COP 4610 Operating Systems Principles

Lab exercise #4: I/O scheduling

Demo in Lab sessions: U01 - Tue Apr 11th; U02: Fri Apr 14th

Deliverables (one submission per team) due on Apr 14, 2017

1. Objectives

a. understand I/O scheduling

b. do real coding inside the kernel

c. use the data structures provided in the kernel

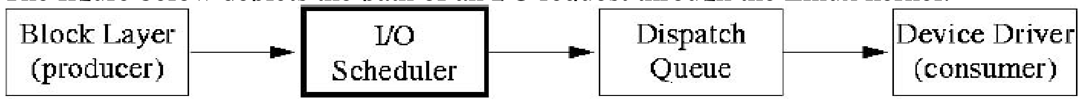
d. test your kernel modules

2. Introduction

In this project, you will implement a **C-LOOK** I/O scheduler. The *C-LOOK* scheduler keeps a list of I/O requests sorted by physical location on the disk. Requests are serviced as the disk head moves across the disk, picking up requests as they are encountered. When the head has serviced the highest block number, it returns to the lowest block number in the list and proceeds to traverse the disk again … etc. You can refer to section 10.4.5 (page 477) of the text book for more information.

Note: You are not required to handle the problem of excessive wait time.

To be able to write an I/O scheduler, you will need to understand how I/O requests flow in Linux. The figure below depicts the path of an I/O request through the Linux kernel:



You will write the I/O scheduler portion of this chain by implementing several of the functions represented in **`struct elevator_type`**, which is defined in **`include/linux/elevator.h`**. The arrows going into and out of the I/O scheduler portion of the chain correspond, respectively, to submitting an I/O request to the scheduler and the scheduler servicing a request, and these functions are performed, respectively, by **`elevator_add_req_fn`** and **`elevator_dispatch_fn`**.

Your **`elevator_add_req_fn`** is invoked by **`elv_insert in block/elevator.c`**, and your **`elevator_dispatch_fn`** is invoked by **`elv_drain_elevator`** in **`block/elevator.c`,** in a while loop.

You can build your *C-LOOK* scheduler on the *noop* scheduler in the current Linux kernel (see **`block/noop-iosched.c`**). You can keep the current **`noop_dispatch`**, which dispatches one request unless the I/O scheduler queue (**`nd->queue`**) is empty. If you look at the while loop in **`elv_drain_elevator`**, you can see that it keeps calling **`q->elevator->ops->elevator_dispatch_fn`** (which is actually **`noop_dispatch`** when the *noop* scheduler is chosen) until it returns 0. Therefore, you do not have to modify **`noop_dispatch`**. You can modify your **`elevator_add_req_fn`** to implement C-LOOK. Right now, **`noop_add_request`** simply append the new request to the end of the I/O scheduler queue, but you may need to insert the new request anywhere in the current scheduler queue.

You may wonder how **`elevator_add_req_fn`** and **`elevator_dispatch_fn`** are supposed to play together. They are invoked in two distinct kinds of interleaving phases: the sorting phase and the dispatching phase. **`elevator_add_req_fn`** is invoked in the sorting phase and **`elevator_dispatch_fn`** is invoked in the dispatching phase. In the sorting phase, all the incoming requests are sorted in the I/O scheduler queue according to some algorithm (e.g., C-LOOK algorithm), and in the dispatching phase, all the sorted requests in the I/O scheduler queue are dispatched to the device driver. These two phases are mutually exclusive (synchronized by **`q->queue_lock`**), which means that in the dispatching phase, there will be no invocations to **`elevator_add_req_fn`** to insert new requests, but between two dispatching phases, there can be 0 or more invocations to **`elevator_add_req_fn`**. Therefore, your **`elevator_add_req_fn`** must make sure that it always leaves the I/O schedule queue in a properly sorted order ready for dispatching.

For more information, see section 4.1 of **`Documentation/block/biodoc.txt`**.

3. Tasks

3.1. Read through Chapter 14 (Block Device Drivers) of the Bovet book and review the `block/noop-iosched.c` I/O scheduler, which is simply FCFS.

3.2. Outline your *plan of attack* describing the changes you will need to make to `block/noop-iosched.c` so that it runs the *C-LOOK* I/O scheduling algorithm.

3.3. Create a new version of the I/O scheduler as described below.

a. Make a working copy of the *noop* I/O scheduler. The source code is in **`block/noop-iosched.c`**. Name your copy **`clook-iosched.c`**. Be sure that your **`clook-iosched.c`** header block contains your group members' names, and a description of the changes made. Change the fields of the author and description macros appropriately.

b. Set up the kernel as follows to recognize your scheduler at build time:

- Add the following line to **`block/Makefile`** so that your scheduler is compiled if it is configured:

```
obj-$(CONFIG_IOSCHED_CLOOK) += clook-iosched.o
```

- Edit **`block/Kconfig.iosched`** as follows to create configuration options for your scheduler in `make config`, etc.:

  - Copy the existing "config IOSCHED_DEADLINE" block and paste it as "config IOSCHED_CLOOK" before the "choice" block. Change the argument of the tristate statement to read "CLOOK I/O scheduler", and edit the help information appropriately. Also, add a section to the "choice" block that reads

```
config DEFAULT_CLOOK
     bool "CLOOK" if IOSCHED_CLOOK=y
```

c. Modify **`clook-iosched.c`** so that I/O requests are serviced as described in the introduction. Be sure that identifier names start with **`clook`** instead of **`noop`**. As you write or modify functions, add function headers describing what those functions do. Use the kernel's linked-list implementation for your scheduling queue.

d. Look at the **elevator_ops** structure in **include/linux/elevator.h** to see what functions comprise the I/O scheduler API. You will likely have to implement at least the following functions:

   i. **elevator_init_fn**: allocates and initializes any data structures or other memory you will need to make your scheduler work, for example, a **list_head** structure to represent the head of your sorted request list; called when your scheduler is selected to handle scheduling for a disk

   ii. **elevator_add_req_fn**: takes an I/O request from the kernel and inserts it into your scheduler in whatever sorted order you choose.

   iii. **elevator_dispatch_fn**: takes the next request to be serviced from your scheduler's list and submits it to the dispatch queue.

   iv. **elevator_exit_fn**: deallocates memory allocated in **elevator_init_fn**; called when your scheduler is relieved of its scheduling duties for a disk

   v. **elevator_queue_empty_fn**: tells the kernel whether or not your scheduler is holding any pending requests

   vi. Probably others, such as **elevator_former_req_fn** and **elevator_latter_req_fn**

e. Use **printk()** statements to prove that your scheduler is working. In particular, every time an I/O request is added to your schedulers "list", issue a **printk()** statement formatted exactly like this:

```
[CLOOK] add <direction> <sector>
```

   where `<direction>` is either R for read or W for write, and `<sector>` is the request's first disk sector. Also, whenever a request is moved into the dispatch queue, issue a **printk()** statement formatted exactly like this:

```
[CLOOK] dsp <direction> <sector>
```

   where `<direction>` and `<sector>` are as defined above. **printk()** statements will show up in the message log, which you can check with **dmesg** or by viewing **/var/log/messages**.

f. Run **`make menuconfig`**, navigate to *Block layer* --> *IO Schedulers*, and mark your scheduler to be compiled as a module. Make sure you leave the *Default I/O scheduler* selection alone. Save your configuration changes.

g. Compile the kernel using **`make && make modules_install`**. Assuming that your kernel hasn't changed, except for the addition of the new I/O scheduler, there shouldn't be much compilation necessary (except the first time you compile). Also, if your kernel hasn't changed, you shouldn't need to go through the steps to install the kernel image in the boot directory, except after the first time you compile.

h. Insert your I/O scheduler module with

`insmod clook-iosched.ko`.

You can later remove the module with

`rmmod clook-iosched`. Follow Section 4 "Testing an I/O scheduler module" below to make the kernel use your scheduler to handle I/O scheduling on **/dev/sdb**. Create traffic on **/dev/sdb** (which should be mounted at **/test**) to test your scheduler.

