CS 411

Project 2: Implement SSTF I/O Scheduler October 30, 2013

Class MWF 9-9:50 AM

Drake Bridgewater

Contents

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

After many days of attempting to understand the kernel and why I could not compile the given kernel with the given patch I got an understanding of simple operation of the Linux environment. I know the intended main point was to understand how the scheduling works in general, but I got more out of this because of my troubles. I now understand how to access basic files within the system, how the diff function works, and what the big deal with git is and how useful it is.

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

Our team was having difficulty getting the kernel to compile correctly therefore our time to work on the actual assignment was limited and we had to implement our solution in the final three days. This led use to poor choices and we found a similar implementation online but after lecture on Oct. 25 I had a better understanding and when we looked at the online code we noticed some major errors and we decided that we should implement the entire thing by ourselves. We thought about doing some sudo code but we thought we had a decent enough understanding to implement it without. In the end we probably could have saved some time if we wrote some sudo code.

As for the algorithm we took our understanding of linked list from the data structures course and applied that along with the new knowledge we gained from I/O.

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

To ensure that our solution is correct we only had time to boot it.

4 What did you learn?

During this project I learned so many things that are barely related to I/O Scheduling, but I know have a much better understanding of Linux in general. I can get around fluidly without have to check cheat sheets every other minute.

Appendix 1: Source Code

```
/*
 * elevator sstf
 */
#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 head_pos;
        int direction, queue_count;
};

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 (q==NULL)
                        return 0;

        if (!list_empty(&nd->queue)) {
                struct request *rq, *nextrq, *prevrq;
                sector_t next, prev;

                nextrq = list_entry(nd->queue.next, struct request, queuelist);
                next = blk_rq_pos(nextrq);

                prevrq = list_entry(nd->queue.prev, struct request, queuelist);
                prev = blk_rq_pos(prevrq);

                if(nd->direction > 0 && next > prev) {
                        /* MOVE UP */
                        rq = nextrq;
                } else if(nd->direction < 0 && next < prev) {
                        /* UPPER BOUND */
                        rq = prevrq;
                        nd->direction = -1;
                } else if(nd->direction < 0 && next < prev) {
                        /* Move DOWN */
                        rq = prevrq;
                } else {
                        /* LOWER BOUND */
                        rq = nextrq;
                        nd->direction = 1;
                }
                /* remove request */
                list_del_init(&rq->queuelist);
                nd->queue_count--;
                nd->head_pos = blk_rq_pos(rq)  + blk_rq_sectors(rq) - 1;
                elv_dispatch_sort(q, rq);
                /* update queue head position
                sstf_balance(nd); */

                if(rq_data_dir(rq) == 0)
                        printk(KERN_INFO "[SSTF] dsp READ %ld\n",
                                        (long)blk_rq_sectors(rq));
                else
                        printk(KERN_INFO "[SSTF] dsp WRITE %ld\n",
                                        (long)blk_rq_sectors(rq));
                return 1;
        }
        return 0;
}

static void sstf_add_request(struct request_queue *q, struct request *rq)
{
        struct sstf_data *sd = q->elevator->elevator_data;
        struct list_head *position;
        /* BASE CASE */
        if(list_empty(&sd->queue))  {
                list_add(&rq->queuelist,&sd->queue);
                sd->queue_count++;
                return;
        }

        sector_t rq_sect, cur_sect, next_sect;
        rq_sect = blk_rq_pos(rq);

        list_for_each(position, &sd->queue) {

                if (sd->queue_count == 1) {
                        list_add(&rq->queuelist, position);
                        sd->queue_count++;
                        printk(KERN_INFO "[SSTF] added when 1 element long\n");
                        return;
                }

                struct request *cur_rq = list_entry(position,
                                struct request, queuelist);
                cur_sect = blk_rq_pos(cur_rq);
                struct request *next_rq = list_entry(position->next,
                                struct request, queuelist);
                next_sect = blk_rq_pos(next_rq);

                if (rq_sect >= cur_sect && rq_sect <= next_sect) {
                        list_add(&rq->queuelist, position);
                        sd->queue_count++;
                        printk(KERN_INFO "[SSTF] added in Sort\n");
                        return;
                }
        }
        /* now add at the correct position */
        list_add_tail(&rq->queuelist, &sd->queue);
        sd->queue_count++;
        printk(KERN_INFO "[SSTF] added to end\n");
}

static struct request *
sstf_former_request(struct request_queue *q, struct request *rq)
{
        struct sstf_data *nd = q->elevator->elevator_data;

        if (rq->queuelist.prev == &nd->queue)
                return NULL;
        return list_entry(rq->queuelist.prev, struct request, queuelist);
}

static struct request *
sstf_latter_request(struct request_queue *q, struct request *rq)
{
        struct sstf_data *nd = q->elevator->elevator_data;

        if (rq->queuelist.next == &nd->queue)
                return NULL;
        return list_entry(rq->queuelist.next, struct request, queuelist);
}

static void *sstf_init_queue(struct request_queue *q)
{
        struct sstf_data *nd;

        nd = kmalloc_node(sizeof(*nd), GFP_KERNEL, q->node);
        if (!nd)
                return NULL;
        INIT_LIST_HEAD(&nd->queue);
        return nd;
}

static void sstf_exit_queue(struct elevator_queue *e)
{
        struct sstf_data *nd = e->elevator_data;

        BUG_ON(!list_empty(&nd->queue));
        kfree(nd);
}

static struct elevator_type elevator_sstf = {
        .ops = {
                .elevator_merge_req_fn                = sstf_merged_requests,
                .elevator_dispatch_fn                 = sstf_dispatch,
                .elevator_add_req_fn                  = sstf_add_request,
                .elevator_former_req_fn     = sstf_former_request,
                .elevator_latter_req_fn     = sstf_latter_request,
                .elevator_init_fn           = sstf_init_queue,
                .elevator_exit_fn           = sstf_exit_queue,
        },
        .elevator_name = "sstf",
        .elevator_owner = THIS_MODULE,
};

static int __init sstf_init(void)
{
        elv_register(&elevator_sstf);

        return 0;
}

static void __exit sstf_exit(void)
{
        elv_unregister(&elevator_sstf);
}

module_init(sstf_init);
module_exit(sstf_exit);


MODULE_AUTHOR("CS411 - GROUP14");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("SSTF IO scheduler");
```