Page Replacement

Prof. Darrell Long

Jack Baskin School of Engineering
University of California, Santa Cruz

Due: 3 March at 1700h

Goals

The primary goal for this project is to experiment with the FreeBSD *pageout daemon* (§6.12 of the FreeBSD book) to modify how it selects pages for replacement, and to see how effective the policy is.

As with Assignment 2, this project will give you further experience in experimenting with operating system kernels, and doing work in such a way that when done incorrectly will almost certainly crash a computer.

Basics

The goal of this assignment is to change the behavior of the *pageout daemon* in FreeBSD, which is responsible for deciding which pages to make available for replacement and which pages to flush back to disk. Obviously, this matters most in a system where there's virtual memory pressure, so you'll probably want to experiment using programs that consume (*slightly*) more memory than your VM has available.

Details

The current virtual memory algorithm is a relative of the CLOCK algorithm that uses two queues. Pages have an activity count with a maximum value of 64 associated with them. The general rule is that a page that is found to be referenced in both this scan and the previous scan is moved to the next lower queue, and a page that's found to be referenced in neither this pass nor the previous pass is moved to the next higher queue. Pages that are referenced in exactly one of the previous two passes remain in their current queue. Pages in the highest queue that are not referenced and dirty are queued for write (remaining in the highest queue), and pages that are in the highest queue and unreferenced in this scan and the previous scan are moved to cache, much as pages in INACTIVE are moved to cache in the current pageout algorithm. New pages are placed into Queue 0 and migrate to higher-numbered queues as their time between references decreases.

Well, that's all well and good, except you're going to implement the *FAT CHANCE* algorithm. It's like *SECOND CHANCE*, except it's not. Instead of putting inactive and invalid pages on the front of the free list, you will look at the page number if it is *even* put them on the *front*, and if it is *odd* then put them on the *rear*. Instead of subtracting from the activity count, you will *divide it by two* and move it to the *front* of the active list instead of to the rear. Finally, when you move a page to the inactive list, it goes on the *front* instead of the rear.

You'll need to run pageout more frequently than it does currently (about every 10 seconds) in order to see much effect; this will also require modifying the kernel.

Most of the VM-relevant code is in `sys/vm`, and the page queues are only referenced in `vm_pageout.c` and `vm_page.c` as well as `vm_page.h`. You may need to modify code elsewhere, but look here first.

As part of the code, pageout should write statistics about its performance to the system log. This should include, for each run of pageout the number of pages in each queue as well as the queues scanned in that run and, for each queue, the number of pages moved to a higher or lower queue. You should also print the number of pages moved to the cache and the number of pages queued for flush. You can also get information using the `sysctl()` system call.

You'll also need to modify the existing pageout code to print some statistics: the number of pages scanned, the number of pages moved from ACTIVE to INACTIVE and INACTIVE to cache / free, and the number of pages queued for flush. This is done for each run of pageout. We recommend doing this to the existing kernel *before* modifying it to add your code, and tagging the commit in which this code works. Then, run code to stress the memory system; we have code available that will provide this stress.

Building the Kernel

Rather than write up our own guide on how to build a FreeBSD kernel, we'll just point you at the guide from the FreeBSD web site. You don't need to worry about taking a hardware inventory, if you don't remove any drivers from the kernel you build (and there's no reason you should do this). Focus on Sections 9.4–9.6, which explain how to build the kernel and how to keep a copy of the "stock" kernel in case something goes wrong. Of course, we're happy to help you with building a kernel in laboratory section or office hours.

A couple of suggestions will help:

- Try building a kernel with no changes first. Create your own `config` file, build the kernel, and boot from it. If you can't do this, it's likely you won't be able to boot from a kernel after you've made changes.
- Make sure all of your changes are committed before you reboot into your kernel. It's unlikely that bugs will kill the file system, but it can happen. Commit anything you care about using `git`, and push your changes to the server before rebooting. "*The OS ate my code*" isn't a valid excuse for not getting the assignment done.

As before, your repository (checked out from `git`) contains all the code you'll need. Don't check out a new version!

