Operating System Design & Implementation

Lab 11: I/O Scheduler

TA: 周昆霖、吳崧銘

Contributor: 鄭加祥

Objective:

In this Lab, you can learn

* How to change the default I/O scheduler
* The principle of Linux I/O scheduler
* How to modify I/O scheduler algorithm

Experiment:

* 11-1 Change the I/O scheduler of *sdb* to NOOP and display the order of request service
* 11-2 Modify the NOOP scheduler to be SSTF (shortest seek time first).
* 11-3 Design an access pattern to validate your SSTF.

Exp. 11-1 Change the I/O scheduler of sdb to NOOP and display the order of request service

Step1: Find the function that adds a new request to its internal queue. Print messages about the new request in this function:

printk(“add %llu”, blk\_rq\_pos(rq));

Hint: this function is in the file “block/noop-iosched.c”

Step2: Find the function that dispatches a request from NOOP’s internal queue to the device queue. Print the dispatch message in this function.

printk(“dispatch %llu”, blk\_rq\_pos(rq));

Hint: this function is in the file “block/noop-iosched.c”

Step3: Create a new virtual disk in your VM

1. VM > Settings and click Add.

2. Select Hard Disk, and click Next.

3. Select SCSI, and click Next.

4. Select Create a new virtual disk, and click Next.

5. Set the capacity for the new virtual disk (1G).

6. Set filename and location for the virtual disk and click Finish.

Format your new virtual disk:

1. fdisk /dev/sdb

2. (command m for help)

n: add a new partition (primary partition)

w: save and leave

3. mke2fs /dev/sdb1

4. mount the newly created partition

Step4: Change the I/O scheduler of your new virtual disk to NOOP

echo noop > /sys/block/sdb/queue/scheduler

Step5:

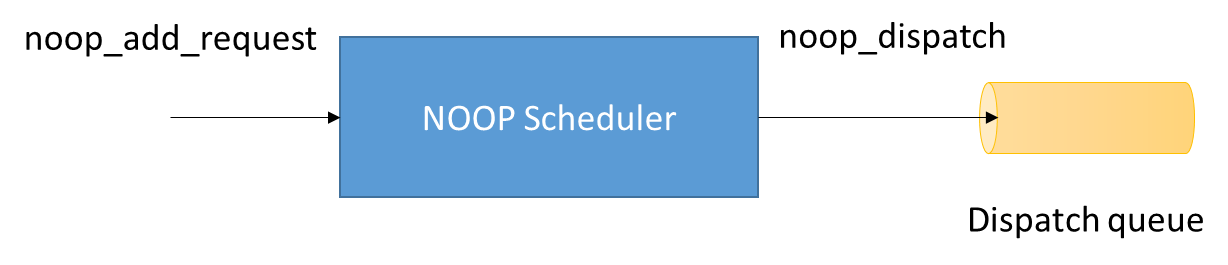
1. Now you can create and delete files or directories in *sdb1*.
2. Use dmesg to display the messages you added in Steps 1 and 2.
3. Now you can see the request-service order of *sdb1* is FIFO.

Exp. 11-2 Modify the NOOP scheduler to be SSTF

Principle: NOOP uses two queues. For example, when noop\_add\_request is called, q->elevator->elevator\_data->queue is the *internal queue* and q is the *device queue* (i.e., dispatch queue). We recommend that you insert new requests to the internal queue in SSTF order and dispatch requests to the device queue in FIFO order.

Two related functions here:

* noop\_add\_request(): adding a request to the NOOP internal queue.
* noop\_dispatch (): moving a request from the NOOP internal queue to the dispatch queue. Notice that it calls elv\_dispatch\_sort() to insert requests to the device queue and sorts the queue in the ascending order.



Step 1: Modify noop\_add\_request() to insert requests to the internal queue in the SSTF order.

Hint: You may need some of the following APIs and data structures

list\_empty(const struct list\_head \*head)

list\_entry(ptr, type, member)

list\_add(struct list\_head \*new, struct list\_head \*head)

list\_move(struct list\_head \*list, struct list\_head \*head)

struct request\_queue -> end\_sector

struct request -> queuelist

struct noop\_data -> queue

blk\_rq\_pos(const struct request \*rq)

// See the kernel source for more details

Step 2: Modify noop\_dispatch () to insert requests to the device queue in

the FIFO order.

