CMPS111 Asgn 3

Barron Wong & John Runnberg
bwong20@ucsc.edu jrunnber@ucsc.edu

February 22, 2019

Introduction

In assignment 3 we took a look at FreeBSD's memory management at the page level. We were tasked with modifying the pageout daemon, so that it would pageout pages in order of oldest to newest (FIFO).

1 Modifications

For this assignment we altered 3 files of the FreeBSD operating system source:

src/sys/vm/vm_page.c
src/sys/vm/vm_page.h
src/sys/vm/vm_pageout.c

In vm_page.c and vm_page.h we made changes to data structures and initialization functions to accommodate the addition of a timestamp to pages, additionally, we removed the functionality of the active queue. For the purposes of our FIFO algorithm it makes the most sense to have only one queue, ordered from oldest to newest. In the vm_page_enqueue() function, in vm_page.c, we route all pages that would be queued into active queue to the inactivate queue instead, ensuring that all pages be sent to the two queues go to the inactive queue.

In vm_pageout.c we changed 3 functions: vm_pageout_flush(), vm_pageout_launder(), and vm_pageout_scan(). Additionally, we removed the marker from the inactive queue. Most of the changes to this module occur in the function vm_pageout_scan(). Firstly, we removed code regarding the reference counts since they are unneeded for a single FIFO queue. We also remove the code responsible for scanning the active queue, since we no longer use it. Our changes in the other two functions, vm_pageout_flush() and vm_pageout_launder(), force all pages sent to the laundry queue to be cleaned and freed, as opposed to having an option to return to the active/inactive queue, as in FreeBSD's algorithm.

1.1 FIFO Algorithm

Our algorithm puts all pages into a single queue upon their creation and inserts pages at the back, ensuring that pages are ordered from oldest to newest. When vm_pageout_scan() is called, we remove non-busied pages starting from the front of the queue to meet our shortage. Dirty pages will still go to the laundry queue, but will always be freed once cleaned.

We maintained some of the FreeBSD pageout_scan's structure, removing parts which dealt with page activity and reference counts.

2 Logging

Each time a page is placed into the FIFO, a timestamp is taken of the current OS time. These timestamps will be referenced each time statistics of the FIFO are printed during a pageout scan. The statistics shown are the total number of pages in the queue and the elapsed time of the head and tail pages.

```
Pageout: Size 919066 Head 56 Tail 4
Pageout: Size 917530 Head 57 Tail 5
Pageout: Size 915994 Head 58 Tail 6
Pageout: Size 915994 Head 59 Tail 7
Pageout: Size 915994 Head 60 Tail 8
Pageout: Size 915994 Head 61 Tail 9
Pageout: Size 915994 Head 62 Tail 10
Pageout: Size 915994 Head 63 Tail 11
Pageout: Size 915994 Head 64 Tail 12
Pageout: Size 915994 Head 65 Tail 13
Pageout: Size 914458 Head 66 Tail 14
Pageout: Size 912922 Head 67 Tail 15
Pageout: Size 912922 Head 68 Tail 16
Pageout: Size 911386 Head 69 Tail 17
Pageout: Size 911386 Head 70 Tail 18
Pageout: Size 911386 Head 71 Tail 19
Pageout: Size 911386 Head 72 Tail 20
Pageout: Size 909852 Head 73 Tail 21
Pageout: Size 909852 Head 73 Tail 21
Pageout: Size 908318 Head 74 Tail 2
Pageout: Size 906782 Head 75 Tail 3
Pageout: Size 906782 Head 76 Tail 4
Pageout: Size 906782 Head 77 Tail 5
Pageout: Size 906782 Head 78 Tail 6
Pageout: Size 905248 Head 79 Tail 7
Pageout: Size 905248 Head 80 Tail 8
Pageout: Size 905248 Head 81 Tail 9
Pageout: Size 905250 Head 81 Tail 3
Pageout: Size 903714 Head 82 Tail 4
Pageout: Size 903714 Head 83 Tail 5
Pageout: Size 903714 Head 84 Tail 6
Pageout: Size 903714 Head 85 Tail 7
Pageout: Size 902178 Head 85 Tail 7
Pageout: Size 900642 Head 86 Tail 8
Pageout: Size 900642 Head 87 Tail 9
Pageout: Size 900642 Head 88 Tail 10
Pageout: Size 900642 Head 89 Tail 11
Pageout: Size 899106 Head 90 Tail 12
```

