưa được kiểm tra sẽ bị loại trừ.

3.7. Các bài tập của chương 3

3.7.1. Các bài tập về che phủ bộ nhớ

Bài tập 3.1. Các chiến lược che phủ

Nếu cho một hệ thống tráo đổi (swappingsystem), mà bộ nhớ của nó bao gồm các khoảng trống theo thứ tự như sau: 10kB, 4kB, 20kB, 18kB, 7kB, 9kB, 12kB, và 15kB. Khoảng trống nào sẽ được chọn đầu tiên cho vừa vặn tốt nhất khi yêu cầu cần nạp các không gian bộ nhớ là 12kB, 10kB, 9kB. Bạn hãy nhắc lại các yêu cầu cho các trường hợp tốt nhất, tồi nhất và tạm được (kể cả bản vẽ sơ đồ).

3.7.2. Các bài tập bộ nhớ ảo

Bài tập 3.2. Xác định địa chỉ

a). Bạn hãy tính toán các địa chỉ vật lý từ các địa chỉ ảo 2204_H và $A226_H$. Độ dịch vị là 13 Bit và số trang là 3 Bit. Bạn hãy sử dụng bảng trang sau đây:

Trang	0	1	2	3	4	5	6	7
Khung trang	7	0	1	6	4	2	3	5

b). Việc quản lý bộ nhớ phải được mở rộng như thế nào, mà nhờ đó, trang được cập phát?

Bài tập 3.3. Về các bảng địa chỉ

Một thiết bị địa chỉ ảo 128 Bit và địa chỉ vật lý 32 Bit. Các trang có giá trị 8 kilo- từ (kilo-word:kW).

a). Có bao nhiêu lần điền vào được sử dụng cho một bảng trang thông thường?

b) Có bao nhiêu bậc được sử dụng cho bảng trang kiểu nhiều bậc (mức), để tồn tại với một kích cở bảng trang nhỏ hơn 1 mêga-từ (mega-word: mW)? Ở đây, người ta hiểu 1 word = 1 sự điền vào.

Bài tập 3.4. Về bảng trang đa bậc

Một thiết bị có một không gian địa chỉ ảo. Việc quản lý bộ nhớ được sử dụng một bảng trang 2 bậc đối với một bộ nhớ Cache (được liên kết lại), được gọi là bộ đệm dịch chuyển phía nhìn thấy (translation lookside buffer: TLB) với một tỷ phần gặp nhau trung bình khoảng 90%. Bạn hãy lưu ý rằng, một sự truy cập có thể thực hiện bằng hai cách: sử dụng bộ đệm TLB hay nhờ bảng trang.

a). Sự tiêu phí thời gian trung bình trên bộ nhớ chính bằng bao nhiêu, nếu thời gian truy cập bộ nhớ là 100ns và thời gian truy cập bộ đệm TLB là 10ns ? Giả sử rằng không có lỗi trang xuất hiện.

b). Ở câu (a) các lỗi trang được bỏ qua. Bây giờ, chúng ta muốn nghiên cứu trường hợp đơn giản, mà trong đó các lỗi trang xuất hiện chỉ khi truy cập bộ nhớ chính. Giả sử các bảng trang đều ở trong bộ nhớ chính và không được cấp phát ! Khi đó tần suất của các lỗi trang là $1:10^5$ và một lỗi trang mất 100ns. Vậy thời gian truy cập trung bình trên bộ nhớ chính là như thế nào?

Gợi ý: Bạn hãy áp dụng biểu thức quan hệ sau đây:

Thời gian truy cập trung bình = Thời gian gặp nhau + Tỷ phần truy cập hỏng x Thời gian truy cập hỏng.

c). Việc điều hành đa chương trình gây ra những vấn đề gì đối với bộ đệm TLB và đối với bộ nhớ Cache khác ? Trước hết, bạn hãy nghĩ tới ở sự nhận dạng các khối.

3.7.3. Các bài tập về quản lý trang và bộ nhớ Cache

Bài tập 3.5. Về các chuỗi tham chiếu

Cho một chuỗi tham chiếu sau đây: 0 1 2 3 3 2 3 1 5 2 1 3 2 5 6 7 6 5. Mỗi lần truy cập một trang xảy ra trong khoảng một đơn vị thời gian.

a). Hỏi có bao nhiêu lỗi trang của tập Working set dẫn ra đối với một kích cở của cửa sổ h =3 ?

b). Nó sẽ dẫn tới vấn đề chiến lược mở rộng nào, khi một trang mới được nạp vào trong tập working set? Bạn hãy suy nghĩ, khi nào thì có lợi để thiết đặt một trang của một tập Working

set, và khi nào thì có lợi để thêm vào một trang và để thay đổi kích cở cửa sổ h một cách năng động; trường hợp kích cở lớn nhất của tập Working set vẫn chưa được đánh giá !

Bài tập 3.6. Về các chiến lược cấp phát

Có một máy tính chiếm 4 khung trang. Tại thời điểm nạp của lần truy cập sau cùng, các giá trị M và R (Bit) đối với mỗi trang được dẫn ra như sau (thời gian tính bằng tiếng kêu tích tắc của đồng hồ):

a). Trang nào được chiến lược NRU thay thế ?

b). Trang nào được chiến lược FIFO thay thế ?

c). Trang nào được chiến lược LRU thay thế ?

Bài tập 3.7. Về tập Working set

a). Dưới khái niệm Working set của một chương trình, người ta hiểu cái gì ? Nó có thể thay đổi như thế nào ?

b). Việc gia tăng số lượng các khung trang ở bộ nhớ công tác có hiệu quả nào trong quan hệ với số lượng các lỗi trang ? Điều đó mang lại hiệu quả nào trên các bảng trang ?

Bài tập 3.8. Về hiệu ứng Thrashing

a). Cái gì gây nên hiệu ứng thrashing ?

b). Hệ điều hành có thể phát hiện hiệu ứng thrashing như thế nào và nó làm gì để chống lại hiệu ứng này ?

c). Một hệ thống có những thông số hệ thống sau đây:

1). s =3; v = 14/15, $\sigma_1 = 0,5$; $\rho_1 = 1/15$; $\rho_w = 1/20$

2). s =3; v= 14/15; $\sigma_1 = 0,5$; $\rho_1 = 1/15$; $\rho_w = 1/10$

Khi số lượng các tiến trình gia tăng, tính chất tải như thế nào ?

Tại chỗ nào có sự gia tăng đột biến ?

Bài tập 3.9. Về bộ nhớ Cache

Bộ nhớ Cache có thời gian truy cập 50ns. Trong thời gian này, bộ vi xử lý có thể truy cập trên bộ nhớ Cache không có chu trình chờ đợi và trên bộ nhớ chính với 3 chu trình chờ đợi. Tỷ phần gặp nhau của việc truy cập trên bộ nhớ Cache là 80%. Thời gian của chu trình hướng bus của bộ vi xử lý là 50ns. Bạn hãy tính:

a). Thời gian truy cập trung bình của ca làm việc ?

b). Số lượng trung bình các chu trình chờ đợi cần thiết ?