Lời mở đầu

Vài năm qua, Linux đã thực sự tạo ra một cuộc cách mạng trong lĩnh vực máy tính. Sự phát triển và những gì chúng mang lại cho máy tính thật đáng kinh ngạc: một hệ điều hành đa nhiệm, đa người dùng. Linux có thể chạy trên nhiều bộ vi xử lý khác nhau như: Intel , Motorola , MC68K , Dec Alpha. Nó tương tác tốt với các hệ điều hành: Apple , Microsoft và Novell.

Không phải ngẫu nhiên mà ngành công nghệ thông tin Việt Nam chọn Linux làm hệ điều hành nền cho các chương trình ứng dụng chủ đạo về kinh tế và quốc phòng. Với mã nguồn mở, sử dụng Linux an toàn hơn các ứng dụng Windows. Linux đem đến cho chúng ta lợi ích về kinh tế với rất nhiều phần mềm miễn phí. Mã nguồn mở của hệ điều hành và của các chương trình trên Linux là tài liệu vô giá để chúng ta học hỏi về kỹ thuật lập trình vốn là những tài liệu không được công bố đối với các ứng dụng Windows.

Trong đồ án này, chúng ta sẽ tìm hiểu một phần rất quan trọng trong hệ điều hành Linux đó là: **quản lý bộ nhớ trong Linux.** Một hệ điều hành muốn chạy ổn định thì phải có một cơ chế quản lý bộ nhớ hiệu quả. Cơ chế này sẽ được trình bày một cách chi tiết trong đồ án và có kèm theo các chương trình minh họa.

fanguoshou

MỤC LỤC

CHƯƠNG I

MỞ ĐẦU

I. Giới thiệu về hệ điều hành Linux

Linux là một hệ điều hành họ UNIX miễn phí được sử dụng rộng rãi hiện nay. Được viết vào năm 1991 bởi Linus Toward, hệ điều hành Linux đã thu được những thành công nhất định. Là một hệ điều hành đa nhiệm, đa người dùng, Linux có thể chạy trên nhiều nền phần cứng khác nhau. Với tính năng ổn định và mềm dẻo, Linux đang dần được sử dụng nhiều trên các máy chủ cũng như các máy trạm trong các mạng máy tính. Linux còn cho phép dễ dàng thực hiện việc tích hợp nó và các hệ điều hành khác trong một mạng máy tính như Windows, Novell, Apple ... Ngoài ra, với tính năng mã nguồn mở, hệ điều hành này còn cho phép khả năng tùy biến cao, thích hợp cho các nhu cầu sử dụng cụ thể.

II. Tổng quan về quản lý bộ nhớ trong Linux

Trong hệ thống máy tính, bộ nhớ là một tài nguyên khan hiếm. Cho dù có bao nhiêu bộ nhớ đi chăng nữa thì vẫn không đáp ứng đủ nhu cầu của người sử dụng. Các máy tính cá nhân hiện nay đã trang bị ít nhất 128Mb bộ nhớ. Các máy chủ server có thể lên đến hàng gigabyte bộ nhớ. Thế nhưng nhu cầu bộ nhớ vẫn không được thỏa mãn.

Linux có cách tiếp cận và quản lý bộ nhớ rất rõ ràng. Các ứng dụng trên Linux không bao giờ được phép truy cập trực tiếp vào địa chỉ vật lý của bộ nhớ. Linux cung cấp cho các chương trình chạy dưới HĐH - còn gọi là tiến trình - một mô hình đánh địa chỉ phẳng không phân đoạn segment:offset như DOS. Mỗi tiến trình chỉ thấy được một vùng không gian địa chỉ của riêng nó. Hầu như tất cả các phiên bản của UNIX đều cung cấp cách bảo vệ bộ nhớ theo cơ chế bảo đảm không có tiến trình nào có thể ghi đè lên vùng nhớ của tiến trình khác đang hoạt động hoặc vùng nhớ của hệ thống. Nói chung, bộ nhớ mà hệ thống cấp phát cho một tiến trình không thể nào đọc hoặc ghi bởi một tiến trình khác.

Trong hầu hết các hệ thống Linux và UNIX, con trỏ được sử dụng là một số nguyên 32 bit trỏ đến một ô nhớ cụ thể. Với 32 bit, hệ thống có thể đánh địa chỉ lên đến 4 GB bộ nhớ. Mô hình bộ nhớ phẳng này dễ truy xuất và xử lý hơn bộ nhớ phân đoạn segment:offset. Ngoài ra, một vài hệ thống còn sử dụng mô hình địa chỉ 64 bit, như vậy không gian địa chỉ có thể mở rộng ra đến terabyte.

Để tăng dung lượng bộ nhớ sẵn có, Linux còn cài đặt chương trình phân trang đĩa tức là một lượng không gian hoán đổi nào đó có thể phân bố trên đĩa. Khi hệ thống yêu cầu nhiều bộ nhớ vật lý, nó sẽ đưa các trang không hoạt động ra đĩa, nhờ vậy bạn có thể chạy những ứng dụng lớn hơn và cùng

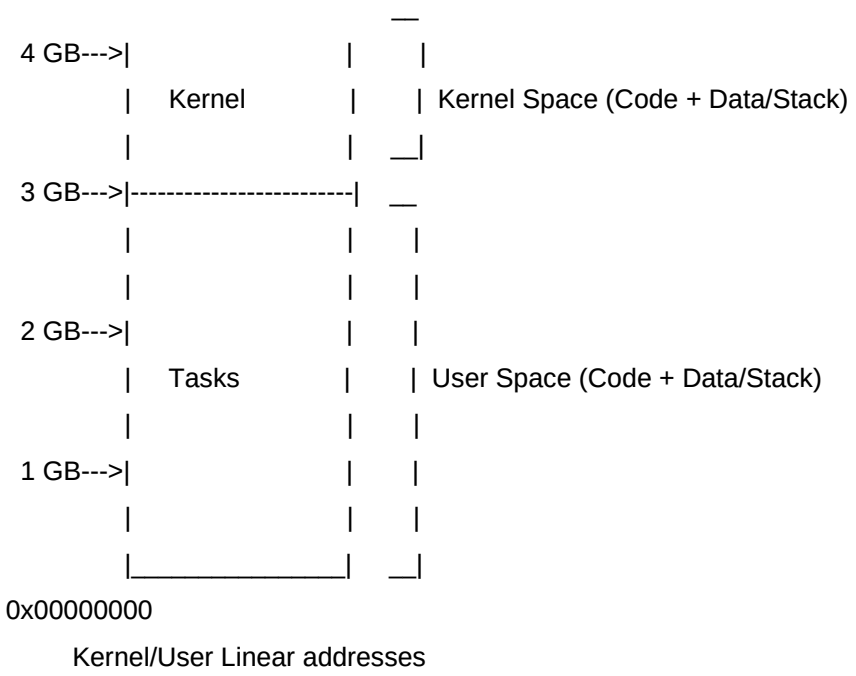
lúc hỗ trợ nhiều người sử dụng. Tuy vậy, việc hoán đổi không thay được RAM vật lý, nó chậm hơn vì cần nhiều thời gian để truy cập đĩa. Kernel cũng cài đặt khối bộ nhớ hợp nhất cho các chương trình người sử dụng và bộ đệm đĩa tạm thời (disk cache). Theo cách này, tất cả bộ nhớ trống dành để nhớ tạm và bộ nhớ đệm (cache) sẽ giảm xuống khi bộ xử lý chạy những chương trình lớn.

CHƯƠNG II

CƠ CHẾ PHÂN ĐOẠN, PHÂN TRANG

I. Sự phân đoạn

Linux sử dụng cơ chế phân đoạn để phân tách các vùng nhớ đã cấp phát cho hạt nhân và các tiến trình. Hai phân đoạn liên quan đến 3GB đầu tiên (từ 0 đến 0xBFFF FFFF) của không gian địa chỉ tiến trình và các nội dung của chúng có thể được đọc và chỉnh sửa trong chế độ người dùng và trong chế độ Kernel. Hai phân đoạn liên quan đến GB thứ 4 (từ 0xC000 0000 đến 0xFFFF FFFF) của không gian địa chỉ tiến trình và các nội dung của nó có thể được đọc và chỉnh sửa duy nhất trong chế độ Kernel. Theo cách này, dữ liệu và mã Kernel được bảo vệ khỏi sự truy cập không hợp lý của các tiến trình chế độ người dùng.



Kernel/User Linear addresses

II. Sự phân trang

1. Nhu cầu phân trang

Vì có quá ít bộ nhớ vật lý so với bộ nhớ ảo nên HĐH phải chú trọng làm sao để không lãng phí bộ nhớ vật lý. Một cách để tiết kiệm bộ nhớ vật lý là chỉ *load* những trang ảo mà hiện đang được sử dụng bởi một chương trình đang thực thi. Ví dụ, một chương trình cơ sở dữ liệu thực hiện một truy vấn vào cơ sở dữ liệu. Trong trường hợp này không phải toàn bộ cơ sở dữ liệu được *load* vào bộ nhớ mà chỉ load những bản ghi có liên quan. Việc mà chỉ load những trang ảo vào bộ nhớ khi chúng được truy cập dẫn đến nhu cầu về phân trang.

2. Trang lưu trữ (page cache)

Cache là tầng nằm giữa phần quản lý bộ nhớ kernel và phần vào ra của đĩa. Các trang mà kernel hoán đổi không được ghi trực tiếp lên đĩa mà được ghi vào cache. Khi cần vùng nhớ trống thì kernel mới ghi các trang từ cache ra đĩa.

Đặc tính chung của các trang trong danh sách trang theo chuẩn LRU(Least Recently Used : ít sử dụng thường xuyên nhất) là :
- *active_list* : là những trang có page -> age > 0, chứa hoặc không chứa dữ liệu, và có thể được ánh xạ bởi một mục trong bảng trang tiến trình.
- *inactive_dirty_list* : là những trang có page -> age == 0, chứa hoặc không chứa dữ liệu, và không được ánh xạ bởi bất kì một mục nào trong bảng trang tiến trình.
- *inactive_clean_list* : mỗi vùng có inactive_dirty_list của riêng nó, chứa các trang clean với age == 0, và không được ánh xạ bởi bất kì một mục nào trong bảng trang tiến trình.

Trong khi quản lý lỗi trang, kernel sẽ tìm kiếm các trang lỗi trong page cache. Nếu lỗi được tìm thấy thì nó được đưa đến active_list để đưa ra thông báo.

* Vòng đời của một User Page

1. Trang P được đọc từ đĩa vào bộ nhớ và được lưu vào page cache. Có thể xảy ra một trong các trường hợp sau :

* Tiến trình A muốn truy cập vào trang P. Nó sẽ được trình quản lý lỗi trang kiểm tra xem có tương ứng với file đã được ánh xạ không. Sau đó nó được lưu vào page cache và bảng trang tiến trình. Từ đây vòng đời của trang bắt đầu trên active_list, nơi mà nó vẫn được lưu giữ kể cả khi đang được sử dụng.

hoặc :

* Trang P được đọc trong suốt quá trình hoạt động của đầu đọc hoán đổi, và được lưu vào page cache. Trong trường hợp này, lý do mà trang được đọc đơn giản chỉ vì nó là một phần của cluster trong các khối trên đĩa. Một loạt các trang liên tiếp nhau trên đĩa sẽ được đọc mà không cần biết các trang này có cần hay không. Chúng ta cũng không cần quan tâm đến việc thông báo cho các trang này nếu chúng mất đi khi không dùng, vì chúng có thể phục hồi ngay lập tức cho dù không còn được tham chiếu đến nữa.

hoặc :

* Trang P được đọc trong suốt quá trình hoạt động của đầu đọc cluster ánh xạ bộ nhớ. Trong trường hợp này, một chuỗi các trang liền nhau tiếp sau trang lỗi trong file ánh xạ bộ nhớ được đọc. Những trang này bắt đầu vòng đời của chúng trong page cache kết hợp với file ánh xạ bộ nhớ và trong active_list.

2. Trang P được ghi bởi tiến trình, do đó có chứa dữ liệu (dirty). Lúc này trang P vẫn ở trên active_list.

3. Trang P không được sử dụng trong một thời gian. Sự kích hoạt định kì của hàm ***kswapd()*** (kernel swap daemon) sẽ giảm dần biến đếm page->age. Hàm ***kswapd()*** sẽ hoạt động nhiều hơn khi nhu cầu về bộ nhớ tăng. Thời gian tồn tại của trang P sẽ giảm dần xuống 0 (age == 0) nếu nó không còn được tham chiếu, dẫn đến sự kích hoạt của hàm ***re_fill inactive()***

4. Nếu bộ nhớ đã đầy, hàm ***swap_out*** sẽ được gọi bởi hàm ***kswapd()*** để cố gắng lấy lại các trang từ không gian địa chỉ ảo của tiến trình A. Vì trang P không còn được tham chiếu và có age == 0, nên các mục trong bảng trang sẽ bị xóa. Tất nhiên, trong thời gian này sẽ không có tiến trình nào ánh xạ đến. Hàm ***swap_out*** thực ra không đưa trang P ra ngoài mà đơn giản là chỉ loại bỏ sự tham chiếu của tiến trình đến trang. Nhờ vào page cache và cơ chế swap mà trang sẽ bảo đảm được ghi lên đĩa khi cần.

5. Thời gian xử lý ít hay nhiều là tùy thuộc vào nhu cầu sử dụng bộ nhớ

6. Tiếp theo, hàm ***refill_inactive_scan()*** tìm các trang mà có thể đưa đến inactive_dirty list. Từ khi trang P không được ánh xạ bởi một tiến trình nào và có age == 0 thì nó được đưa từ active_list đến inactive_dirty list.

7. Tiến trình A truy cập vào trang P, nhưng nó hiện không có trong bộ nhớ ảo tiến trình các mục trong bảng trang đã bị xóa bởi hàm swap_out(). Trình điều khiển lỗi gọi hàm find_page_nolock() để xác định vị trí trang P trong page cache. Sau khi tìm thấy, các mục trong bảng trang sẽ được phục hồi ngay lập tức và trang P được đưa đến active_list.

8. Quá trình này mất nhiều thời gian do hàm swap_out() xóa các mục trong bảng trang của tiến trình A, hàm ***refill_inactive_scan()*** vô hiệu hóa trang P, đưa nó đến inactive_dirty list. Việc tốn nhiều thời gian sẽ làm bộ nhớ trở nên chậm.

9. Hàm page_ launder() được kích hoạt để làm sạch các trang dirty. Nó tìm trang P trong inactive_dirty list và ghi trang P ra đĩa. Sau đó, trang được đưa đến inactive_clean_list. Khi hàm page_ launder() thực sự quyết định ghi lên trang thì sẽ thực hiện các bước sau :

- Khóa trang
- Gọi phương thức writepage. Lời gọi này kích hoạt một vài đoạn mã đặc biệt để thực hiện ghi lên đĩa (không đồng bộ) với trang đã bị khóa. Lúc này, hàm page_ launder() đã hoàn thành nhiệm vụ, trang vẫn ở trong inactive_dirty_list và sẽ được mở khóa cho đến khi việc ghi hoàn tất.
- Hàm page_ launder() được gọi lại để tìm trang clean để đưa nó đến inactive_clean_list, giả sử trong thời gian này không có tiến trình nào tham chiếu đến nó trong page cache.

10. Hàm page_ launder() thực hiện lại để tìm các trang không sử dụng và clean, đưa chúng đến inactive_clean_list

11. Giả sử cần một trang trống riêng lẻ. Điều này có thể thực hiện bằng cách lấy lại một trang inactive_clean, trang P sẽ được chọn. Hàm reclaim_page() loại bỏ trang P ra khỏi page cache (điều này bảo đảm rằng không có tiến trình nào khác tham chiếu đến nó trong quá trình quản lý lỗi trang), và nó được đưa cho lời gọi như là một trang trống.

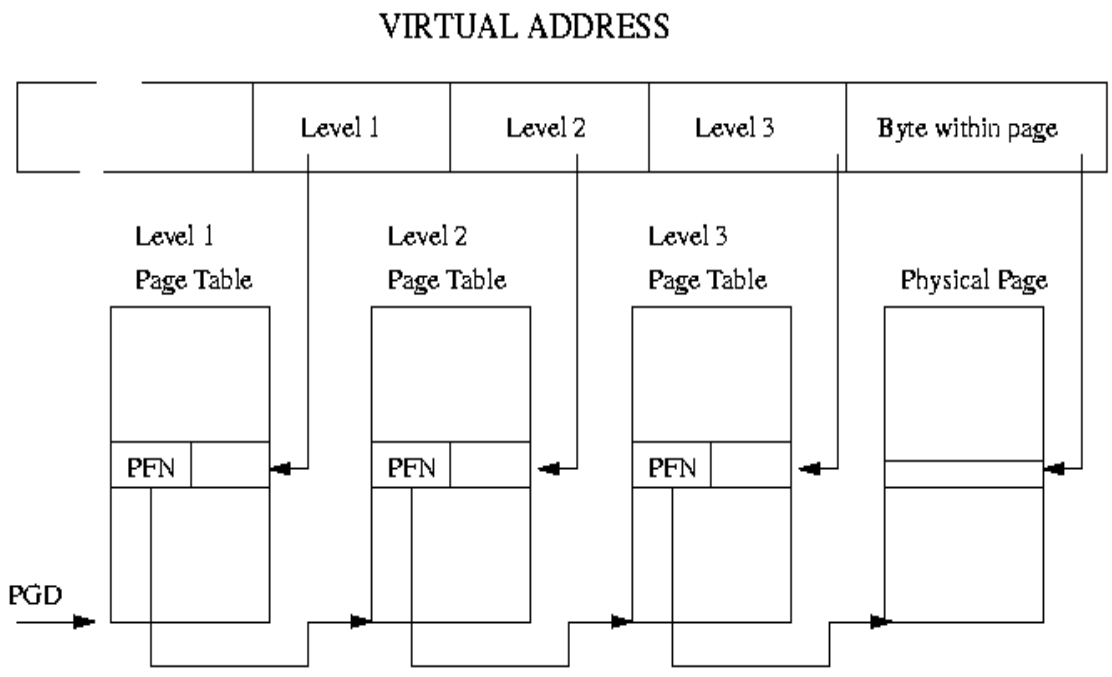
Hoặc :

Hàm kreclaimd() cố gắng tạo bộ nhớ trống. Nó giành lại trang P và xóa nó.

Đây chỉ là một chuỗi các sự kiện hợp lý : một trang có thể sống trong page cache trong một thời gian dài, rồi chết đi, rồi lại được phục hồi trở lại, ... Trang có thể được phục hồi từ inactive_clean, active lists hay inactive_dirty list. Trang chỉ đọc là những trang không phải dirty, vì vậy hàm page_ launder() có thể đưa chúng từ inactive_dirty_list đến inactive_clean_list để làm trống nó.

Các trang trong inactive_clean list được kiểm tra định kì nhằm tạo ra các khối nhớ trống lớn liên tiếp nhau để đáp ứng khi có yêu cầu. Tóm lại, trang P thực chất chỉ là một trang logic, do đó nó được thể hiện bằng một vài trang vật lý cụ thể.

3. Bảng trang (page table)



Hình 1 : 3 mức bảng trang

Linux giả sử rằng có 3 mức bảng trang. Mỗi bảng trang chứa số khung trang của bảng trang ở mức tiếp theo. Hình 1 chỉ ra cách mà địa chỉ ảo được chia

thành các trường. Mỗi trường cung cấp một địa chỉ offset đến một bảng trang cụ thể. Để chuyển địa chỉ ảo thành địa chỉ vật lý, bộ xử lý phải lấy nội dung của các trường rồi chuyển thành địa chỉ offset đến trang vật lý chứa bảng trang và đọc số khung trang của bảng trang ở mức tiếp theo. Việc này lặp lại 3 lần cho đến khi số khung trang của trang vật lý chứa địa chỉ ảo được tìm ra. Bây giờ, trường cuối cùng trong địa chỉ ảo được sử dụng để tìm dữ liệu trong trang.

Mỗi nền mà Linux chạy trên đó phải cung cấp sự chuyển đổi các macro cho phép kernel có thể hiểu được các bảng trang tương ứng trên nền đó. Do đó, kernel không cần biết định dạng của các mục trong bảng trang cũng như cách sắp xếp của nó. Điều này giúp cho Linux thành công trong việc sử dụng cùng một đoạn mã để xử lý các bảng trang đối với bộ xử lý Alpha (có 3 mức bảng trang) và đối với bộ xử lý Intel x86 (có 2 mức bảng trang).

4. Định vị và giải phóng trang

Có nhiều nhu cầu về trang vật lý trong hệ thống. Ví dụ, khi một ảnh được load vào bộ nhớ, HĐH cần định vị trang. Những trang này sẽ được làm trống khi ảnh đã xử lý xong và không còn load nữa. Một công dụng khác của trang vật lý là chứa cấu trúc dữ liệu cụ thể về kernel như cấu trúc của chính các trang này. Cơ chế và cấu trúc dữ liệu được sử dụng để định vị và giải phóng trang có ý nghĩa vô cùng quan trọng trong việc quản lý một cách hiệu quả bộ nhớ ảo.

Tất cả các trang vật lý trong hệ thống được mô tả bởi cấu trúc dữ liệu mem_map, đây là một danh sách gồm các cấu trúc dữ liệu mem_map_t được khởi tạo lúc khởi động. Mỗi mem_map_t mô tả một trang vật lý trong hệ thống. Các trường quan trọng có liên quan đến việc quản lý bộ nhớ là :

- Count : lưu số lượng người sử dụng của trang này. Count > 1 khi trang được chia sẻ bởi nhiều tiến trình
- age : trường này mô tả "tuổi" của trang và được dùng để quyết định trang bị loại bỏ hay hoán đổi
- map_nr : đây là số khung trang vật lý mà mem_map_t này mô tả

Vector free_area được sử dụng bởi đoạn mã định vị trang để tìm và làm trống trang. Toàn bộ lược đồ quản lý bộ đệm được hỗ trợ bởi cơ chế này và các đoạn mã liên quan, còn kích thước của trang và cơ chế phân trang vật lý được sử dụng bởi bộ xử lý thì không liên quan.

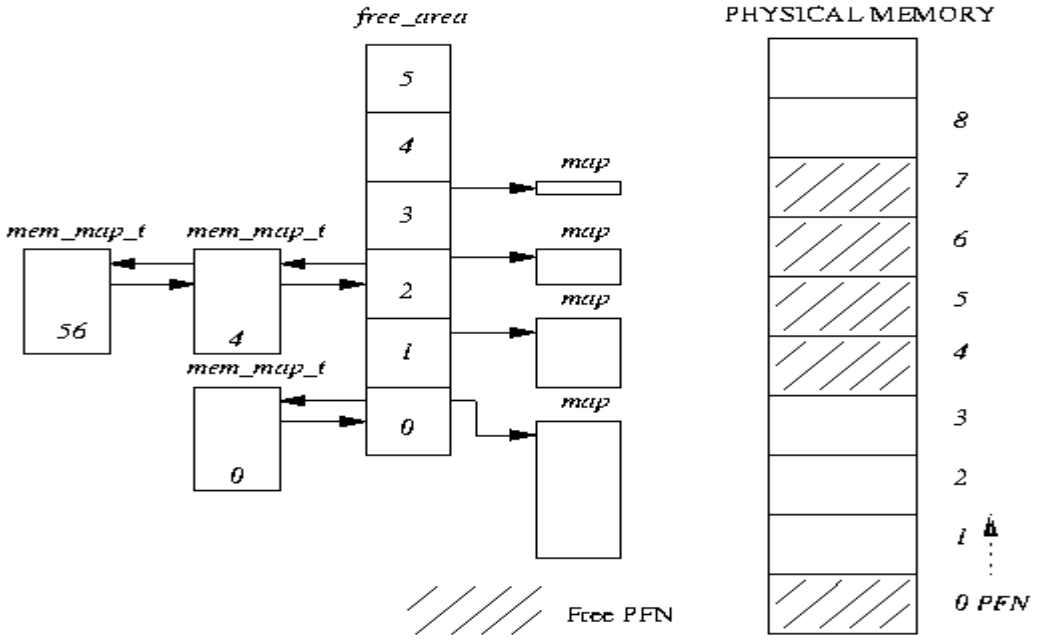
Mỗi phần tử của free_area chứa thông tin về các khối của trang. Phần tử thứ nhất trong mảng mô tả các trang đơn lẻ, các khối gồm 2 trang tiếp theo, các khối gồm 4 trang tiếp theo, và cứ tăng như thế theo lũy thừa 2. Phần tử *list* đứng đầu hàng đợi và trỏ đến cấu trúc dữ liệu *page* trong mảng *mem_map*. Các khối trống của trang được xếp ở đây. Con trỏ *map* trỏ đến ảnh bitmap để theo dõi các nhóm trang đã được định vị theo kích thước như trên. Bit thứ N của ảnh bitmap được thiết lập nếu khối thứ N của trang là trống.

Hình 2 cho thấy cấu trúc của free_area. Phần tử 0 có một trang trống (khung trang số 0), phần tử 2 có 2 khối trống gồm 4 trang (khối đầu tiên ở khung trang số 4, khối thứ hai ở khung trang số 56).

4.1 Định vị trang

Linux sử dụng thuật toán Buddy để định vị và giải phóng một cách hiệu quả các khối của trang. Đoạn mã định vị trang xác định một khối của một hay nhiều trang vật lý. Những trang được định vị trong khối có kích thước là lũy thừa của 2. Điều đó có nghĩa là nó có thể định vị một khối gồm 1 trang, 2 trang, 4 trang,... Khi có đủ số trang trống trong hệ thống để cấp cho một yêu cầu, đoạn mã định vị sẽ tìm trong free_area một khối các trang có kích thước như yêu cầu. Mỗi phần tử của free_area ánh xạ đến các khối trang trống có kích thước tương ứng. Ví dụ, phần tử thứ 2 của mảng ánh xạ đến các khối gồm 4 trang trống đã được định vị.

Trước hết, thuật toán định vị tìm các khối trang có kích thước như yêu cầu. Nó tìm theo một chuỗi các trang trống đã được sắp xếp trong phần tử *list* của free_area. Nếu không có khối trang trống có kích thước như yêu cầu thì các khối có kích thước tiếp theo (gấp đôi kích thước yêu cầu) sẽ được tìm. Tiến trình này sẽ tiếp tục cho đến khi tất cả free_area được tìm hoặc một khối trang nào đó được tìm thấy. Nếu khối trang tìm thấy lớn hơn kích thước yêu cầu thì nó phải chia nhỏ ra cho đến khi có một khối đúng kích thước. Bởi vì mỗi khối có số trang là lũy thừa của 2 nên việc chia nhỏ được thực hiện một cách dễ dàng bằng cách chia đôi khối. Phần trống của khối được đưa vào hàng đợi tương ứng, phần còn lại được cung cấp cho lời gọi.



Hình 2 : Cấu trúc dữ liệu của free_area

Ví dụ, trong hình 2, nếu một khối gồm 2 trang được yêu cầu thì khối 4 trang thứ nhất (bắt đầu ở khung trang số 4) sẽ được chia thành hai khối 2 trang. Khối thứ nhất (bắt đầu ở khung trang số 4) sẽ được cung cấp cho lời gọi và khối thứ hai (bắt đầu ở khung trang số 6) sẽ được đưa vào hàng đợi như là một khối 2 trang trống ở phần tử thứ nhất của mảng free_area.

4.2 Giải phóng trang

Việc định vị các khối trang làm cho bộ nhớ bị phân mảnh do các khối trang lớn bị chia nhỏ. Đoạn mã giải phóng trang kết hợp các trang lại thành một khối lớn các trang trống bất cứ khi nào có thể. Khi có một khối trang trống thì các khối lân cận có cùng kích thước được kiểm tra xem có trống không. Nếu có thì chúng được kết hợp với nhau để tạo ra một khối trang có kích thước gấp đôi. Đoạn mã giải phóng trang lại tìm cách kết hợp khối mới này với một khối khác. Theo cách này, khối các trang trống sẽ lớn dần.

Ví dụ, trong hình 2, nếu khung trang số 1 trống thì nó sẽ được kết hợp với khung trang số 0 đã trống sẵn để tạo thành một khối 2 trang và được xếp vào phần tử thứ nhất của free_area.

CHƯƠNG III

CƠ CHẾ QUẢN LÝ BỘ NHỚ ẢO

I. Khái niệm bộ nhớ ảo, không gian hoán đổi

Linux hỗ trợ bộ nhớ ảo, nghĩa là nó sử dụng một phần của đĩa như là RAM để tăng kích thước của bộ nhớ. Kernel sẽ ghi nội dung của một khối nhớ hiện không sử dụng lên đĩa cứng để bộ nhớ được sử dụng cho mục đích khác. Khi cần lại những nội dung này thì chúng sẽ được đọc trở lại vào bộ nhớ. Việc này hoàn toàn trong suốt đối với người sử dụng, các chương trình chạy trong Linux chỉ thấy một số lượng lớn bộ nhớ có sẵn mà không quan tâm rằng những phần đó nằm trên đĩa. Tất nhiên, việc đọc và ghi lên đĩa thì chậm hơn (khoảng một ngàn lần) so với sử dụng bộ nhớ thật, vì vậy chương trình chạy không nhanh. Phần đĩa cứng được sử dụng như là bộ nhớ ảo được gọi là không gian hoán đổi.

Linux có thể sử dụng môt file thông thường trong file hệ thống hoặc một phân vùng riêng để làm không gian hoán đổi. Một phân vùng swap thì nhanh hơn nhưng lại dễ hơn trong việc thay đổi kích thước của một file swap. Khi bạn biết mình cần bao nhiêu không gian hoán đổi thì bạn bắt đầu tạo một phân vùng swap, nhưng nếu bạn không chắc thì bạn nên sử dụng một file swap trước, sử dụng hệ thống trong một thời gian để biết chắc không gian hoán đổi mà mình cần rồi sau đó mới tạo phân vùng swap.

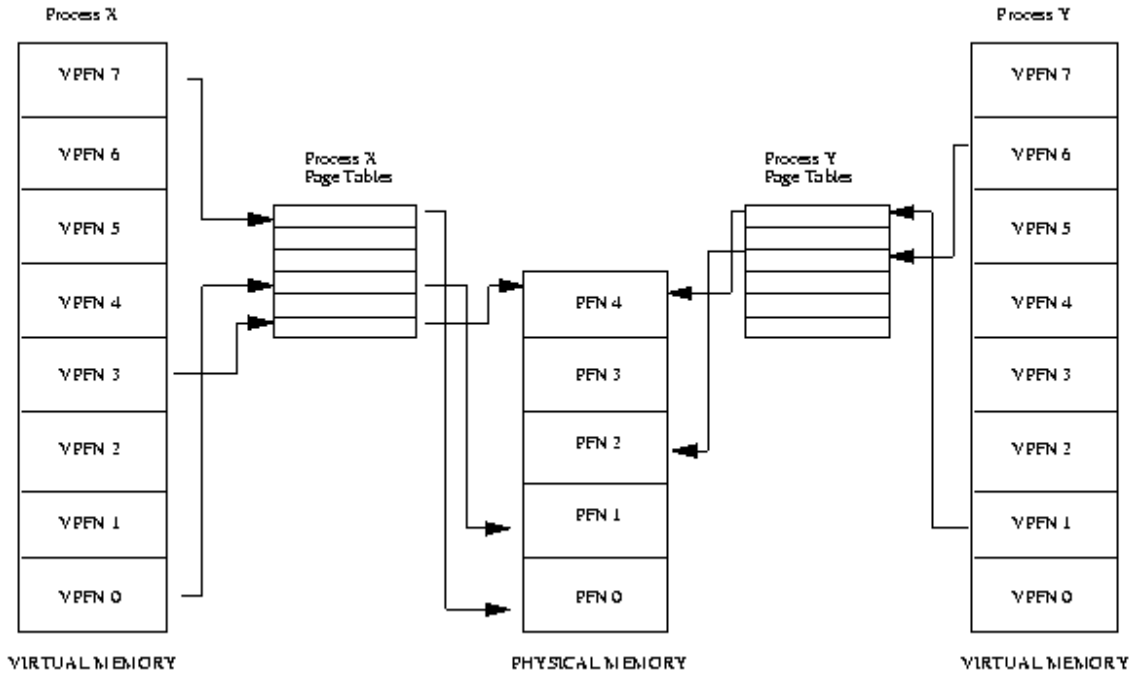
II. Mô hình bộ nhớ ảo

Trước khi tìm hiểu các phương thức mà Linux sử dụng để hỗ trợ bộ nhớ ảo, chúng ta nên tìm hiểu một ít về mô hình trừu tượng của nó.

Khi bộ xử lý thực hiện một chương trình, nó đọc một chỉ lệnh từ bộ nhớ và giải mã chỉ lệnh đó. Trong khi giải mã chỉ lệnh, nó có thể lấy về hay lưu trữ nội dung của một vị trí trong bộ nhớ. Sau đó bộ xử lý sẽ thực hiện chỉ lệnh và di chuyển đến chỉ lệnh tiếp theo trong chương trình. Theo cách này, bộ xử lý sẽ luôn luôn truy cập bộ nhớ để lấy chỉ lệnh về hoặc lấy và lưu trữ dữ liệu.

Tất cả các địa chỉ trong bộ nhớ ảo là địa chỉ ảo chứ không phải địa chỉ vật lý. Bộ xử lý chuyển những địa chỉ ảo này thành địa chỉ vật lý dựa vào thông tin trong các bảng được quản lý bởi HĐH.

Để cho sự chuyển đổi dễ dàng hơn thì bộ nhớ ảo và bộ nhớ vật lý được chia thành nhiều khúc có kích thước thích hợp gọi là trang. Tất cả các trang này có cùng kích thước để dễ quản lý. Linux trên hệ thống Alpha AXP sử dụng trang 8Kbyte, còn trên hệ thống Intel x86 là trang 4Kbyte. Mỗi trang được cung cấp một số duy nhất gọi là số khung trang (PFN : Page Frame Number).



Hình 3 : Mô hình trừu tượng của sự ánh xạ từ địa chỉ ảo đến địa chỉ vật lý

Trong mô hình này, một địa chỉ ảo bao gồm hai phần : địa chỉ offset và số khung trang ảo. Nếu kích thước trang là 4Kbyte thì từ bit 11 đến bit 0 của địa chỉ ảo chứa địa chỉ offset, còn từ bit 12 trở lên là số khung trang ảo. Mỗi lần bộ xử lý bắt gặp một địa chỉ ảo, nó sẽ lấy địa chỉ offset và số khung trang ảo ra. Bộ xử lý phải chuyển từ số khung trang ảo sang số khung trang vật lý và sau đó truy cập vào vị trí tại địa chỉ offset trong trang vật lý đó. Để làm được điều này thì bộ xử lý sử dụng bảng trang.

Hình 3 chỉ ra không gian địa chỉ ảo của hai tiến trình X và Y, mỗi tiến trình có một bảng trang riêng. Các bảng trang này ánh xạ trang ảo của mỗi tiến trình vào trang vật lý trong bộ nhớ. Khung trang ảo số 0 của tiến trình X được ánh xạ vào bộ nhớ tại khung trang vật lý số 1 và khung trang ảo số 1 của tiến trình Y được ánh xạ vào khung trang vật lý số 4. Mỗi mục trong bảng trang theo lý thuyết là chứa những thông tin sau :

- Cờ hợp lệ : cho biết mục bảng trang có hợp lệ hay không
- Số khung trang vật lý mà mục này mô tả
- Thông tin điều khiển truy cập : mô tả trang được sử dụng như thế nào ?, nó có thể được ghi hay không ?, nó có chứa đoạn mã thực thi hay không ?

Bảng trang được truy cập nhờ sử dụng số khung trang ảo như là địa chỉ offset. Khung trang ảo số 5 sẽ là phần tử số 6 của bảng (bắt đầu là phần tử 0).

Để chuyển từ địa chỉ ảo sang địa chỉ vật lý, bộ xử lý trước hết phải làm việc với số khung trang ảo và địa chỉ offset trong trang ảo đó. Xem lại hình 3 và giả thiết rằng kích thước trang là 0x2000 byte và một địa chỉ là 0x2194 trong không gian địa chỉ ảo của tiến trình Y, bộ xử lý sẽ chuyển địa chỉ đó thành địa chỉ offset 0x194 vào khung trang ảo số 1.

Bộ xử lý sử dụng số khung trang ảo như là chỉ mục vào bảng trang các tiến trình để truy xuất vào từng mục của bảng trang. Nếu mục của bảng trang tại địa chỉ offset đó là hợp lệ thì bộ xử lý sẽ lấy số khung trang vật lý từ mục này. Nếu mục này không hợp lệ thì tiến trình sẽ truy cập vào một vùng không tồn tại của bộ nhớ ảo. Trong trường hợp này, bộ xử lý sẽ không thể làm việc với địa chỉ này mà chuyển điều khiển cho HĐH để khắc phục lỗi đó.

Chúng ta hãy xem cách mà bộ xử lý báo cho HĐH biết tiến trình cố gắng truy cập vào địa chỉ ảo không hợp lệ. Điều này được gọi là lỗi trang, nó được bộ xử lý chuyển đến HĐH. HĐH được thông báo về địa chỉ ảo gây ra lỗi và nguyên nhân của lỗi trang.

Giả sử đây là một mục hợp lệ của bảng trang, bộ xử lý lấy số khung trang vật lý đó nhân với kích thước trang để lấy địa chỉ của trang cơ sở trong bộ nhớ vật lý. Cuối cùng, bộ xử lý cộng gộp vào địa chỉ offset để được chỉ lệnh hay dữ liệu cần dùng.

Sử dụng lại ví dụ trên, khung trang ảo số 1 của tiến trình Y được ánh xạ đến khung trang vật lý số 4 bắt đầu tại 0x8000 (4 x 0x2000), cộng với địa chỉ offset là 0x194 sẽ cho ta địa chỉ vật lý cuối cùng là 0x8194.

Bằng cách ánh xạ địa chỉ ảo và địa chỉ vật lý như thế này, bộ nhớ ảo có thể được ánh xạ vào bộ nhớ vật lý của hệ thống theo bất kì thứ tự nào. Ví dụ, trong hình 3, khung trang ảo số 0 của tiến trình X được ánh xạ đến khung trang vật lý số 1 trong khi khung trang ảo số 7 được ánh xạ đến khung trang vật lý số 0 mặc dù nó cao hơn khung trang ảo số 0 trong bộ nhớ ảo. Điều này chứng minh cho một kết luận thú vị về bộ nhớ ảo là : các trang trong bộ nhớ ảo được hiển thị trong bộ nhớ vật lý không theo bất kì một thứ tự nào.

III. Tạo không gian hoán đổi

Một file swap là một file thông thường, không có gì đặc biệt đối với kernel. Điều duy nhất mà nó có nghĩa đối với kernel là nó không có vùng trống. Nó được chuẩn bị để sử dụng với hàm mkswap(). Nó phải thường trú trên đĩa cục bộ.

Bit về các vùng trống là rất quan trọng. File swap dự trữ không gian đĩa để kernel có thể đưa trang ra ngoài nhanh chóng mà không phải thực hiện tất cả

các bước cần thiết khi định vị disk sector cho một file. Bởi vì một vùng trống trong một file có nghĩa là không có disk sector được định vị nên kernel không thể sử dụng được file đó.
Cách tốt nhất để tạo file swap mà không có vùng trống là thực hiện đoạn lệnh sau :

```
$ dd if=/dev/zero of=/extra-swap bs=1024
count=1024
1024+0 records in
1024+0 records out
$
```

Trong đó, **/extra-swap** là tên của file swap và kích thước được cho sau **count=.** Kích thước tốt nhất là bội số của 4 vì kernel ghi ra các trang nhớ, mỗi trang có kích thước 4 Kbyte. Nếu kích thước không phải là bội số của 4 thì cặp Kbyte cuối có thể không được sử dụng.

Một phân vùng swap cũng không có gì đặc biệt. Bạn tạo nó giống như các phân vùng khác, sự khác nhau duy nhất là nó được sử dụng như là một phân vùng thô, nó sẽ không chứa bất kì file hệ thống nào. Phân vùng swap được đánh dấu là loại 82 (Linux swap), điều này giúp cho việc liệt kê sự phân vùng rõ ràng hơn mặc dù nó không hoàn toàn cần thiết đối với kernel.

Sau khi bạn tạo một phân vùng swap hoặc một phân vùng swap, bạn cần ghi một chữ ký lên nơi bắt đầu của nó. Chữ ký này được sử dụng bởi kernel và chứa một số thông tin về việc quản lý. Đoạn lệnh để làm việc này là hàm mkswap(), được sử dụng như sau :

```
$ mkswap /extra-swap 1024
Setting up swapspace, size = 1044480
bytes
$
```

Chú ý là không gian hoán đổi vẫn chưa được sử dụng, nó tồn tại nhưng kernel không sử dụng nó để cung cấp bộ nhớ ảo.

Bạn nên cẩn thận khi sử dụng hàm mkswap() bởi vì nó không kiểm tra file hay phân vùng này sử dụng chưa. Bạn có thể dễ dàng ghi đè lên file hay phân vùng quan trọng với hàm mkswap(). Tốt hơn hết là bạn chỉ nên sử dụng hàm này khi cài đặt lên hệ thống của bạn.

Trình quản lý bộ nhớ trong Linux giới hạn kích thước của mỗi không gian hoán đổi là 127MB. Tuy nhiên bạn có thể sử dụng cùng lúc tới 8 không gian hoán đổi nên tổng kích thước lên đến 1GB. Điều này không còn đúng, với những phiên bản kernel mới hơn thì giới hạn này sẽ thay đổi tùy thuộc vào cấu trúc. Ví dụ đối với bộ xử lý i386 giới hạn này là 2GB.

IV. Sử dụng không gian hoán đổi

Một không gian hoán đổi đã khởi tạo sẽ được lấy để sử dụng nhờ lệnh **swapon**. Lệnh này báo cho kernel rằng không gian hoán đổi có thể được sử dụng. Đường dẫn đến không gian hoán đổi được cấp như là đối số, vì vậy để bắt đầu hoán đổi trên một file swap tạm thời, bạn có thể sử dụng đoạn lệnh sau :

```
$ swapon /extra-swap
$
```

Không gian hoán đổi có thể được sử dụng tự động bằng cách liệt kê chúng trong file **/etc/fstab**

```
/dev/hda8       none       swap       sw    0     0
/swapfile       none       swap       sw    0     0
```

Đoạn mã khởi động sẽ chạy lệnh **swapon -a,** lệnh này sẽ bắt đầu thực hiện hoán đổi trên tất cả các không gian hoán đổi được liệt kê trong file **/etc/fstab.** Do đó lệnh **swapon** chỉ thường được sử dụng khi cần hoán đổi thêm.

Bạn có thể quản lý việc sử dụng không gian hoán đổi với lệnh **free**. Nó sẽ cho biết tổng số không gian hoán đổi được sử dụng

```
$ free
             total       used       free     shared
 buffers
Mem:         15152      14896        256      12404       2528
-/+ buffers:            12368       2784
Swap:        32452       6684      25768
$
```

Một không gian hoán đổi có thể bị loại bỏ bằng lệnh **swapoff**. Thông thường không cần phải dùng lệnh này ngoại trừ đối với không gian hoán đổi tạm thời. Bất kì trang nào đang được sử dụng trong không gian hoán đổi đều được đưa vào trước. Nếu không có đủ bộ nhớ vật lý để chứa chúng thì chúng sẽ được đưa ra đến một không gian hoán đổi khác. Nếu không có đủ bộ nhớ ảo để chứa tất cả các trang, Linux sẽ bắt đầu dừng lại (thrash), sau một khoảng thời gian dài nó sẽ khôi phục nhưng trong lúc ấy hệ thống không thể sử dụng. Bạn nên kiểm tra (bằng lệnh **free**) để xem có đủ bộ nhớ trống không trước khi loại bỏ một không gian hoán đổi.

Tất cả không gian hoán đổi được sử dụng tự động nhờ lệnh **swapon -a** đều có thể được loại bỏ với lệnh **swapoff -a**, lệnh này tìm thông tin trong file **/etc/fstab** để loại bỏ. Còn các không gian hoán đổi được điều khiển bằng tay thì vẫn sử dụng bình thường.

Đôi khi có nhiều không gian hoán đổi được sử dụng mặc dù có nhiều bộ nhớ vật lý trống. Điều này có thể xảy ra nếu tại một thời điểm có nhu cầu về hoán đổi, nhưng sau đó một tiến trình lớn chiếm nhiều bộ nhớ vật lý kết thúc và giải phóng bộ nhớ. Dữ liệu đã đưa ra sẽ không tự động đưa vào cho đến khi nó cần, vì vậy bộ nhớ vật lý sẽ còn trống trong một thời gian dài. Bạn không cần phải lo lắng về điều này mà chỉ cần biết điều gì đang xảy ra.

V. Định vị không gian hoán đổi

Người ta thường nói rằng bạn nên định vị không gian hoán đổi gấp đôi bộ nhớ vật lý, nhưng đây không phải là một quy luật đúng. Bạn hãy xem cách làm đúng sau đây :

+ Dự đoán tổng bộ nhớ mà bạn cần. Đây là số lượng bộ nhớ lớn nhất mà bạn cần tại một thời điểm nào đó, là tổng bộ nhớ cần thiết cho tất cả các chương trình mà bạn muốn chạy cùng một lúc.

Lệnh **free** và **ps** sẽ có ích để đoán lượng bộ nhớ cần dùng.

+ Cộng thêm một ít vào dự đoán ở bước 1, bởi vì dự đoán về kích thước các chương trình có thể sai do bạn quên một số chương trình mà bạn muốn chạy, và để chắc chắn bạn nên chuẩn bị một không gian phụ để dùng khi cần. Nên định vị dư hơn là thiếu nhưng không dư nhiều quá sẽ gây lãng phí. Bạn cũng nên làm tròn lên thành một số chẵn megabyte.

+ Dựa trên những tính toán trên, bạn biết sẽ cần tổng cộng bao nhiêu bộ nhớ. Vì vậy, để định vị không gian hoán đổi, bạn chỉ cần lấy tổng bộ nhớ sẽ dùng trừ cho bộ nhớ vật lý.

+ Nếu không gian hoán đổi mà bạn đã tính lớn hơn hai lần bộ nhớ vật lý thì bạn nên mua thêm RAM, nếu không hiệu năng của máy sẽ thấp.

Tốt hơn hết là nên có một vài không gian hoán đổi cho dù theo sự tính toán của bạn là không cần. Linux sử dụng không gian hoán đổi khá linh hoạt. Linux sẽ đưa các trang nhớ không sử dụng ra ngoài cho dù bộ nhớ chưa cần dùng. Điều này giúp tránh việc chờ đợi hoán đổi khi cần bộ nhớ.

CHƯƠNG IV

CƠ CHẾ QUẢN LÝ BỘ NHỚ VẬT LÝ, ÁNH XẠ BỘ NHỚ

I. Quản lý bộ nhớ vật lý

1. Bộ định vùng

Các bảng trang kernel ánh xạ tối đa bộ nhớ vật lý vào dãy địa chỉ bắt đầu tại PAGE_OFFSET. Các trang vật lý chiếm bởi đoạn mã và dữ liệu kernel sẽ được dành riêng và không sử dụng cho bất kì mục đích nào khác. Các trang vật lý khác được định vị cho bộ nhớ ảo tiến trình, bộ nhớ đệm, bộ nhớ ảo kernel, ... khi cần. Để làm được điều này, chúng ta phải có cách theo dõi trang vật lý nào được sử dụng và được sử dụng bởi ai.

Bộ định vùng (zone allocator) quản lý bộ nhớ vật lý. Bất kì mã kernel nào đều có thể gọi tới bộ định vùng thông qua hàm alloc_pages() và được cấp một khối gồm 2^n trang được canh trên một đường biên tương ứng. Chúng ta cũng có thể chuyển khối các trang này lại cho bộ định vùng nhờ hàm free_pages(). Số mũ n được gọi là thứ tự của sự định vùng. Các khối không cần phải được giải phóng giống cách mà chúng được xác định, nhưng phải được giải phóng chính xác, và khung trang đầu tiên của khối phải có một số tham chiếu khác 0. Ví dụ, khi bạn yêu cầu một khối 8 trang, sau đó giải phóng một khối 2 trang trong khối 8 trang đó, muốn làm điều này trước hết bạn phải tăng biến tham chiếu của trang đầu tiên trong khối 2 trang. Lý do là vì khi bạn định vị một khối 8 trang, chỉ có biến tham chiếu của trang đầu tiên trong khối được tăng, mặt khác đoạn mã giải phóng trang sẽ không giải phóng một trang có biến tham chiếu là 0, nên biến tham chiếu của các trang khác phải điều khiển bằng tay. Việc tăng tất cả các biến tham chiếu của các trang trong khối là không cần thiết đồng thời làm tăng kích thước đoạn mã định vị trang.

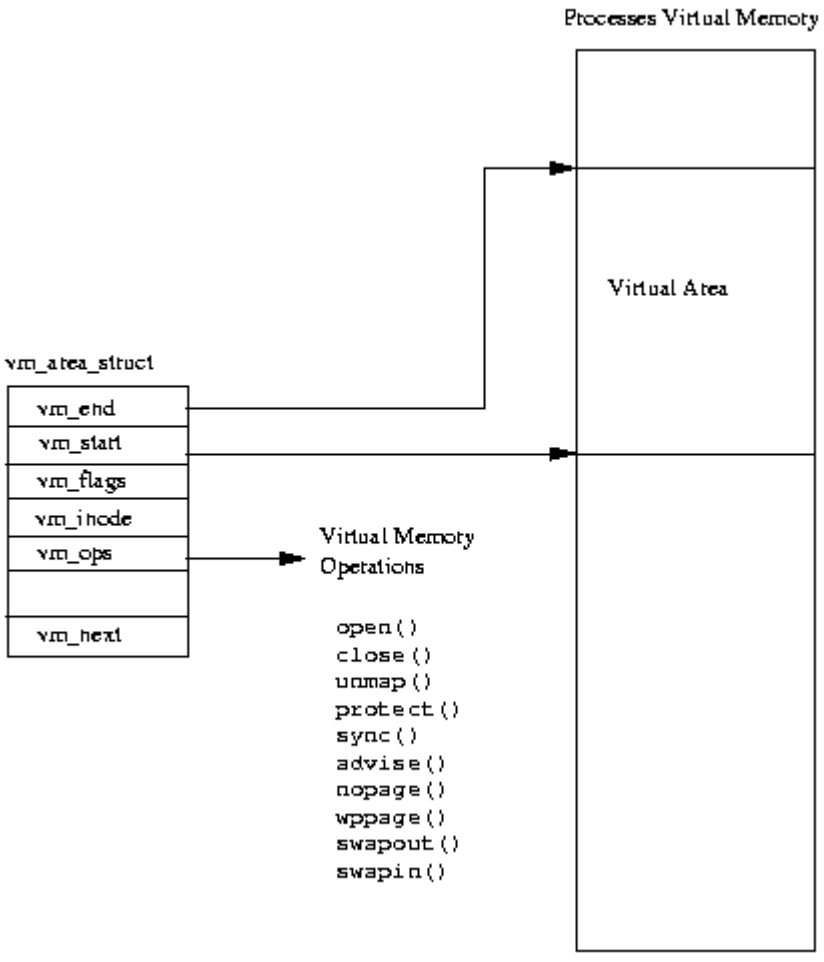
2. Các vùng

Các dãy trang vật lý khác nhau thì có các thuộc tính khác nhau, phục vụ cho các mục đích của kernel. Ví dụ, DMA (Direct Memory Access) cho phép các thiết bị ngoại vi đọc và viết dữ liệu trực tiếp lên RAM mà không có sự can thiệp của CPU, chỉ có thể làm việc với địa chỉ vật lý nhỏ hơn 16MB. Một vài hệ thống có bộ nhớ vật lý nhiều hơn có thể được ánh xạ giữa PAGE_OFFSET và 4GB, những trang vật lý này không thể truy cập trực tiếp đến kernel, vì vậy chúng phải được quản lý theo cách khác. Bộ định vùng quản lý những sự khác nhau như vậy bằng cách chia bộ nhớ thành các vùng và xem mỗi vùng là một đơn vị cho sự định vị.

Cấu trúc dữ liệu root được quản lý bởi bộ định vùng là zone_struct, gồm tập hợp tất cả dữ liệu liên quan đến việc quản lý một vùng cụ thể. Zonelist_struct bao gồm một mảng các con trỏ zone_struct và một gfp_mask chỉ ra cơ chế định vị nào có thể sử dụng zone_list. Zone_struct offset chỉ ra địa chỉ offset của nơi bắt đầu một vùng trong bộ nhớ vật lý.

II. Ánh xạ bộ nhớ (mm3 - Memory Mapping)

Khi một ảnh được thực hiện, nội dung của nó phải được đưa vào không gian địa chỉ ảo tiến trình. File thực thi không thực sự được mang vào bộ nhớ vật lý, mà thay vào đó nó chỉ đơn thuần được liên kết đến bộ nhớ ảo tiến trình. Sau đó ảnh được mang vào bộ nhớ như là một phần của chương trình được tham chiếu bởi ứng dụng đang chạy. Sự liên kết một ảnh vào không gian địa chỉ ảo tiến trình được gọi là ánh xạ bộ nhớ.



Hình 4 : Areas of Virtual Memory

Mỗi bộ nhớ ảo tiến trình được miêu tả bằng một cấu trúc dữ liệu mm_struct. Cấu trúc này chứa thông tin về ảnh hiện đang thực thi và các con trỏ đến một số cấu trúc dữ liệu vm_area_struct. Mỗi cấu trúc dữ liệu vm_area_struct mô tả điểm bắt đầu và kết thúc của khu vực bộ nhớ ảo, quyền truy cập của các tiến trình đến bộ nhớ đó và các hoạt động của bộ nhớ đó. Các hoạt động này là tập hợp các thường trình (routine) mà Linux phải sử dụng khi xử lý khu vực bộ nhớ ảo này. Ví dụ, một trong những hoạt động của bộ nhớ ảo là thực hiện các hiệu chỉnh khi một tiến trình tìm cách truy cập vào bộ nhớ ảo mà không có giá trị tương ứng trong bộ nhớ vật lý (thông qua lỗi trang). Hoạt động này là hoạt động nopage. Hoạt động nopage được sử dụng khi Linux yêu cầu các trang của một ảnh thực thi trong bộ nhớ.

Khi một ảnh thực thi được ánh xạ vào địa chỉ ảo tiến trình, một tập hợp các cấu trúc dữ liệu vm_area_struct được thực hiện. Mỗi cấu trúc dữ liệu vm_area_struct mô tả một phần của ảnh thực thi, đoạn mã thực thi, dữ liệu được khởi tạo (là các biến), dữ liệu không được khởi tạo,... Linux cũng hỗ trợ một số các hoạt động bộ nhớ ảo chuẩn.

CHƯƠNG V

CẤP PHÁT VÀ GIẢI PHÓNG VÙNG NHỚ

I. Cấp phát vùng nhớ

1. Cấp phát vùng nhớ giản đơn

Vùng nhớ giản đơn là vùng nhớ có kích thước nhỏ hơn kích thước của bộ nhớ vật lý. Chúng ta cấp phát vùng nhớ cho tiến trình dựa vào hàm *malloc()* của thư viện C chuẩn.

void *malloc(size_t size);

Không giống như trên DOS và Windows, khi sử dụng hàm này, chúng ta không cần phải khai báo thư viện *malloc.h*

size : kích thước vùng nhớ muốn cấp phát, *size* có kiểu *size_t*. Thực sự *size_t* được định nghĩa là kiểu nguyên không dấu *unsigned int*.

Ví dụ : memory1.c

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#define Kich_thuoc(1024*1024) //1 Mb bộ nhớ
/*-----------------------------------------------*/
int main( ){
char *some_memory;
int megabyte=Kich_thuoc;
int exit_code=EXIT_FAILURE;
some_memory=(char*) malloc(megabyte);
if(some_memory != NULL){
sprintf(some_memory,"Cap phat vung nho gian don ");
printf("%s",some_memory);
exit_code=EXIT_SUCCESS;
}
exit(exit_code)
}
```

Chương trình yêu cầu hàm malloc( ) cấp phát và trả về con trỏ trỏ đến vùng nhớ 1 MB. Chương trình kiểm tra con trỏ Null để đảm bảo hàm malloc( ) cấp phát thành công. Tiếp theo chuỗi "Cap phat vung nho gian don" được đặt vào phần đầu của vùng nhớ. Tuy chuỗi " Cap phat vung nho gian don" không chiếm hết 1 MB vùng nhớ nhưng hàm malloc( ) vẫn cấp phát vùng nhớ đúng bằng kích cỡ chương trình yêu cầu.

Hàm malloc( ) trả về con trỏ *void(con trỏ dạng tổng quát), vì vậy trong chương trình cần phải chuyển về con trỏ dạng char* để truy cập đến vùng nhớ theo dạng chuỗi kí tự. Hàm malloc( ) bảo đảm vùng nhớ cấp phát là một

dãy các byte xếp liền nhau. Vì lý do này, có thể định nghĩa kiểu trả về là bất kì một con trỏ dữ liệu nào.

2. Cấp phát vùng nhớ lớn

Vùng nhớ lớn có thể có kích thước vượt hơn kích thước thật của bộ nhớ vật lý. Do chưa biết hệ thống có chấp nhận hay không nên trong quá trình xin cấp phát, có thể hệ thống sẽ ngắt ngang nếu bộ nhớ cạn kiệt.

Ví dụ : memory2.c

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#define Kich_thuoc(1024*1024) //1 Mb bộ nhớ
/*-----------------------------------------------*/
int main( ){
char *some_memory;
size_t size_to_allocate= Kich_thuoc;
int megs_obtained=0;
while( megs_obtainde<16){
some_memory=(char *)malloc(size_ro_allocate);
if (some_memory != NULL){
megs_obtained ++;
sprintf(some_memory,"Cap phat vung nho lon");
printf("%s-now
allocated%dMegabytes\n",some_memory,megs_obtained);
}
else{
exit(EXIT_FAILURE);
}
}
exit(EXIT_SUCCESS);
}
```

Chương trình này tương tự như chương trình cấp phát vùng nhớ giản đơn. Nó thực hiện vòng lặp và yêu cầu cấp phát liên tục vùng nhớ (đến 512 Mb). Chương trình chạy hoàn toàn tốt đẹp và chỉ chạy trong chớp mắt. Rõ ràng hệ thống có khả năng đáp ứng nhu cầu xin cấp phát vùng nhớ khá lớn, lớn hơn cả vùng nhớ vật lý có sẵn trên máy.

Tuy nhiên, chúng ta hãy xem trong lần cấp phát vùng nhớ ở chương trình memory3.c này, khả năng đáp ứng của hệ thống còn đủ hay không. Chương trình memory3.c chỉ xin cấp phát mỗi lần 1Kb bộ nhớ và ghi dữ liệu vào đó. Chương trình này thực sự có thể làm cạn kiệt tài nguyên hay chậm đi cả hệ thống, bạn nên đóng tất cả ứng dụng trước khi chạy thử.