Figure 1: Logging

As seen in the figure above, the head of the queue is always older than the tail, this is expected behavior for our FIFO implementation. At a glance there is a lot of data that can be hard to analyze by inspection, the sections below shows how we benchmarked and analyzed the data using a MatLab script.

3 Benchmarking

Two program were written for bench marking purposes and a script was written in MatLab to analyze the data.

The first program is called "benchmark", this is where bulk of the memory stress is. The program began by using a system call to determine the amount of physical memory the system has. The benchmark program takes the total amount of memory and divides it by 20. It then allocates two arrays that are each a twentieth of the total memory size and performs random accesses and insertions.

The second application is called "stress". This program creates 10 forks of the benchmark program and runs it for 5 minutes. The amount of memory used in the 11 forks is greater than the total memory, so that page fault will happen more frequently.

```
last pid:   732;  load averages:  3.61,  4.16,  2.82
34 processes:  12 running, 22 sleeping
CPU: 95.3% user,  0.0% nice,  3.5% system,  1.2% interrupt,  0.0% idle
Mem: 40M Inact, 3671M Laundry, 151M Wired, 69M Free
ARC: 47M Total, 18M MFU, 28M MRU, 172K Anon, 244K Header, 1151K Other
     12M Compressed, 35M Uncompressed, 2.93:1 Ratio
Swap: 2048M Total, 165M Used, 1883M Free, 8% Inuse, 63M In, 6144K Out

  PID USERNAME    THR PRI NICE   SIZE    RES STATE    TIME    WCPU COMMAND
  729 bwong20       1  75    0   390M   383M RUN      0:02  19.19% benchmark
  730 bwong20       1  75    0   390M   383M RUN      0:02  18.02% benchmark
  728 bwong20       1  75    0   390M   383M RUN      0:02  16.93% benchmark
  731 bwong20       1  75    0   390M   383M RUN      0:03  16.35% benchmark
  721 bwong20       1  75    0  6288K   488K RUN      0:02  13.34% stress
  724 bwong20       1  52    0   390M   368M RUN      0:01   4.96% benchmark
  723 bwong20       1  52    0   390M   346M RUN      0:01   4.65% benchmark
  722 bwong20       1  52    0   390M   352M RUN      0:01   0.58% benchmark
  727 bwong20       1  72    0   390M   376M RUN      0:01   0.48% benchmark
  726 bwong20       1  52    0   390M   373M RUN      0:01   0.47% benchmark
  725 bwong20       1  52    0   390M   371M RUN      0:01   0.41% benchmark
  420 root          1  20    0  6420K  1068K select   0:00   0.09% syslogd
  347 root          1  20    0  9180K  1528K select   0:00   0.08% devd
  697 bwong20       1  20    0 13160K   144K select   0:00   0.04% sshd
  732 bwong20       1  20    0  7916K  1496K RUN      0:00   0.03% top
  692 bwong20       1  20    0 13160K    32K select   0:00   0.00% sshd
  627 root          1  20    0 10452K   136K select   0:00   0.00% sendmail
  693 bwong20       1  20    0 10128K    32K select   0:00   0.00% sftp-server
  689 root          1  27    0 13160K    32K select   0:00   0.00% sshd
  698 bwong20       1  20    0  7064K     0K wait     0:00   0.00% <sh>
  694 root          1  26    0 13160K    32K select   0:00   0.00% sshd
  634 root          1  20    0  6464K     0K nanslp   0:00   0.00% <cron>
  681 root          1  52    0  6408K    16K ttyin    0:00   0.00% getty
  684 root          1  52    0  6408K    16K ttyin    0:00   0.00% getty
  624 root          1  24    0 12848K    32K select   0:00   0.00% sshd
  686 root          1  52    0  6408K    16K ttyin    0:00   0.00% getty
  685 root          1  52    0  6408K    16K ttyin    0:00   0.00% getty
  688 root          1  52    0  6408K    16K ttyin    0:00   0.00% getty
  687 root          1  52    0  6408K    16K ttyin    0:00   0.00% getty
```

