Assignment 3 Design Doc – Page Replacement

Team: Prateek Chawla (pchawla), Adel Danandeh (addanand), Tim Mertogul (tmertogu, Team Captain)

Project Overview:

The goal of this project is to modify the current paging implementation of FreeBSD. The purpose of these modifications are to compare the current paging algorithm to our modified version of it and statistically analyze the difference.

Page Replacement Overview:

We are implementing a Slim Chance algorithm where inactive and invalid pages are placed on the front of the free list. Additionally, rather than subtracting from the activity count we are dividing it by two and moving it to the front of the active list. Lastly, when a page is moved to the inactive list, it inserted in the opposite end of the queue. Additionally, we are generating stats about performance and writing them to the system log in order to analyze and confirm our speculation (hypothesis) of this Slim Chance algorithm being inferior to the default (built-in) paging mechanism.

Project Specifics:

Specifically, the bulk of our modifications occur in vm_pageout.c (please note other files were modified as well) as that is where the vm_pageout_scan() method is located. Notably, vm_pageout_scan() is where the 'dirty' work for the pageout daemon is done. Here we declare and initialize variables/counter for logging purposes. For more details on where these variables are incremented see the Modified Files and Method section. Additionally, in this method we make the required modifications in order to implement Slim Chance. Calls to our modified methods are inserted as needed. To place inactive and invalid pages on the front of the free list vm_page_free_rear() is called, which calls vm_page_free_toq_rear(), which calls vm_phys_free_pages_rear(), which finally calls vm_freelist_add_rear() where the actual queue modification occurs. Additionally, the activity count is divided by two as specified in the project spec. Lastly, when a page is moved the to inactive list a call to vm_page_deactivate_front() occurs, which calls _vm_page_deactivate_front() where the page is added to the front rather than than the rear. Also, please note pageout is run more frequently (every 10 seconds). Finally, the statistics are logged.

Stats Generated:

All the stats analysis is done in a function called *vm_pageout_scan()* which can be found under *sys/vm* directory. In each iteration of page daemon, all the pages in active and inactive are being scanned. Initially, time interval for it was set to 600 seconds (10 minutes), but it has been changed to 10 seconds. There are counters defined in order to keep track of the stats as we are asked on the assignment:

- *number of pages scanned*
- *number of pages moved from active to inactive*
- *number of pages moved from inactive to cache*

- *number of pages moved from inactive to free*
- *number of pages queued for flush*

Each individual counter gets incremented in the proper place (see the appropriate subsection in 'Modified Files and Methods'). Specifically, in order to find the total number of pages scanned, in each iteration of pages in active and inactive queues the *number of pages scanned* is incremented. Again, all statistical information is explained in further detail in it's appropriate section below, see Fig. 2 for declarations. Importantly, all the stats are printed to a log file called messages which it could be found under */var/log.*

Additionally, we created a stress test script to help generate these statistics. Important components of this stress test included the `vmstat`, `/usr/bin/time` and `stress` commands. Notably, this testing (and general system configurations) were consistent between the standard standard FreeBSD code and our modifications.

Specifically we ran `/usr/bin/time -l stress -m 3 --vm-bytes 512M -t 20s`

- `/usr/bin/time -l`
    - Reports system resource usage during execution of a given command
    - We added this command to get specific page faults and reclaims for our stress test, rather than having to rely on deltas
- `stress -m 3 --vm-bytes 512M -t 20s`
    - A stress test (workload generator)
    - The `-m 3` flag spawned 3 workers spinning on malloc()/free()
    - The `--vm-bytes 512M` flag malloc 512M per vm worker
    - The `-t 20s` flag is a 20 second timeout

While, vmstat reported additional virtual memory statistics before and after each stress test. We additionally, wrote to log and printed the dates for reference purposes. The full code can be found in `stress.sh` Results and any further explanation of stats generated can be found in WRITEUP.pdf.

FreeBSD Configurations:

In order to maintain consistency we used the same virtual machine configurations for all testing (i.e. the system specs did not change between runs of the standard FreeBSD code and our modifications). Pertinent configurations are as follows (using vmware fusion):

- 128MB RAM
- 2 Cores

Files Modified:

Here is the list of files that we have modified in order to implement our paging algorithm:

- vm_pageout.c

- vm_page.h
- vm_page.c
- vm_phys.h
- vm_phys.c

Modified Files and Methods:

Please note ellipses (...) indicates unmodified sections of code from the default FreeBSD mechanism.

