Implementation of Page Replacement Algorithm with Temporal Filtering for Linux

Vasundhara Rathod, Department of Computer Engineering, Veerama Jijabai Technological Institute, Mumbai
Pramila Chawan, Professor, Department of Computer Engineering, Veerama Jijabai Technological Institute, Mumbai

ABSTRACT
This paper is a study of memory management systems of an operating system and implementation of Clock with Adaptive Replacement using Temporal Filtering. The essential requirement of a memory management system is to provide ways to dynamically allocate portions of memory to programs at their request, and freeing it for reuse when no longer crucial. This is vital to the computer system. Several methods have been devised that increase the effectiveness of memory management. Virtual memory systems separate the memory addresses used by a process from actual physical addresses, allowing separation of processes and increasing the effectively available amount of RAM using paging or swapping to secondary storage. The quality of the virtual memory manager can have an extensive effect on overall system performance. We begin with a brief introduction to memory management systems and then we develop a new algorithm Clock with Adaptive Replacement using Temporal Filtering for resourceful usage of memory.

Keywords: Linux Memory Manager, Page Replacement, Clock with Adaptive Replacement, Temporal Filtering.

Introduction
In this paper, we will be comparing the Memory Management Subsystems of BSD 4.4, Linux 2.6 and Windows in brief. BSD 4.4 is a representative Unix version including important operating system design principles, and today many operating systems like FreeBSD, NetBSD and OpenBSD are based on it. Windows is a very popular operating system for use as a desktop especially with beginners, and has now evolved into a mature operating system. Linux 2.6 [4] was chosen because it is growing more and more popular by the day, has an important place in the future. Finally we propose a new page replacement algorithm Clock with Adaptive Replacement using Temporal Filtering and give implementation details.

Memory Management Systems
1. Virtual Memory
An essential notion in the perspective of Memory Management Systems is Virtual Memory. In the early days of computing, researchers sensed the ever growing memory requirements of application programs, so the idea of Virtual Memory was born. The insight is to provide an application program the false impression of the presence of a very large amount of memory available for usage. The kernel provides such capability by making use of secondary storage i.e. the hard disk to fulfill the additional space needs. [5]. We require some mapping function for the virtual memory system to work, which will perform address translation i.e. converting the virtual address to the physical address. The virtual address is the address which the application uses to refer to a memory location, and the physical address is the actual memory location passed out to the local memory bus. This function is generally Paging, or Segmentation, or both depending on the kernel, architecture of the processor and its state.

2. Paging
In paging, the virtual and real address space is divided into fixed-sized (though they can be unequal sized [6]) pages. The pages can be separately manipulated, and positioned at different places in the physical memory and the hard disk. The address translation is actually done by the Memory Management Unit (MMU) of the processor by using a page table, as shown in Figure 1. Page tables indicate the mapping between virtual pages and physical pages i.e. which virtual memory page at present occupies which physical page. The MMU converts the virtual memory address to a physical address which consists of a page frame number and an offset within that page.

Virtual Address Space  Physical Memory

![Page Table Diagram]

www.borjournals.com  Blue Ocean Research Journals  72
Some similarities of are enumerated below:

1. The BSD4.4 VM system is based on Mach 2.0, 2.5 and 3.0 VM code. Windows was developed in a long series of operating systems since MSDOS. The Linux 2.6 has been developed by hackers originally founded by Linus Torvalds.

2. The BDS4.4 VM system is based on Mach 2.0, 2.5 and 3.0 VM code. Windows was developed in a long series of operating systems since MSDOS. The Linux 2.6 has been developed by hackers originally founded by Linus Torvalds.

3. All the three systems have modern Memory Management systems, and have a lot in common. The data structures and the features are also quite similar. Some similarities of are enumerated below:
   1. Hardware Abstraction Layer: All Operating Systems have a layer called the hardware abstraction layer (HAL) which does the system-dependent work, and hence enables the rest of the kernel to be coded in platform independent fashion. This eases porting it to other platforms.
   2. Copy-on-write: When a page is to be shared, the system uses only one page with both processes sharing that same copy of the page. Yet, when one of the process does a write onto the page, a private copy is made for that process, which it can then manipulate individually. This gives a lot better efficiency.
   3. Shadow paging: A shadow object is created for an original object such that the shadow object has some of its pages modified from the original object, but shares the rest of the pages with the original object. They are formed as a result of Copy-On-Write action.
   4. Memory mapped Files: A file can be mapped onto memory, which then can be used with simple memory read/write instructions.
   5. Inter-Process Communication: The memory mapped files are allowed to be then shared between processes forming a method for inter-process communication.
   6. Background daemon: There exists a background daemon which is invoked periodically and performs tasks like page flushing, freeing unused memory, etc.

Linux implements the virtual memory data structure in a similar manner to UNIX. It maintains a linked list of vm area structs. These are structures which represent continuous memory areas which have the same protection parameters etc. This list is searched whenever a page is to be found that consists a particular location. The structure also records the range of address it is mapping onto, protection mode, whether it is pinned in memory (not-page-able), and the direction (up/down) it will grow in. It also records whether the area is public or private. If the number of entries grows greater than a particular number, usually 32, then the linked list is converted into a tree. This is a quite good approach which uses the best structure in the best situations.

1. Distribution of Process Address Space
   All the three systems distribute the process virtual address space in a similar manner. Higher part of it is used by the kernel, while the process can use the lower part. The kernel part of the space of all process usually point to the same kernel code. So while switching a process, we need to switch the page table entries of the lower part, while the upper part can remain the same. In Linux and BSD, usually 3GB is kept for the process and 1 GB given to the kernel, while in Windows, 2GB are kept for each.

2. Page Replacement
   Page Replacement is an important part of any Memory Management System. Basically, page...
replacement concerns with choosing which page to page-out - i.e. swap out from memory, whenever the need for more free memory arises.

The Ideal Page Replacement Algorithm is to remove the page which will be required for access in the most distant future. Doing this will cause the least number of page faults, and thus least wastage of time doing swapping, in turn improving system performance and throughput. But since it is not possible to know what pages will be accessed in the future, the ideal page replacement algorithm is impossible to implement.

Let us see how each of the system works for page replacement.

Up to Linux 2.2, the Linux VM had focused on simplicity and low overhead. Hence it was rather quite primitive and had many problems, especially under heavy load. It was influenced by System V. However Riel [11] has worked a lot on the Linux VM in the past couple of years, and improved it a lot for the Linux 2.6 release. Linux uses a demand paged system with no pre-paging.

Until kernel version 2.2, Linux used NRU algorithm for page replacement, but due to the various shortcomings of the algorithm, they have changed it and implemented an approximate Least Recently Used in 2.6.

The aging to effect LRU is brought about by increasing the age (a counter associated with a page) of a page by a constant when the page is found to be referenced during a scan, and, decreased exponentially (divided by 2) when found not to have been referenced. This method approximates LRU fairly well.

Linux 2.6 divides the virtual pages into 4 lists [9]-
1. Active list
2. Inactive-dirty list
3. Inactive-clean list
4. Free list

To separate the pages which were chosen for eviction, the inactive-dirty list was made. Normally, the active pages are on the list 1. But as time passes, if some of the pages are not active, then their age decreases and goes down to 0, which indicates it is a candidate for eviction. Such pages are moved from list 1 to list 2.

The inactive list in BSD4.4 (and FreeBSD) has a target of 33%, which the daemon manages to keep it at that level. However, for linux 2.6, the inactive list size was made dynamic. Now the system itself will decide how many inactive pages it should keep in the memory given the particular situation.

The unification of the buffer cache and page cache has been completed in 2.6, as had been implemented in FreeBSD.

Another optimization present in the Linux Kernel, is that they now recognize continuous I/O, i.e. they now decrease the priority of the page “behind” and so that page becomes a candidate for eviction sooner.

The page daemon in Linux is kswapd which awakens once a second, and frees memory if enough is not available.

And the flushing is done by another daemon bdflush, which periodically awakes to flush dirty pages back to the disk. The page flushing that takes place from the inactive list, does not happen in an ad-hoc fashion, but the system waits for the right time, when clustering could be used, and disk read-writes could be minimized, thus optimizing the flushing.

