CS411: Individual Writeup - Group 13
Project 2: SSTF IO Scheduler

Trevor Bramwell

October 22, 2012

Introduction

The main point of this assignment was to gain an understanding of the Linux I/O Scheduler interface, and to teach us about scheduling algorithms.

Throughout this assignment I have learned that the Linux Kernel has four different I/O Scheduling algorithms. They are:

1. Noop
2. Deadline
3. Anticipitory
4. CFQ

Note that the 'Linus Elevator' is no longer in the kernel. Though Robert Love's book implies it is, he merely uses it as an example of a simple elevator algorithm for I/O scheduling.

The I/O scheduling algorithm my group implemented is the Shortest Seek Time First (SSTF) algorithm. This algorithm works to make the heads of a hard disk seek as little as possible. It accomplishes this by organizing I/O requests based on their sector positions, and adding requests closest to the disk head to the dispatch queue first.

Though SSTF increases the overall throughput of the I/O, it comes at the cost of possible process starvation. If a large enough number of I/O operations continues to take place in one area of the disk, requests in areas far away from there might never get dispatched, and thus starve.

First Steps

The default I/O scheduler is chosen either at compile time (using the configuration option *CONFIG_DEFAULT_IOSCHED*), or during boot with the *elevator=* option.

So our first step as a group was to get the I/O scheduler recognized and configured as the default. To start off, we copied *block/noop-iosched.c* to *block/sstf-ioched.c* and changed the function names and module parameters. Now that we had a scheduler in placed called SSTF, we added lines to *block/Kconfig.iosched* and *block/Makefile* in order to enable it when running *make config*.

Our group figured most of this out by using a previous CS411 class assignemnt description written by Rob Hess.

Design Decisions

When we got around to implementing the actual SSTF algorithm, me and Matt bounced ideas off of each other. My plan was to use a circularly linked list and keep a reference to the last request dispatched from it. Like someone holding their finger in a Rolodex. New requests are inserted in sorted order. Meaning the list should always be in order. When a request is to be dispatched, my algorithm would check the request on either side of the previous request, and find the one with the smallest difference in sector. This request would then be dispatched and the reference to the last dispatch request would be updated.

But, due to the complexity involved in my solution, our group decided it would be better to keep it simple and use Matt's solution. His solution was to add all requests to the end of the list.

```
/*
 * This will not need to be touched.
 * adding a request will just add it to the end of the list
 */
static void sstf_add_request(struct request_queue *q, struct request *rq)
{
        struct sstf_data *nd = q->elevator->elevator_data;

        list_add_tail(&rq->queuelist, &nd->queue);
        printk("[SSTF] add %u %llu\n",
                (rq->cmd_flags & REQ_WRITE),
                (unsigned long long)blk_rq_pos(rq));
}
```

When a request is to be dispatched, the last request is compared to all the requests in the list until the smallest difference in sectors is found. That request is then put on the dispatch queue.

```
/*
 * elv_dispatch_sort is the function that actually adds the request to
 * the dispatch queue. It is exported...so it needs to be edited?
 */
static int sstf_dispatch(struct request_queue *q, int force)
{
        struct sstf_data *nd = q->elevator->elevator_data;
        struct request *closest_rq;
        int abs, closest = -1;


        if (!list_empty(&nd->queue)) {
                struct request *rq;
                /*
                 * Go through the request queue and find the smallest difference
                 * between the last sector in the dispatch queue and the first
                 * sector of each request
                 */
                list_for_each_entry(rq, &nd->queue, queuelist) {

                        /*
                         * For each request, get the difference between the last
                         * sector in the dispatch queue, and the first sector of the current request
                        */
                        abs = find_last_sector(rq);
                        /*
                         * If the difference is smaller than the current smallest
                         * difference, update closest and closest_rq. If this is the first
                         * item in the list, update.
                         */
```

```
                        if( closest > abs || closest == -1 ) {
                                closest = abs;
                                closest_rq = rq;
                        }
                }

                last_sector = blk_rq_pos(closest_rq) +
                                blk_rq_sectors(closest_rq);

                list_del_init(&closest_rq->queuelist);
                elv_dispatch_add_tail(q, closest_rq);

                printk("[SSTF] dsp %u %llu\n",
                        (current_rq->cmd_flags & REQ_WRITE),
                        (unsigned long long)blk_rq_pos(closest_rq));
                return 1;
        }
        return 0;
}
```

Proof

After trying a couple different methods, we decided our *printk()* statements were reasonable proof, since they list the order in which requests are added and dispatched along with their sector numbers. We found some requests that got added, took longer to get dispatched then some other. This was because they accessed sectors far enough away from the current requests being handled.

One other solution we attempted was to write a program that had three reader processes and one writer process. But due to disk caching we weren't able to get it to work properly.

Learning

As a whole, this assignment went much better than before. I feel our group worked together much better than last time, and actually put in a great effort to complete the assignment. Both Matt and Brian worked hard to complete this assignment on time.