---------- **sys/vm/vm_pageout.c** ----------

The include statement in Fig.1 was added in order to print the counter variables (below) to the log file.

```
#include <sys/syslog.h>
```

Fig 1.

vm_pageout_scan()

We have declared variables inside the function vm_pageout_scan() in order to keep track of the movement of the pages, specifically for logging and statistical analysis purposes. This is done here because vm_pageout_scan() is where the 'dirty' work for the page daemon is done. See Fig. 2.

```
declare number of pages scanned
declare number of pages moved  from active to inactive
declare number of pages moved from inactive to cache
declare number of pages moved from inactive to free
declare  number of pages queued for flush
```

Fig 2.

The *number of pages scanned* (num_pages_scanned) is incremented for each iteration of the the inactive queue (see Fig. 3). Please note this only corresponds to half the total number of pages scanned as there pages scanned in the active queue. This is accounted for later.

```
loop through inactive queue
        …
      increment number of pages scanned by one
        …
end loop
```

Fig 3.

The *number of pages moved from inactive to free* (num_pages_moved_inactive_to_free) is incremented within the same inactive queue loop as above. This is done here because this loop is where the inactive queue is scanned for pages that can be moved to cache or free. Specifically, the incrementation occurs within the conditional statement in Fig. 4 because this is where we free the page. Importantly, our freeing the page is a modified version of the built-in mechanism, as per the design requirements. Specifically, the inactive and invalid pages are place on the front of the free list. The actual modification, of the queues, occurs in the vm_phys.c file. Specifically, vm_page_free_rear() calls vm_page_free_toq_rear(), which calls vm_phys_free_pages_rear(), which finally calls vm_freelist_add_rear(). See the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (below).

********************Duplicate Methods*********************

We created duplicate methods with our changes in order to ensure that no other pre-existing functionality was unintentionally broken (beyond the scope of this project). Essentially, the original methods are called from portions of the code not specifically meant to be modified by us. There is a potential dependency chain outside of the project specifications that we did not want to change.

*****************Duplicate Methods (Reasoning)*****************

```
loop through inactive queue
      …
      if page is valid
         free the page
         increment number of pages moved from inactive to free by one
      end if
      …
 end loop
```

Fig. 4

Within the same loop, iterating through the inactive queue, as above there is a conditional check to see if the page is dirty and to move it the cache. Notice here if the pages are clean, so they are placed on the cache queue (effectively freeing them). Therefore, this was the only logical place to increment *number of pages moved from inactive queue to cache* (num_pages_moved_inactive_to_cache). See Fig. 5.

```
loop through inactive queue
      …
       if page is dirty
         move the clean pages to cache queue
         increment number of pages moved from inactive queue to cache by one
      …
```

```
        end if
        …
    end loop
```
Fig. 5

Within the same loop, iterating through the inactive queue, we increment the *number of pages queued for flush* (num_pages_queued_for_flush) within the conditional in Fig. 6. Notice if a page is dirty, then it can either be being washed or is in the laundry. If it is in the laundry than the cleaning operation still has to begin. Laundering a page is essentially marking it for flush. Importantly, dirty pages are queued for flush as the modifications/changes have to be written out to disk prior to a page out. In other terms, if a page is dirty, we check to see **if (vm_pageout_clean(m) != 0)** we could say that the page is being flushed.

```
    loop through inactive queue
        …
        if page is dirty
           increment the number of pages queued for flush by one
        end if
        …
    end loop
```
Fig. 6

At this point we are done iterating through the inactive queue and have to begin iterating through the active queue. Therefore, the *number of pages scanned* (num_pages_scanned) is incremented for each iteration of the the active queue (see Fig. 7). Notably, this accounts to the other half of the total number of pages scanned mentioned above, where total numbers of pages scanned equals the number of pages scanned in the active queue plus the number of pages scanned in the inactive queue.

```
    loop through active queue
        …
        increment number of pages scanned by one
        …
    end loop
```
Fig. 7

Please Note: We have decided to increment the *number of pages* scanned towards top of the inactive and active loops. Specifically, this is done before all the given conditional if statements, because there are continue statements within some of the if blocks that may skip over any further iterating through pages.

Regardless, we assumed such a page to be scanned as in order for some decision process to conclude that this page should be skipped required it to be iterated over (scanned), at least partially.