Deliverables

Select one team member as the **CAPTAIN**. This person is the only one in whose repository work will be done.

Each team member must do the following:

1. Switch to the **master** branch in their own directory:

```
git checkout master
```

2. Create a file `assgn3.txt` containing the names and CruzIDs of each team member, with **CAPTAIN** next to the captain's name. For example, a file might look like this:

```
kmgreen (Kevin Greenan)
awleung (Andrew Leung)
mstorer (Mark Storer, CAPTAIN)
wildani (Avani Wildani)
```

3. Add `assgn3.txt` to the repository, commit it, and push it:

```
git add assgn3.txt
git commit –am "Team file for Assignment 3"
git push --all
```

4. The team captain (and ***only*** *the captain*) now needs to set up a branch for the assignment in his/her directory and push it to the server:

```
git branch assgn3 initial_repo
git push --all
```

5. The team captain needs to give each team member write access to the repository using a command that looks like this:

```
ssh git@git2.soe.ucsc.edu perms classes/cmps111/winter16/captain_cruzid + WRITERS team_member_cruzid
```

   This needs to be done once for each team member.

6. Each team member (other than the CAPTAIN, who already has a copy) needs to clone the CAPTAIN's repo:

```
git clone git@git2.soe.ucsc.edu:classes/cmps111/winter16/captain_cruzid
```

You may rename the repo anything you want; it remembers where it came from when you push changes.

As you work on the assignment

Switch to the appropriate branch (if necessary—`git` remembers the last branch you were on:

```
git checkout assgn3
```

Repeat as needed:

- Make changes to one or more files.
- Add one or more new files to the repository:

```
git add file1 file2 ...
```

- Commit all of your changes to the repository:

```
git commit -am "Your commit message goes here"
```

- As desired, push all of your changes to the `git` server. This includes *all* commits you've already made, not just the most recent one. Of course, only those that are "missing" on the remote side are actually sent.

```
git push --all
```

Testing your project

You've got print-outs that list how many pages are scanned / moved for both your code and the existing FreeBSD pageout daemon. Run the test programs and see how many pages get scanned and moved for your code as well as the standard FreeBSD code. Please write up your findings (text and graphs) and submit them as WRITEUP.pdf.

Submitting your project

For team assignments, this only must be done once on the captain's repository. Any team member may do this step.

Submit your assignment by running these three commands:

```
git push --all
git tag -a assgn3_submission -m "Submission for Assignment X"
git push --tags
```

As with other assignments, we'll grade the last tag before the deadline, or the first tag after the deadline. Your team can get extra credit for submitting early, but only if you tag exactly one commit. If you tag multiple commits for submission, you don't get extra credit. Also, if you forget to tag your commit for submission, we'll assume your team is using all of their remaining grace days, so make sure you tag your code to submit it.

One final thing: you need to write up a few paragraphs describing what *you* contributed to the group effort, and how you'd rate the other members of your group. Be honest, but nice, since the others can read this file. This (one) plain-text (ASCII) file must be submitted in the README.CruzID file that is part of the submission. The goal here is for us to understand how each person contributed to the group effort. We don't expect everyone to have done the same thing, but we expect that everyone will contribute to the project.

Hints

- *START NOW*! Meet with your group *NOW* to discuss your plan and write up your design document. design, and check it over with the course staff.
- *EXPERIMENT*! You're running in an emulated system—you can't crash the whole computer (and if you can, let us know...).
- Test your memory allocation. To do this, write a program that uses slightly more than 1/*n* of your virtual memory space, and run *n* copies of it. We'll provide such a program, but please feel free to use your own. This project doesn't require a lot of coding (typically several hundred lines of code), but does require that you understand FreeBSD and how to use basic system calls. You're encouraged to go to the class discussion section or talk with the course staff during office hours to get help if you need it.

IMPORTANT: As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it's impossible to complete a 20-hour project in the remaining 12 hours before it's due.

IMPORTANT: The README.Captain file should contain any special instructions that we should know, and your view of your own contributions and those of your teammates.