Figure 2: Benchmarking: Top

The figure above shows the output of top after the stress program is ran. Notice the low amount of free memory that is available, this forces the OS to perform more pageout scans to make room for incoming faults.

After benchmarking the memory for 3 minutes, the stress test uses dmesg to log all of the kernel's print statements into a file called "rawdata.txt". The program then takes the raw data and removes any other unrelated prints from the kernel and saves it into a filed called "data.txt".

4 Analysis

Once the "data.txt" file is produced, it can be analyzed using the MatLab script, "benchmark.m". The script takes each of the three statistics being measured and places each of them into their own array; these arrays are then plotted in order to view the queues behavior during the benchmark.

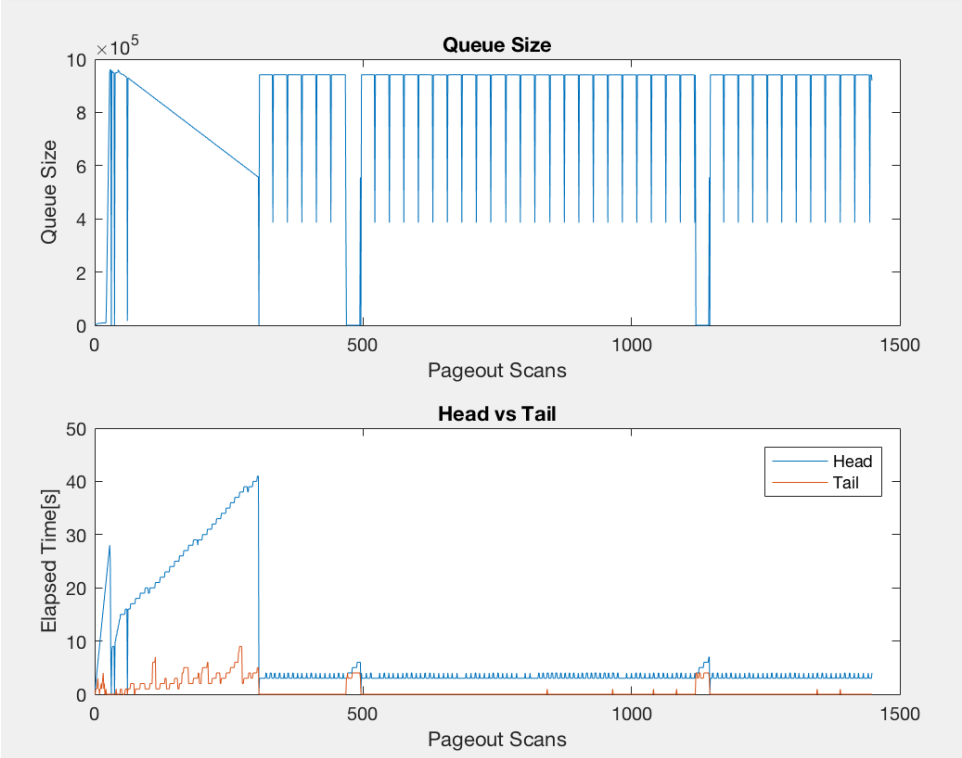

Figure 3: FIFO Stats

The graph above shows how the queue size, head, and tail, as they are changing through each page out scan. The data is taken from a 3 minute run of the stress program.