While iterating through the active queue loop the activity count is modified. Originally two was subtracted from it, but our modifications consist of dividing it by two and moving the page to the front of the active list. This is the done to meet the specified requirements of the project. Additionally, this is the only logical place to do this modification as it is where the original documented functionality exists. See Fig. 8.

```
loop through active queue
        …
       if not act delta
          divide the act count by two
          move the page to the front of active list
        end if
        …
end loop
```

Fig. 8

Additionally, while iterating through the active queue loop when a page is moved to the inactive list it is placed on the rear. Our modifications consist of moving it to the front instead, as per the design specifications/requirements (see Fig. 9). Originally there was a call to vm_page_deactivate(m), which calls _vm_page_deactivate() with an athead value of 0 where the page is inserted in in rear. We created the following two methods vm_page_deactivate_front() and _vm_page_deactivate_front() to perform the opposite operation (inserting the page in the front). See the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (above). Additionally, a more detailed explanation of the two new methods listed above can be found in their corresponding subsections. Please note this modification is done here specifically because this portion of the code corresponds to where the page is moved from active to inactive.

```
loop through active queue
        …
       if act delta equals zero
          …
          insert the page in front of the inactive list.
          …
        end if
        …
end loop
```

Fig. 9

Finally, we print all the updates that are taking place at the bottom of vm_pageout_scan(). We utilize the log() function from sys/syslog.h with the logging level LOG_INFO. We write to the log at the end of vm_pageout_scan() because we want statistics for each execution/pass of the page daemon. See Fig. 10.

Please note the logging path: /var/log/messages

```
log number of pages scanned
log  number of pages moved  from active to inactive
log  number of pages moved from inactive to cache
log  number of pages moved from inactive to free
log  number of pages queued for flush
```

Fig. 10

Please Note: We keep track of number of pages scanned in inactive queue and the active queue using one variable called *num_pages_scanned***.** Since, we want to show how many pages are being scanned, there is no need for creating a separate counter for each queue and then display the sum of the two values. Essentially, the total numbers of pages scanned equals the number of pages scanned in the active queue plus the number of pages scanned in the inactive queue.

vm_pageout_init():

We changed the time interval for page scan from the default value which was 600 seconds (10 minutes) to 10 seconds as per the specifications on the assignment. To do this we located the default mechanism and replaced the default value of 600 with 10. See Fig. 11.

```
if page update period is zero
    set the time interval to 10 seconds
end if
```

Fig. 11

---------- **sys/vm/vm_phys.h** ----------

Here we included our function declaration as required by C. See the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (above).

```
void vm_phys_free_pages_rear(vm_page_t m, int order);
```

Fig. 12

---------- **sys/vm/vm_phys.c** ----------

vm_freelist_add_rear()

In order to place inactive and invalid pages on the front of the free list we created the function vm_freelist_add_rear(). Please see the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (above). This function does the opposite of the original function vm_freelist_add(), wherein it places inactive and invalid pages on the front of the free list. This occurs because its caller vm_phys_free_pages_rear() passes 1 (or True in C) as the value for the *tail* variable, and the *tail* variable dictates if the insertion occurs in the head or tail. To insert in the front rather than the rear all that was required was to flip the insert statements, as per the design spec and Professor Long's recommendation. See Figure 13.

```
fnc: vm_freelist_add_rear()
     if tail
        insert into the head
     else
        insert into the tail
     end if/else block
end fnc
```

Fig. 13

vm_phys_free_pages_rear():

We added this function as it part of the call path that is required to place inactive and invalid pages on the front of the free list. No other functionality other than modifying the called method vm_freelist_add_rear() was added (see Fig. 14). See the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (above).

The call path to place inactive and invalid pages on the front of the list is as follows: vm_page_free_rear() calls vm_page_free_toq_rear(), which calls vm_phys_free_pages_rear(), that calls vm_freelist_add_rear(), where the actual queue modification occurs.

```
fnc: vm_phys_free_pages_rear()
     …
     place inactive and invalid pages on the front of the free list (call vm_freelist_add_rear())
end fnc
```

Fig. 14

---------- **sys/vm/vm_page.h** ----------

We have included the declaration, as required, of the three new functions that were added in vm_page.c. See their respective sections for a description. See Fig. 15 for declarations.

```
void vm_page_deactivate_front (vm_page_t)
void vm_page_free_toq_rear(vm_page_t m)
void vm_page_free_rear(vm_page_t m)
```

Fig. 15

---------- sys/vm/vm_page.c ----------

vm_page_free_rear()

As part of the call path to place inactive and invalid pages on the front of the free list, this method was added. Please, see the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (above). The only addition/difference between this method and the original is a call to vm_page_free_toq_rear() rather than vm_page_free_toq(). See Fig. 16.

```
fnc: vm_page_free_rear()
    …
    place inactive and invalid pages on the front of the free list (call vm_page_free_toq_rear())
end fnc
```

Fig. 16

vm_page_free_toq_rear()

As part of the call path to place inactive and invalid pages on the front of the free list, this method was added. Please, see the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (above). The only addition/difference between this method and the original is a call to vm_phys_free_pages_rear() rather than vm_phys_free_pages(). See Fig. 17.

```
fnc: vm_page_free_rear()
    …
    place inactive and invalid pages on the front of the free list (call vm_phys_free_pages_rear())
end fnc
```

