Design Document for Assignment 3 - Page Replacement

Kenneth Mayer
Marian Vladoi

March 5, 2018

1 Introduction

In this assignment, we experiment with changing the page replacement algorithm in FreeBSD. Page replacement algorithms determine which pages are removed from physical memory when a new page needs to enter. Ideally, an algorithm wants to remove pages that are not going to be used again for a long time, minimizing the number of pages that have to be moved from disk to memory.

FreeBSD currently uses an algorithm that approximates global LRU. In this algorithm, there are two queues, one for active pages and one for inactive pages. Additionally, a list of free pages is maintained. Active pages are pages that are currently being used by an active process. Inactive pages may be dirty and are usually not part of an active region. Free pages will be used to fulfill new page fault requests.

FreeBSD's algorithm has a concept of page shortage. It enforces a rule that a certain percentage of available memory should be in the free and inactive lists. Thus, if we are running low on memory, the pageout daemon will attempt to move active pages to the inactive pages and inactive pages to the free list to maintain its targets. It will do this based on recent activity counters. If after scanning the active list, the targets are still not met, memory is swapped out if swap space is available. Otherwise, it kills the most memory hungry process.

2 Fat Chance Algorithm

We will be modifying the algorithm in a few key ways to implement the fat chance algorithm.

Firstly, when moving the page to the free list, we look at its page number; if it is even, we place it on the front of the list, if it is odd, we place it on the rear of the list. Currently any pages are moved to the rear of the list.

Secondly, when decreasing an active page's active count, we do so exponentially, rather than subtracting a fixed value, by dividing the activity count by two.

Thirdly, when a page is referenced, we decide randomly whether it should have its reference bit set.

Fourthly, when a page is requeued onto the active list (for example, when its activity count is decreased), it is placed on the front of the list instead of the rear.

Finally, when a page is moved to the inactive list, it is placed on the front instead of the rear.

Additionally, we need the pageout daemon to run more often, every ten seconds instead of every ten minutes.

The pageout daemon also needs to log some statistics to the system log. These statistics include which lists it scanned during its run, how many pages it moved from the active queue to the inactive queue, how many

pages it moved from the inactive queue to the cache/free list, how many dirty pages it queued for flush, and the total number of active and pages are in the two queues at the end.

2.1 Analysis

This algorithm is obviously not a good algorithm, which is reflected in its name. A few elements specified by the algorithm are quite suspect. It's probably not a good idea to decrease a page's activity count exponentially, as that could severely penalize very active threads if they happen to be inactive on just one pass of the pageout daemon. However, in a workload, where for example, pages are being used heavily for a short period of time and then discarded, it might be beneficial to very quickly make them inactive.

It also uses two very arbitrary mechanisms for deciding certain decisions.

Firstly, the page number's parity affects whether it's likely to be replaced first when it's placed in the free/cache list, which makes very little sense. The only way this might improve the performance is if the workload often requests pages it hasn't used for a while. In this case, maybe some of the more recently used pages would have been replaced in the free/cache list, where the default algorithm would have replaced them more quickly.

Secondly, choosing randomly whether to set the reference bit also makes very little sense. It could definitely harm a process's performance if some pages are uncommonly referenced, but are likely to be reused. Some of those pages will never be counted as referenced.

Finally, the choice of queuing and requeuing pages at the front is probably a bad idea, as it means that recently moved pages will be checked first by the pageout daemon for satisfying a page shortage. This could be beneficial for a page that quickly becomes active, at is may quickly be moved up to the active queue. However, for a page that is occasionally active, the daemon may move it between the queues very often and potentially free it before it gets a chance to be used again.

3 Submitted Files

```
usr                    // a folder containing the modified FreeBSD source
count.c                // a ``script'' that helps extract data from the log file
pageout_results.xlsx   // a mess of results put into excel after being parsed
DESIGN.pdf             // this file
WRITEUP.pdf            // shows the results of testing done on the paging system
assgn3.txt             // List of team members
README.krmayer         // Kenneth Mayer's contribution to the project
README.mvladoi         // Marian Vladoi's contribution to the project
```

4 Modified files

Only a few files needed to be modified to implement this algorithm.

```
/usr/src/sys/vm/vm_pageout.c // Here we modify the pageout daemon
/usr/src/sys/vm/vm_page.c    // Some routines called by the pageout daemon are in here
/usr/src/sys/vm/vm_phys.c    // The actual freeing is done here, so it needs to be modifed
```

5 Implementation

Note: changes in the source code are marked by "Vladoi Marian" or "Marian Vladoi", use the string "Vladoi" or "Marian" to find them.

