CS411 Operating Systems II Fall 2008

Project #4

Due 6:00 am, Saturday, 22 November 2008

Objectives:

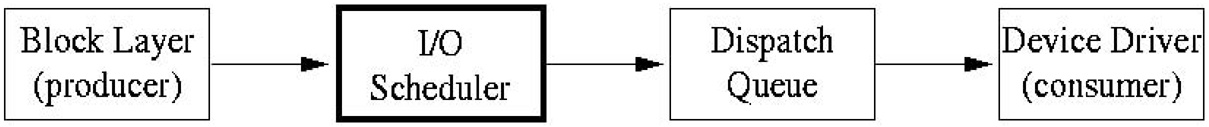
1. understand I/O scheduling
2. do real coding inside the kernel
3. use the data structures provided in the kernel
4. test your kernel modules

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.

Note: You are not required to handle the problem of excessive wait time; this is an extra-credit enhancement.

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`**.

You can build your *C-LOOK* scheduler on the *noop* scheduler used in Project #1. For more information, see section 4.1 of **`Documentation/block/biodoc.txt`**.

Tasks:

1. Read through Chapter 13 of the Love textbook and review the `block/noop-iosched.c` I/O scheduler, which is simply FCFS.
2. Write up 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. Your plan of attack must be submitted by Wednesday, November 12. It will not be evaluated based on the correctness of your proposed plan, but rather on the degree to which it demonstrates that you have examined the existing Linux code and understand roughly how it works.

3. Create a new version of the I/O scheduler as described below.
   a. Create a new **`linux-2.6.23.17-clook_iosched`** branch in your repository. Make a working copy of the *noop* I/O scheduler. The source code is in **`block/noop-iosched.c`** on your virtual machine. Name your copy **`clook-iosched.c`**. Be sure that your **`clook-iosched.c`** header block contains your team number, members' names, and a <u>description of the changes made</u>. 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 one of the existing "config IOSCHED_xxx" blocks 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. <u>Use the kernel's linked-list implementation</u> 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. (Note: Make a typescript of this section.) Insert your I/O scheduler module with **`` `modprobe clook-iosched` ``**. You can later remove the module with **`` `modprobe -r clook-iosched` ``**. Follow the "Testing an I/O scheduler module" section of the "Kernel Module Guide" on the course wiki 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.

Additional Requirements, Notes, and Caveats:

1. ***Do not code directly on your VM!***
2. Use version 2.6.23.17 of the kernel.
3. Any data structure you want to use is already implemented in the Linux kernel. Use the course mailing list to help each other figure out how to use the appropriate data structures (the TAs, of course, receive mailing list traffic, too).
4. Your VM has a 500M disk, **`/dev/sdb`**, specifically for testing I/O schedulers. This disk is mounted at **`/test`**. Use it to test your scheduler thoroughly. 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.
5. 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.
6. Remember, we are working within an open-source code base. That means that at some point in the future, someone else might have to try to understand your code. Follow Greg Kroah-Hartman's kernel coding style guidelines (his slides are available on the course wiki).
7. Frequently commit your changes to your source code repository. Be sure that each member of your team commits changes to the repository at least once.
8. 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.

Useful resources:

There is a link on the class wiki to a cross-referenced HTML version of the kernel sources. 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 student wiki. If you don't find what you need, figure it out and add it (contribution credit is available).

Chapter 13 of your textbook will also be highly useful. Check the course wiki for other resources.

What to turn in:

1. Submit your *Plan of Attack* at http://engr.oregonstate.edu/teach by midnight Wednesday, Nov. 12.
2. Create a gzipped tar archive containing the items below and submit it at http://engr.oregonstate.edu/teach .
   - The source code for your scheduler module.
   - Your compiled kernel image, named **`vmlinuz-clook_iosched-teamXX`**
   - Your compiled scheduler module, named **`clook-iosched.ko`**
   - A patch file named **`clook_iosched-teamXX.patch`** (created using **`` `svn diff` ``**) representing the changes between your **`linux-2.6.23.17-clook_iosched`** branch and the original vanilla kernel
   - The changelog from your source code repository.
   - The typescript from task 2h.
3. Each member of your team must individually write a separate review document. (See the course wiki for the requirements.) This document must be submitted by 11:59 pm on Monday, Nov. 24, at http://engr.oregonstate.edu/teach.
4. Before 9:00 am on Monday, Nov. 24, your team must submit a signed hardcopy of your completed Credit Distribution Agreement.

You should always keep in mind that additional credit is available for contributions to the course mailing list and/or the student wiki. Consider for example the glory that would be yours if you expand the I/O scheduler API on the student wiki.

Extra Credit (Part 1 up to 5 points of extra credit**):**

The problem with a strictly C-LOOK scheduler is that new request may arrive in such a way that requests that came in after the head passed, may have to wait a long time to be serviced. This can happen any time there is constant activity in one area of the disk while requests for lower-numbered blocks are pending. The "enhancement" for this scheduler is to implement aging, which you will handle by assigning a soft deadline to each request. This will make the scheduler more fair, but at a possible cost to throughput.

To keep track of which requests have been waiting too long, each request will be placed into two FIFO queues. Actually, you will use three queues: one for the scheduler, one for reads, and one for writes. I.E., each request will be placed in the *scheduler* queue and in either the *read* queue or the *write* queue. The *read* and *write* queues are FIFO. Each request will be assigned a deadline based on the time it was submitted to the scheduler. Every time the kernel tells the scheduler to dispatch a request, the scheduler will first check the heads of the *read* and *write* queues to determine if there are expired deadlines. If the request at the head of a queue has an expired deadline, the scheduler dispatches that request. Otherwise, the scheduler dispatches a request from the *scheduler* queue, C-LOOK style. Note that every request will be in the scheduler queue and also in one of the FIFO queues, so when a request is dispatched, be sure to remove it from both structures.

1. Make a new copy of your *C-LOOK* scheduler at **`block/fair-clook-iosched.c`**.
2. Implement the aging queues for reads and writes.
3. Revise the appropriate functions to use your aging scheme.
4. Make appropriate changes analogous to those described in the **Task** section.
5. Submit a separate gzipped tar archive as described in the **What to hand in** section.
   Note: The extra credit must be submitted in addition to the original project; it does not replace the original project.

Extra Credit (Part 2 up to 5 points of extra credit**):**

The CLOOK-ordered list is one way to implement a C-LOOK scheduler, but insertion in the list is O(n). A more efficient ordering structure, the red-black tree (RB-tree) is available in the kernel tree. Figuring out how to use the RB-tree structure is not a trivial matter. Here is a link to a very helpful article on RB-trees: http://lwn.net/Articles/184495/

1. Make a new copy of your *C-LOOK* scheduler at **`block/rb-clook-iosched.c`**.
2. Replace the CLOOK-ordered list structure with an RB-tree.
3. Revise the appropriate functions to use the RB-tree.
4. Make appropriate changes analogous to those described in the **Task** section.
5. Submit a separate gzipped tar archive as described in the **What to hand in** section.
   Note: The extra credit must be submitted in addition to the original project; it does not replace the original project.

Credits:

Rob Hess:

- Prototyping, defining, and testing.