Fig. 17

_vm_page_deactivate_front()

Since, pages that are moved into the inactive list it now go in the front rather than the rear the inserts in the original method _vm_page_deactivate() required modification, simply because it is where pages are moved to inactive queue. As per our overall design decision we created this duplicate function rather than modifying the existing one. Please, see the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (above). Since, this function is called with an *athead,* where *athead* is variable passed from vm_page_deactivate_front(), value of 0 (or False in C) from

the call path of the page daemon, our changes consisted of flipping the insertions. See Fig. 18. These "flips", which consisted of inverting the insert statements, match design specs and Professor Long's instructions. Other than flipping the insert statements no other distinguishing changes were made to this duplicate method.

Notably, the modified call path to move place a page in the front of the inactive list is:
vm_page_deactivate_front() -> _vm_page_deactivate_front()

```
fnc: _vm_page_deactivate_front()
     …
     if athead
        insert into the tail
     else
        insert into the head
     end if/else block
     …
end fnc
```

Fig. 18

vm_page_deactivate_front()
As part of the call path to place pages that are moved into the inactive list on the front rather than the rear this method was added. Please, see the Duplicate Methods subsection for our reasoning on why created duplicate methods rather than modifying preexisting methods (above). The only addition/difference between this method and the original is a call to _vm_page_deactivate_front() rather than _vm_page_deactivate(). See Fig. 19.

```
fnc: vm_page_deactivate_front()
     insert the page in the front of the inactive queue (call _vm_page_deactivate_front())
end fnc
```

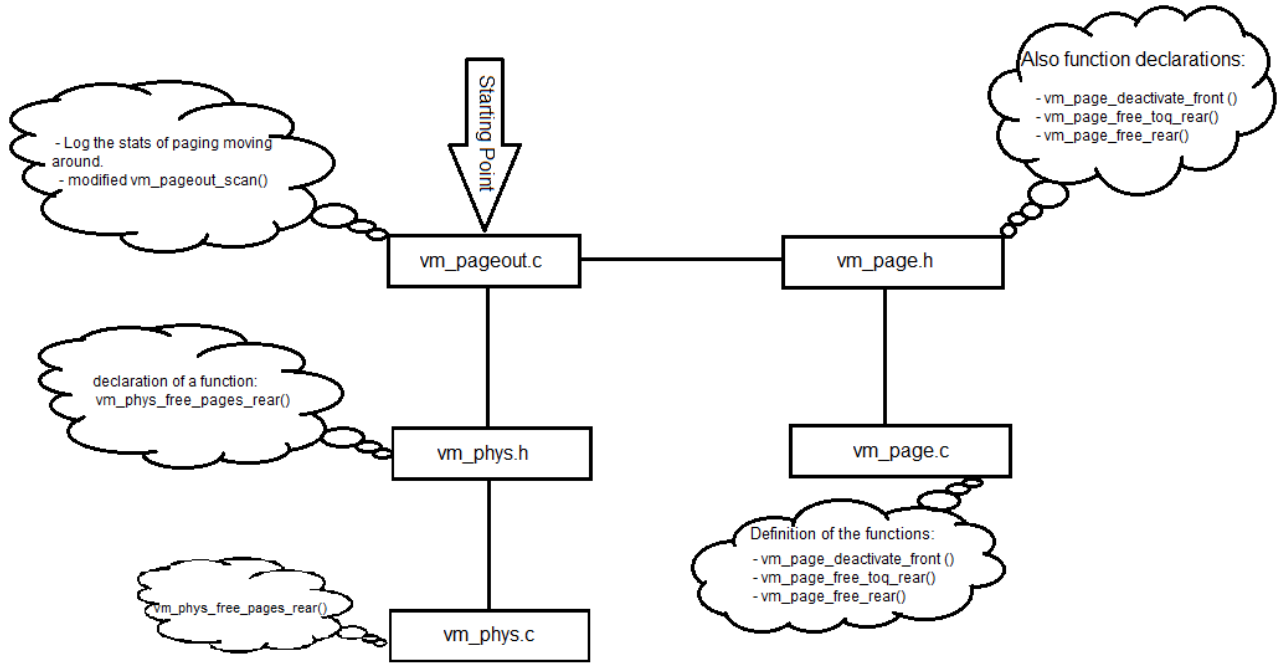
Fig. 19

Testing:

See WRITEUP.pdf for a detailed explanation of our testing procedures, hypothesis and findings.

See Next Page For workflow and pictorial overview.

Essential workflow:



Pictorial Overview:

Here are the files that we have modified under **sys/vm** directory: