Project 2: I/O Elevators

Jui-Hung Lu Yi Li Rahul Borkar

May 6th, 2018

CS544: Operating Systems II
Spring 2018

Abstract

In this second assignment we implement the C-LOOK elevator using the existing noop I/O scheduler code as a skeleton. We then rebuild our kernel from Project 1 and run the emulator using our new elevator.

1 DESIGN

This assignment required additional research to determine the exact characteristics of a LOOK or C-LOOK I/O Scheduler. Our first task was to choose an algorithm. We decided to implement C-LOOK because it tends to be less biased than LOOK.

CLOOK is similar to CSCAN disk scheduling. In CLOOK, the disk arm only scans in one direction, which allows it to be less biased than LOOK based algorithms. LOOK may not service a request that is far away for a long time, while CLOOK will be much more consistent in this regard.

To start our design, we looked at the *noop_iosched.c* file. The bulk of our work went into designing the *dispatch* and *add_request* functions. When adding a new request is being added, we use first check if the queue is empty. If it is, we can immediately put the request into the queue. If not, we determine the correct position to place it in according to the current head of the queue by looping through the current queue.

To handle dispatching requests, we first check if the queue is not empty. If it isn't then we service the head of the queue, remove it, do a dispatch sort, and update the queue head.

2 VERSION CONTROL LOG

Date	Commit	Message
5/3/18	37bebf	Start HW2
5/3/18	da8003	Update Makefile and Kconfig
5/6/18	a9b93b	Done with code
5/6/18	071921	Fixed patch file

3 WORK LOG

Date	Hours	Work Done
5/1/18	1	Group meeting, discussed which algorithm to use and started pseudocode
5/3/18	4	Finished implementing solution
5/6/18	2	Finished writeup, patch created, submitted

4 QUESTIONS

4.1 What do you think the main point of this assignment is?

There are a couple of things we learned from this assignment. First, we gained some insight into how block I/O works and specifically how it is handled in the Linux kernel. Knowing how to handle block I/O is important for doing more complex kernel programming.

Another big step for this assignment was learning how to modify and rebuild the Linux kernel and to learn how to debug our modifications. This portion of the assignment was very difficult, because it can be difficult to tell if things aren't working if proper debugging and testing have not been set up.

4.2 How did you personally approach the problem? Design decisions, algorithm, etc.

As mentioned before, we decided to use CLOOK because it is generally fairer and the linear operation seemed simpler to us.

From here we read up on the exact operation of the algorithm, and pseudocoded our solution until we agreed on a general algorithm. While our algorithm was sound for the most part, there were a couple of issues we did not consider, like the queue being empty.

The hardest part by far was debugging. We knew that our solution was good, or at least close to correct, but for some reason our printk statements were not being output to the kernel beyond the queue initialization. It turns out this was an issue with how we were booting up our VM, and after fixing that using the qemu documentation[1] we could see our full console output.

4.3 How did you ensure your solution was correct? Testing details, for instance.

There were a few steps to testing our solution. First, we added printk statements after any operation was completed by the scheduler, so we could see a verbose output of what it was exactly doing. Within these statements, we included whether it was reading or writing, and what sector it was performing that operation on. With this, we could easily tell if it was working correctly; it should have been servicing requests in a linear fashion and not skipping things or going backwards. We we re able to verify this by booting up the VM and doing some basic I/O operations by hand and using dmesg — grep SSTF to observe our algorithm working correctly.

4.4 What did you learn?

In this assignment, we learned how to implement our own block I/O scheduler, and rebuild and boot the Linux kernel with our changes. We also learned some of the differences between some of the block scheduling algorithms that are commonly used. As mentioned earlier, we also learned the importance of properly setting your VM flags to boot the way you need to; we forgot to turn off virtio which unnecessarily wasted a lot of time. In the future we will be more careful in examining our VM settings.

4.5 How should the TA evaluate your work? Provide detailed steps to prove correctness.

1) Set up the VM as HW1 specified, including running the source activation script, but do not build the kernel.
2) From the *linux-yocto-3.19* directory apply the patch from our submission by running: git apply *hw2_group16.patch*
3) Run the following command:
   make menuconfig
4) From the menuconfig go to block layer – IO schedulers – default IO scheduler – select SSTF.
   Save this to .config and exit menuconfig.
5) Rebuild the kernel with:
   make -j4 all
6) Run the VM with the following command:
   qemu-system-i386 -nographic -kernel arch/x86/boot/bzImage -drive file=core-image-lsb-sdk-qemux86.ext4 -enable-kvm -net none -usb -localtime –no-reboot –append ”root=/dev/hda rw console =ttySO debug elevator=sstf”
7) Once inside the VM, the following command can be used to see the activity of the IO scheduler:
   dmesg — grep SSTF

REFERENCES

[1] [Online]. Available: https://qemu.weilnetz.de/doc/qemu-doc.html