The first graph shows how the queue size changes while the stress is running. At first the queue is over loaded with the amount of pages being accessed, here you can see the queue size grow to almost 1,000,000 pages in size. The kernel then begins to page out items in the queue to make room for more pages. After going below a certain threshold, the kernel seems to find a balance state where it takes on about 1,000,000 pages, purges half of them, so that new ones can come in. This seems as expected behavior of the page out algorithm; as more pages are needed the queue decreases to compensate for them and quickly after the queue get completely filled with the newly faulted pages.

The second graph displays the change in elapsed time in the head and tail pages. As it can be seen in the charts, the page at the head always has a higher elapsed time than the page at the tail. This is to be expected since our queue has been implemented as a FIFO; the tail is always the youngest page and the head is always the oldest. In the beginning of the graph one can see that as the queue grow larger, the amount of time spend in the queue is lower; this is better seen in the next graph.

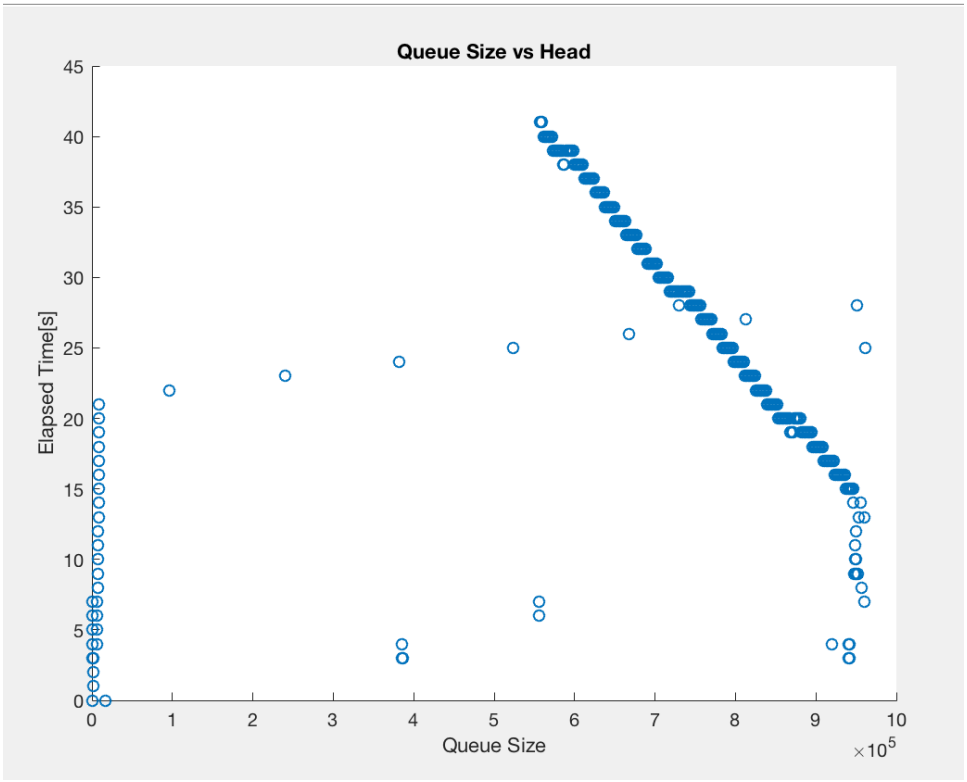

Figure 4: FIFO Stats

The next plot that is made has the queue size as its X axis and the amount of

time the head page has spent in the queue as its Y axis. The graph shows that as the size of the queue increases, the elapsed time of the head page decreases; this is inline with our expectations. Since the amount of memory allocated in our stress program is larger than the amount of physical memory the system has, more page outs are necessary to keep up with memory demands.

Conclusion

This assignment has given us insight into various levels of abstraction and complexity found in the FreeBSD source. Understanding their paging algorithms required understanding of functions and interactions in many source files. Though our modifications were reductive, in a sense, we gained knowledge in keeping the promises abstraction offers in such a vast system. We also gained more, valuable experience in benchmarking, as we collected statistics and used them to verify our algorithm.