Cart Implementation

1. Zone
The data structures related to CAR algorithm are defined in mmzone.h. We will go through each of these definitions.

Active_list and frequent_list are T1 and T2 lists of CAR. They contain actual pages in memory. ghost_active_list and ghost_frequent_list are B1 and B2 lists of CAR. They contain metadata of previously accessed pages. hashtable structure is used to define hashtables for ghost lists. It is used for efficient retrieval of metadata of previously accessed pages.

ghost_active_hashtable and ghost_frequent_hashtable are hashtables for B1 and B2 ghost lists.

nr_active and nr_frequent are number of pages present in T1 and T2 lists.

p is the parameter of CAR indicating the expected size of T1.

We have reused the active_list and nr_active provided by the linux kernel for T1 list. We do not require inactive_list and nr_inactive because we have introduced frequent_list and nr_frequent for T2.
However, we do not remove the definitions for inactive list since they are used in the system for various other purposes.

2. Initialization
The car_init() function initializes all the list heads (in the zone structure) in all the zones of every node. It also initializes other variables (counters) pertaining to CAR. To allocate memory for ghost list nodes in the cache directory, a pool of memory is allocated.

3. Update CAR Parameters
The update_car_params() function is called whenever there is a page miss. For the required page, its corresponding zone is checked. When the page is not in the cache, it is either in B1, or in B2 or not available in the cache directory. Whenever it is in B1 or B2, the value of p is updated. The page reference bit is updated and the page is moved to the appropriate list.

The CART algorithm modified to suit the Linux kernel is presented below:

```
INPUT: The requested page x.
if (x is in T1 [ T2) then
    /*cache hit*/
    Set the page reference bit to 1.
else
    update_car_params()
end

Algorithm 1: Algorithm for Clock with Adaptive Replacement.
```

location is used to store whether the page is in B1 or in B2 or not in both. A non-zero value in location indicates it is in one of B1 or B2.

ghost_active contains a non-zero value when the page is in B1. ghost_frequent contains a non-zero value when the page is in B2.

if (x is in B1) then
    /*cache directory hit*/
    Adapt: Increase the target size for the list T1 as:
    p = min(p + max(1, |B2|=|B1|), c)
    Move x at the tail of T2 Set the page reference bit of
    x to 0.
else if (x is in B2) then
    /*cache directory hit*/
    Adapt: Decrease the target size for the list T1 as:
    p = max(p - max(1, |B1|=|B2|), 0)
    Move x at the tail of T2 Set the page reference bit of
    x to 0.
else
    /*cache directory miss*/
    Insert x at the tail of T1. Set the page reference bit of
    x to 0.
end

```

Algorithm 2: Update CART parameters
```
zone contains the zone in which the page is stored. The values of location, ghost_active, ghost_frequent, zone are updated by the corresponding functions find_in_ghost_list(), GhostActive(), GhostFrequent() and page_zone().

found = 0
while found = 0 do
    if |T1| >= max(1, p) then
        if the page reference bit of head page in T1
        then
            found = 1
            Demote the head page in T1 and make it the
            MRU page in B1.
        else
            Set the page reference bit of head page in T1
            to 0, and make it the tail page in T2.
end
else if (x is in B2) then
    /*cache directory hit*/
    Adapt: Decrease the target size for the list T1 as:
    p = max(p - max(1, |B1|=|B2|), 0)
    Move x at the tail of T2 Set the page reference bit of
    x to 0.
else
    /*cache directory miss*/
    Insert x at the tail of T1. Set the page reference bit of
    x to 0.
end

Algorithm 3: Modified Replace()
```

Conclusion
CART is an approximation to ARC and CAR that has been found to give comparable results at a much lesser overhead. We decided to implement CART in
the Linux kernel. Our work consists of a patch to the Linux kernel consisting of methods like update_cart_params() and replace() to use the CART policy for page reclamation. Future enhancements involve testing the modified virtual memory system for various workloads. Detailed comparative analysis with the existing page reclamation policy needs to be carried out so as to determine the correctness in the eviction of pages as well as efficiency in maintaining the cache directory.

References