5.1 vm_pageout.c

In this file modify the functions `vm_pageout_scan()`, which is the function that runs the pageout daemon and we modified the function `vm_pageout_init()`.

5.1.1 vm_pageout_init()

The modification to `vm_pageout_init()` was very simple.

```
if(vm_pageout_update_period is 0){
    vm_pageout_update_period = 10 seconds
}
```

This causes the pageout daemon to run every 10 seconds. Default is 600 seconds (or 10 minutes).

5.1.2 vm_pageout_scan()

`vm_pageout_scan()` is substantially more involved. The pseudocode below shows how it works after modification. Quite a bit of it was kept the same.

```
vm_pageout_scan(){
    calculate page shortage (according to targets)

    scan the inactive list{
        inactive_scanned = 1;
        lock the current scanned page

        if(page is invalid){
            free page
            inactive_to_free++
        }

        if(page is referenced){
            clear referenced bit
            act_delta = 1
        }
        else{
            act_delta = 0
        }

        if(act_delta != 0){
            if(page is referenced){
                activate the page (place it at the tail of the active)
                increase page activity count
            }
            else{
                place page at the tail of the inactive list
            }
        }

        if(page is not referenced){
```

```
            check whether the page is dirty
            if(page is not dirty)
                remove memory mappings
        }

        if(page is not dirty){
            free page
            inactive_to_free++
            page shortage--
        }
        else{
            launder the page
            inactive_to_flush++
        }
    }

    scan the active list{
        if(page is referenced){
            act_delta = 1
        }
        else{
            act_delta = 0
        }

        if(act_delta != 0){
            increase page activity count
        }
        else{
            divide page activity count by 2
        }

        if(activity count is 0){
            if(inactq_shortage <= 0){
                deactivate the page (move to front of inactive list)
                active_to_inactive++
            }
            else{
                if(page is clean){
                    deactive the page
                    active_to_inactive++
                    decrease inactq_shortage
                }
                else{
                    launder the page
                }
            }
        }
    }

    check the number of active page
    check the number of inactive pages
    log(
        which queues were scanned
        active_to_inactive
        inactive_to_free
        inactive_to_flush
        total active pages
        total inactive pages
    )
}
```

5.2 vm_page.c

Only a few modifications are made here, but these small modifications are fairly large changes are as far as how the page replacement algorithm works.

5.2.1 vm_page_reference()

Here we modify the code that sets the reference flag to only do it half the time.

```
if(random % 2 == 1){
    vm_page_aflag_set(m, PGA_REFERENCED);
 }
```

5.2.2 vm_page_requeue()

This is a simple change; place the page at the head of its current queue rather than the tail.

```
TAILQ_INSERT_HEAD(&pq->pq_pl, m, plinks.q);
```

We also tried modifiying code in `vm_object.c`, to randomly increment the object's member `ref_count`, however, no matter what method we used (whether we called the `random()` function or not), this would cause the OS to hang on boot. We suspect that if a page gets referenced, its object's `ref_count` must always be incremented for the system to work correctly.

5.2.3 vm_page_requeue_locked()

Same change as the previous function.

```
TAILQ_INSERT_HEAD(&pq->pq_pl, m, plinks.q);
```

Making these last two changes means that any time a page is requeued, it is placed at the head of the queue, rather than the front. This means that a few changes that were not specified will occur. Specifically, that even an active page that has its activity count increased is requeued at the head rather than the tail. I think it's best to keep the changes consistent; the original algorithm has all requeued processes moving to the tail, so we changed it to make all requeued processes move to the head instead.

5.2.4 _vm_page_deactivate()

The change here is once again the same.

```
TAILQ_INSERT_HEAD(&pq->pq_pl, m, plinks.q);
```

5.3 vm_phys.c

One function is modified here.

5.3.1 vm_phys_free_pages()

In this function, we change it so that free pages are placed in different locations on the free list based on their page number. If it is odd place it on the tail, if it is even, place it on the front.

```
if((m->phys_addr >> 12) & 1){
    vm_freelist_add(fl, m, order, 1);
}
else{
    vm_freelist_add(fl, m, order, 0);
}
```

The page number is extracted from the physical address by removing the 12-bit page offset and bitwise AND'ing the result with 1. If that's true, it means the page number was odd.

6 Conclusion

This assignment was quite successful. Implementing the algorithm itself was straightforward, but gathering and tabulating the results proved very challenging. If we had more time, We'd have wrote some more scripts to assemble the data into a more useful format (some of it, we had to do by hand or with excel) and we'd make the charts reflect an average over all the runs we performed rather than just the two that we selected.