Project 2: I/O Elevators

Andrew Chase

OS II

Spring 2015

Abstract: Describes work done to build and run a custom Shortest Seek Time First I/O scheduler.

CONTENTS

I. Work Log

Date	Work Done
Apr 26	Write module, compile and test on os-class.

II. Questions

A. The design you plan to use to implement the SSTF algorithms.

The original plan I had was to keep the queue alone, and simply seek the queue each time a dispatch happens to find the request closest to the magnetic reader head. I looked through two other solutions posted online (cited in source), I liked the design of the second solution. It was essentially the same as my idea, but at request time instead of dispatch time. When requests were added, the queue is iterated over and the request is put in the lowest position.

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

I think the main point of the assignment is to use the data structure primitives taught in class and re-enforce some of the I/O scheduler concepts. Although I/O schedulers aren't as important on personal computers as they once were (due to SSDs being more common), the I/O scheduler concepts are still important because they give an example of the kinds of complex problems operating system kernels have to solve efficiently.

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

First I went out and gathered information on the problem and possible solutions. I compared two different solutions to the noop scheduler. One of the two solutions seem to be what I had in mind for how to approach the solution, while the other seemed needlessly complex.

After I implemented and tested the solution I had in mind, the output from the kernel looked ok, but it seemed wrong. There'd be random jumps down to the first sector number. Then, later, after leaving and doing something else I realized that the solution I had used was wrong. By always sorting the lowest sector first in the queue this ignores the fact that the head might be in a high position.

I went back and changed the algorithm so that the queue is kept sorted by the distance from the last head sector that went to disk. This seemed to return much more correct results in testing.

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

I inserted kprint calls and ran the kernel in the emulator. The kprint calls recorded the sector number.

Example:

```
~~~~ 537064
```

```
~~~~ 536744
~~~~ 0
~~~~ 4134152
~~~~ 4134248
~~~~ 4456448
~~~~ 4593168
~~~~ 4718592
~~~~ 5505040
```

The testing helped because the first solution I had implemented wasn't correct and it was pretty obvious to see from the kernel messages. At first, my module didn't load, and using dmesg — grep I was able to find the error message to fix the problem.

E. What did you learn?

I learned a bit about the noop io scheduler work in linux. How to compile a elevator module and have it run in linux.

III. Concurrency Exercise: Git Log

acronym	**meaning**
V	`version`
tag	`git tag`
MF	Number of `modified files`.
AL	Number of `added lines`.
DL	Number of `deleted lines`.

V	**tag**	**date**	**commit message**	**MF**	**AL**	**DL**
1		2016-04-26	init	0	144	0

IV. Code Listing

A. sstf-iosched.c

```
1 /*
2  * elevator stf
3  * References:
4          * http://staff.osuosl.org/~bkero/proj4diff
```

```
 * https://github.com/ryleyherrington/linux_kernel_411/blob/master/sstf-io/sstf
 * http://www.makelinux.net/ldd3/chp-11-sect-5
 * http://lxr.free-electrons.com/source/block/elevator.c#L351
 */
#include <linux/blkdev.h>
#include <linux/elevator.h>
#include <linux/bio.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/init.h>

struct sstf_data {
        struct list_head queue;
        sector_t last_sector;
};

static void sstf_merged_requests(struct request_queue *q, struct request *rq,
                                 struct request *next)
{
        list_del_init(&next->queuelist);
}

static int sstf_dispatch(struct request_queue *q, int force)
{
        struct sstf_data *nd = q->elevator->elevator_data;

        if (!list_empty(&nd->queue)) {
                struct request *rq;
                rq = list_entry(nd->queue.next, struct request, queuelist);
                nd->last_sector = rq->bio->bi_iter.bi_sector;
                list_del_init(&rq->queuelist);
                elv_dispatch_sort(q, rq);
                printk("~~~~ %lu\n", (unsigned long)rq->bio->bi_iter.bi_sector);
                return 1;
```

```
39          }
40          return 0;
41  }
42
43  static void sstf_add_request(struct request_queue *q, struct request *rq)
44  {
45          struct sstf_data *nd = q->elevator->elevator_data;
46          struct list_head *request_head;
47          struct request *request_item;
48
49          list_for_each(request_head, &nd->queue) {
50                  request_item = list_entry(request_head, struct request, queuelist);
51                  if (rq->bio->bi_iter.bi_sector <= abs(nd->last_sector - request_item->bi
52                          list_add_tail(&rq->queuelist, request_head);
53                          return;
54                  }
55          }
56
57          printk("~~~~ ------ \n");
58          list_add_tail(&rq->queuelist, &nd->queue);
59  }
60
61  static struct request *
62  sstf_former_request(struct request_queue *q, struct request *rq)
63  {
64          struct sstf_data *nd = q->elevator->elevator_data;
65
66          if (rq->queuelist.prev == &nd->queue)
67                  return NULL;
68          return list_entry(rq->queuelist.prev, struct request, queuelist);
69  }
70
71  static struct request *
72  sstf_latter_request(struct request_queue *q, struct request *rq)
```

```
73  {
74          struct sstf_data *nd = q->elevator->elevator_data;
75
76          if (rq->queuelist.next == &nd->queue)
77                  return NULL;
78          return list_entry(rq->queuelist.next, struct request, queuelist);
79  }
80
81  static int sstf_init_queue(struct request_queue *q, struct elevator_type *e)
82  {
83          struct sstf_data *nd;
84          struct elevator_queue *eq;
85
86          eq = elevator_alloc(q, e);
87          if (!eq)
88                  return -ENOMEM;
89
90          nd = kmalloc_node(sizeof(*nd), GFP_KERNEL, q->node);
91          if (!nd) {
92                  kobject_put(&eq->kobj);
93                  return -ENOMEM;
94          }
95          eq->elevator_data = nd;
96
97          INIT_LIST_HEAD(&nd->queue);
98
99          spin_lock_irq(q->queue_lock);
100         q->elevator = eq;
101         spin_unlock_irq(q->queue_lock);
102         return 0;
103 }
104
105 static void sstf_exit_queue(struct elevator_queue *e)
106 {
```

```
107         struct sstf_data *nd = e->elevator_data;
108
109         BUG_ON(!list_empty(&nd->queue));
110         kfree(nd);
111 }
112
113 static struct elevator_type elevator_sstf = {
114                 .ops = {
115                                 .elevator_merge_req_fn                = sstf_merged_requ
116                                 .elevator_dispatch_fn                 = sstf_dispatch,
117                                 .elevator_add_req_fn                  = sstf_add_request,
118                                 .elevator_former_req_fn                 = sstf_former_req
119                                 .elevator_latter_req_fn                 = sstf_latter_req
120                                 .elevator_init_fn               = sstf_init_queue,
121                                 .elevator_exit_fn               = sstf_exit_queue,
122                 },
123                 .elevator_name = "sstf",
124                 .elevator_owner = THIS_MODULE,
125 };
126
127 static int __init sstf_init(void)
128 {
129         return elv_register(&elevator_sstf);
130 }
131
132 static void __exit sstf_exit(void)
133 {
134         elv_unregister(&elevator_sstf);
135 }
136
137 module_init(sstf_init);
138 module_exit(sstf_exit);
139
140
```

```
141 MODULE_AUTHOR("Andy Chase");
142 MODULE_LICENSE("GPL");
143 MODULE_DESCRIPTION("sstf IO scheduler");
```