4. **Testing an I/O scheduler module**

If you've inserted an I/O scheduler module into the kernel, it won't automatically be used. You must explicitly tell the kernel to use your I/O scheduler for a specific disk. Here are instructions on how to do so. You need to be "root" to try the following steps.

a. Assuming that you have built and inserted an I/O scheduler module named `clook-iosched.ko`. I.e., `insmod clook-iosched.ko.` Let's also assume you want to use your scheduler on `/dev/sdb`. First, list the I/O schedulers available for **sdb**, and make sure yours is available:

```
% cat /sys/block/sdb/queue/scheduler
noop [anticipatory] deadline cfq clook
```

We can see that your scheduler (`clook`) is available. The scheduler in brackets is the one currently being used for **sdb**.

b. Now tell the kernel to use your scheduler:

```
echo clook > /sys/block/sdb/queue/scheduler
```

If you list the schedulers again, yours should be bracketed:

```
% cat /sys/block/sdb/queue/scheduler
noop anticipatory deadline cfq [clook]
```

That means I/O scheduling for **sdb** is now being handled by your scheduler. Cool.

c. Before you remove your scheduler module from the kernel, make sure you switch back to another scheduler:

```
echo anticipatory > /sys/block/sdb/queue/scheduler
rmmod clook-iosched
```

5. Additional Requirements, Notes, and Caveats

a. Use version 2.6.36 of the kernel.

b. Any data structure you want to use is already implemented in the Linux kernel. Use Moodle lab discussion forum to help each other figure out how to use the appropriate data structures (the TA, of course, receive forum postings, too).

c. To test your scheduler, you need a hard disk. A 512 MB disk has been prepared that can be downloaded from `http://users.cis.fiu.edu/~prabakar/cop4610/SmallHardDisk.vdi` Make a copy from this file for yourself, and reconfigure your virtual machine to add your SmallHardDisk.vdi as its *second* hard disk. After starting your virtual machine, SmallHardDisk.vdi will be recognized as **/dev/sdb** on your VM. Then become root by doing a "**sudo bash**", run "**mkdir /test**", and then "**mount /dev/sdb5 /test**". If you could find a file named "**hello.c**" under **/test** which has a size of 48 bytes, the second disk is now successfully mounted at **/test**. Use it to test your scheduler thoroughly (follow section 4 to pick up your I/O scheduler for **/dev/sdb** first). Do not just rely on I/O generated randomly as a thorough test of your scheduler. Think about how your scheduler is designed to work and write a small program or script to test it.

d. The Linux block layer relies heavily on caching. A second *read* of a sector will almost always be pulled from cache. *Writes* are not as problematic, since they eventually need to be written to disk. Your scheduler, of course, must work for both *reads* and *writes*. It would be a good idea to test *reads* using a very big file.

e. The evaluation criteria will be posted at least one week before the due date. Be sure to check this posting before making your final submission.

6. Useful resources

Visit the following link to a cross-referenced HTML version of the kernel sources at

http://lxr.free-electrons.com/source/?v=2.6.36 .

You will likely find this highly useful. The following files will be particularly useful:

- **`Documentation/block/biodoc.txt`** (especially section 4)
- **`Documentation/block/request.txt`**
- **`include/linux/blkdev.h`**
- **`block/noop-iosched.c`**
- **`Documentation/CodingStyle`**

Other related files are:

- **`block/elevator.c`**
- **`include/linux/bio.h`**

Check the lab discussion forum. If you don't find what you need, figure it out and add it. Chapter 13 of your textbook will also be highly useful. Check Moodle for other resources.

7. What to turn in for deliverables:

Create a gzipped tar archive containing the items below and submit it in Moodle.

- The source code for your scheduler module
- A patch file named **`clook_iosched.patch`** created using the **`Kernel Patch Guide`** representing the changes you have made to the kernel
- Source code of the test program or script that you developed to generate traffic for your I/O scheduler in section 3.3.h and section 5
- A report on the test result of your C-LOOK I/O scheduler, including the `printk` output log

Tentative evaluation criteria:

Item	Possible points
gzipped `tar` archive submission	1
Submitted **`clook-iosched.c`**, patch file, source code of the test program in task 5 including a Makefile	4
`clook-iosched.c` header block with team member names, program description, etc. Change the fields of the author and description macros appropriately.	4
Useful function headers (for functions changed) o Description of changes o Useful comment outline	4
The responsibility of each team member is clearly stated (in the report or comments)	3
report (clarity, presentation, etc)	15
The **C-LOOK** algorithm is correctly implemented (can compile, can run, and can run correctly)	45
Test program and test cases	20
Clean implementation	4
Cites sources for borrowed code	0 points, but deduct up to 10 points for sources not cited