Hint: noop\_dispatch() select a request from NOOP internal queue, and call elv\_dispatch\_sort() to insert this request to the device queue. You may need to modify both the two functions, as elv\_dispatch\_sort() will mess your SSTF up.

elv\_dispatch\_sort() is in “block/elevator.c”

Step 3: Rebuild your kernel; reboot.

Exp. 11-3 Write a workload generator to validate your SSTF

Write a program that accesses logical sectors of sdb1. Design your own write pattern that sufficiently validates your SSTF. Notice that we recommend using a kernel module instead of a user-mode program. This is because the page cache sits in the middle between user land and I/O scheduler, and the page cache sometimes unexpectedly re-orders requests from user-mode programs.

Re-use the following kernel module source code and the makefile, and insert your own write pattern in the kernel module.

* Makefile:

obj-m = mymodule.o

KVERSION = $(shell uname -r)

all:

make -C /lib/modules/$(KVERSION)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(KVERSION)/build M=$(PWD) clean

* mymodule.c

#include <linux/module.h>

#include <linux/init.h>

#include <linux/buffer\_head.h>

#include <linux/blkdev.h>

static int set\_size = 512;

static struct block\_device \*bdev;

static int \_\_init init\_read(void)

{

int i=0;

printk("init!!!\n");

bdev = open\_by\_devnum(MKDEV(8,17), 0x08000);

if(IS\_ERR(bdev)){

return -EIO;

}

if(set\_blocksize(bdev,set\_size)){

printk("set block size error\n");

return -EIO;

}

// Design your write pattern here!!

for(i=0; i<100; i++){

\_\_breadahead(bdev, i\*4096, set\_size);

}

return 0;

}

static void \_\_exit exit\_read(void)

{

printk("exit!!!\n");

if (bdev)

{

blkdev\_put(bdev, 0x08000);

bdev = NULL;

}

}

module\_init(init\_read);

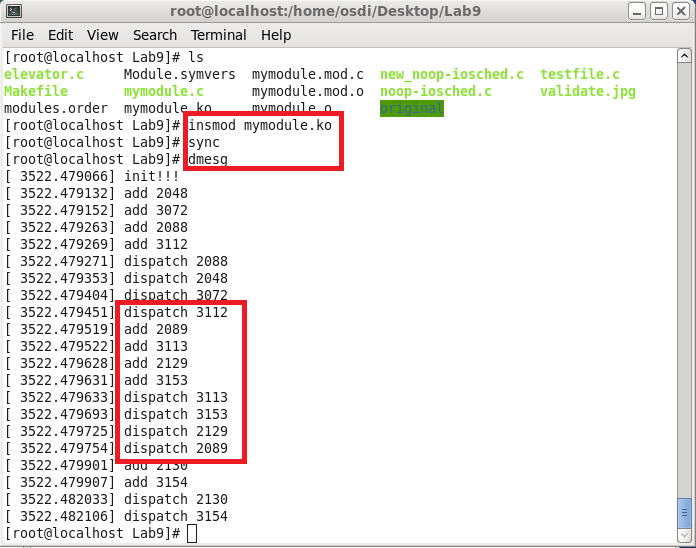
module\_exit(exit\_read);

You should use the “sync” command to force the kernel to write all dirty pages to disks. In addition, before you run the workload generator for the second time, you should clean the page cache using the following command:

echo 3 > /proc/sys/vm/drop\_caches

The kernel will discard all clean pages, including user data pages and metadata pages (such as inode and dentry).

Demonstration



In this example, for the requests in the red box, the latest dispatched request was at 3112. After this, requests at 2089, 3113, 2129, and 3153 arrived (in order). If the scheduler is NOOP, then the dispatch order will be 2089, 3113, 2129, and 3153. But since the scheduler is now SSTF, it will dispatch 3113 which is nearest to 3112, and then 3153, 2129, and 2089.