Ví dụ : memory3.c

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#define ONE_K(1024)
/*-----------------------------------------------------------------*/
int main(){
char  *some_memory;
int size_to_allocate= ONE_K;
int megs_obtained=0;
int ks_obtained=0;
while(1){
   for (ks_obtained = 0; ks_obtained < 1024; ks_obtained++){
        some_memory= (char *)malloc(size_to_allocate);
        if (some_memory == NULL) exit (EXIT_FAILURE);
        sprintf(some_memory, "Hello world");
    }
megs_obtained++;
printf("Now allocated %d Megabytes\n", megs_obtained);
}
exit(EXIT_SUCCESS);
}
```

Lần này chỉ cấp phát được 154Mb là chương trình chấm dứt. Hơn nữa chương trình này chạy chậm hơn memeory2.c. Tuy nhiên, bộ nhớ xin cấp phát vẫn có khả năng lớn hơn bộ nhớ vật lý có sẵn.

Bộ nhớ mà ứng dụng yêu cầu phân bổ được quản lý bởi hạt nhân Linux và UNIX. Mỗi lần chương trình yêu cầu vùng nhớ hoặc cố đọc ghi vào vùng nhớ được phân bổ trước đó, hạt nhân Linux sẽ theo dõi và quyết định xem cần xử lý yêu cầu này như thế nào.

Khởi đầu, hạt nhân hoàn toàn có thể đơn giản sử dụng vùng nhớ vật lý còn trống để thỏa mãn yêu cầu về vùng nhớ cho ứng dụng. Tuy nhiên , khi vùng nhớ vật lý bị đầy hay đã cấp phát hết, hạt nhân bắt đầu dùng đến không gian hoán đổi. Trên hầu hết các phiên bản của Linux và UNIX, vùng không gian hoán đổi này nằm riêng biệt trên một phân vùng. Hạt nhân thực hiện việc di chuyển dữ liệu và mã lệnh của chương trình từ vùng nhớ vật lý ra không gian tráo đổi và ngược lại.
Khi ứng dụng yêu cầu cấp phát vùng nhớ mà không gian tráo đổi lẫn bộ nhớ vật lý thật đã đầy thì hệ thống không thể cấp phát và hoán chuyển bộ nhớ được nữa.

Trong quá trình xin cấp phát vùng nhớ, cần phải đặc biệt lưu ý đến số lần và kích thước vùng nhớ xin cấp phát. Điều này sẽ ảnh hưởng đến hiệu quả vùng nhớ được cấp phát. Ví dụ nếu yêu cầu cấp phát 10 lần , mỗi lần 1 Kb, có thể

thu được 10 khối nhớ rời nhau , mỗi khối vẫn đảm bảo 1 Kb. Tuy nhiên để tăng hiệu quả có thể yêu cầu cấp phát một lần với kích thước khối nhớ là 10 Kb xếp liên tục gần nhau. Hệ điều hành quản lý các khối nhớ cấp phát theo danh sách liên kết . Nếu các khối nhớ nhỏ và rời rạc thì danh sách liên kết này sẽ lớn và chiếm nhiều không gian và thời gian quản lý hơn.

3. Vùng nhớ được bảo vệ

Mặc dù tiến trình được dành cho 4 Gb không gian địa chỉ nhưng tiến trình cũng không thể đọc/ghi dữ liệu tùy tiện nếu chưa xin HĐH cấp phát. Bạn chỉ có thể chép được dữ liệu vào vùng nhớ mà HĐH cấp phát thông qua *malloc()*. Nếu tiến trình cố gắng đọc hay ghi dữ liệu vào vùng nhớ mà chưa được cấp phát, HĐH sẽ quyết định cắt ngang chương trình với lỗi trang hay còn gọi là lỗi phân đoạn (segment fault) cho dù vùng nhớ đó vẫn nằm trong không gian địa chỉ 4 Gb. Chương trình sau xin cấp phát vùng nhớ 1024 bytes rồi ghi dữ liệu vào từng byte. Ta cố ý ghi dữ liệu vào byte cuối cùng của vùng nhớ (byte thứ 1025).

Ví dụ : memory4.c

```
#include<unistd.h>
#include<stdlib.h>
#define ONE_K(1024)
/*-----------------------------------------------------------------*/
int main(){
char  *some_memory;
char  *scan_ptr;
int count=0;
some_memory= (char *)malloc(ONE_K);
if (some_memory==NULL) exit (EXIT_FAILURE);
scan_ptr= some_memory;
while(1){
prinf("write byte %d", ++count);
*scan_ptr ='\0';
scan_ptr ++;
}
exit(EXIT_SUCCESS);
}
```

Khi chương trình bị lỗi hệ thống , HĐH sẽ ghi toàn bộ ảnh của tiến trình trong bộ nhớ (bao gồm các chỉ thị lệnh thực thi bị lỗi) xuống đĩa với tên file là core. Có thì dùng file này với các chương trình debug để biết nguyên nhân sinh ra lỗi.

4. Một số hàm cấp phát vùng nhớ khác

a. Hàm calloc()

Hàm này không được sử dụng phổ biến như hàm *malloc( )*. Hàm cũng cho phép một vùng nhớ mới được cấp phát nhưng nó được thiết kế phân bổ vùng nhớ cho một mảng cấu trúc (table)

void *calloc(size_t nmemb, size_t size);

Hàm này yêu cầu các tham số khó hơn hàm *malloc( )*:

+ tham số nmemb là số phần tử của mảng (số ô trong table)
+ tham số size chỉ thị kích cỡ của mỗi phần tử trong mảng (kích thước một ô trong table).

Vùng nhớ cấp phát được khởi tạo với giá trị zero. Nếu thành công , giá trị trả về của hàm là con trỏ trỏ đến phần tử đầu tiên của mảng ngược lại hàm trả về giá trị NULL. Tương tự như hàm malloc( ) mỗi lần gọi tiếp theo, calloc( ) không đảm bảo sẽ cung cấp cho chương trình vùng nhớ liên tục với lời gọi trước đó.

b. Hàm realloc()

Hàm này cũng không được sử dụng phổ biến. Hàm chỉnh sửa kích cỡ của một khối nhớ được cấp phát trước đó.

void *realloc(void *ptr, size_t size);

Hàm này nhận đối số là con trỏ *ptr* trỏ đến vùng nhớ trả về bởi các hàm cấp phát như malloc( ), calloc( ), thậm chí kể cả hàm realloc( ), sau đó thực hiện co giãn hay tăng giảm khối nhớ theo kích thước size chỉ định . Hàm đảm bảo vùng nhớ mới đạt kích thước như yêu cầu .

Nếu thành công, giá trị trả về của hàm là con trỏ trỏ đến vùng nhớ đã thay đổi kích thước. Một điều lưu ý là trong chương trình nên dùng một con trỏ khác để nhận kết quả do hàm realloc( ) trả về, không bao giờ sử dụng lại con trỏ chuyển cho realloc( ) trước đó. Ngược lại , nếu vùng nhớ không thể thay đổi kích thước như yêu cầu , hàm sẽ trả về con trỏ NULL. Do đó nếu gán cho giá trị trả về của hàm realloc( ) là một con trỏ đang sử dụng thì khi hàm không thành công, nó sẽ trả về giá trị NULL và vùng nhớ con trỏ trỏ đến trước đó sẽ bị thất lạc.

II. Giải phóng vùng nhớ

Đối với các tiến trình chỉ yêu cầu cấp phát vùng nhớ nhỏ, sử dụng chúng trong một thờii gian ngắn và kết thúc thì HĐH sẽ tự động giải phóng vùng nhớ khi tiến trình kết thúc. Tuy nhiên hầu hết các tiến trình đều có nhu cầu cấp phát vùng nhớ động khá lớn, đối với các yêu cầu cấp phát này , HĐH không hỗ trợ việc giải phóng vùng nhớ, do đó chương trình luôn cần phải giải phóng vùng nhớ khi không dùng đến nữa bằng cách sử dụng hàm free( )

void free(void *ptr_to_memory);

Khi sử dụng khai báo kèm theo thư viện ***stdlib.h***
**** ptr_to_memory*** là con trỏ trỏ đến vùng nhớ cần giải phóng. Thường con trỏ này do hàm malloc( ), calloc( ) hoặc realoc( ) trả về.

Ví dụ : memory5.c

```
#include <stdlib.h>
#define ONE_K(1024)
int main(){
char*some_memory;
int exit_code=EXIT_FAILURE;
some_memory=(char*)malloc(ONE_K)
if(some_memory!=NULL){
        free(some_memory);
        exit_code=EXIT_SUCCESS ;
        }
        exit(exit_code)
}
```

Sau khi vùng nhớ được giải phóng, thì con trỏ trỏ đến vùng nhớ đã giải phóng không thể sử dụng được nữa (không thể đọc/ghi) do hệ thống đã loại bỏ vùng nhớ khỏi sự quản lý của HĐH. Mọi thao tác đọc ghi trên vùng nhớ đã giải phóng sẽ gây ra lỗi .

III. Truy xuất con trỏ NULL

Linux bảo vệ rất chặt chẽ việc đọc/ghi trên con trỏ NULL. Do Linux sử dụng thư viện chuẩn GNU (hàm sprintf và printf) nên việc đọc con trỏ NULL là được phép. Thư viện GNU đơn giản trả về cho bạn chuỗi "null", tuy nhiên ghi vào con trỏ Null là bị cấm.

Ví dụ : memory6.c

```
#include <unistd.h>
#include<stdlib.h>
#include<stdio.h>
int main() {
        char *some_memory=(char*)0;
        printf("doc tu con tro null %s\n", some_memory);
        sprintf(some_memory,"ghi vao con tro null");
        exit (EXIT_SUCCESS);
}
```

Nếu không sử dụng các hàm thư viện GNU thì việc đọc trên con trỏ Null cũng bị cấm.

Tài liệu tham khảo

1. Lập trình Linux (tập 1) - Nguyễn Phương Lan, Hoàng Đức Hải
2. Linux Kernel - Đỗ Duy Việt, Nguyễn Hoàng Thanh
3. Làm chủ hệ điều hành Linux - Elicom
4. http://home.earthlink.net/~jknapka/linux-mm/vmoutline.html
5. http://www.tldp.org/LDP/sag/html/memory-management.html
6. http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO-7.html
7. http://www.linuxpowered.com/LDP/LDP/tlk/mm/memory.html