CS 444 Group 15

Project 2

Member:

Xiaomei WANG

Changxu YAN

Xilun GUO

supervised by

D. Kevin McGrath Brown

October 22, 2016

Command log for running the initial kernel

Version control log

Date	Person	Message
Sat Oct 08 15:33:33	Changxu	Added the solution for concurrency problem 1
Fri Oct 07 17:35:21	Xiaomei	version 2
Fri Oct 07 17:09:30	Xiaomei	Concurrency file
Sun Oct 02 03:27:11	Changxu	Cleaned up the dir
Sun Oct 02 03:16:26	Changxu	Updated README.md, some description files
Sat Oct 01 04:10:18	Changxu	Added tex file for generating git log graph
Sat Oct 01 02:19:02	Changxu	Concurrency file

Work Log

Date	Person	Task
Sat Oct 1	Changxu	pthreads
Sat Oct 1	Changxu	random number generator
Sun Oct 2	Xiaomei	pthreads
Wed Oct 5	Xilun	Finished pthreads
Fri Oct 7	Xiaoemi	Multiple threads
Fri Oct 7	Changxu	random number generator
Mon Oct 10	Xilun	Finished integrating
Mon Oct 10	Xiaomei	style guide
Mon Oct 10	Chnagxu	style guide

Appendix 1: Look elevator code

```
/*
 * elevator noop
 */
#include <linux/blkdev.h>
#include <linux/elevator.h>
#include <linux/bio.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/linkage.h>

struct noop_data {
        struct list_head queue;
  struct list_head prev;
//struct list_head first;
};

/*asmlinkage long sys_get_mem_usage(void){

  return 5;
  }*/


static int direction;

static void sstf_merged_requests(struct request_queue *q, struct request *rq,
                                 struct request *next)
{
  //printk("merge\n");
        list_del_init(&next->queuelist);
}

static int sstf_dispatch(struct request_queue *q, int force)
{
  //  printk("dispatch\n");
        struct noop_data *nd = q->elevator->elevator_data;
        struct list_head *curr2 = nd->queue.next;
        if (!list_empty(&nd->queue)) {
                struct request *rq;
                rq = list_entry(nd->queue.next, struct request, queuelist);
                int i =1;
                for(;blk_rq_pos(list_entry(curr2,struct request, queuelist)) != blk_rq_pos( list_ent
                printk("dispatch:  %llu\nqueue size:   %d\n", (unsigned long long)blk_rq_pos(rq),i);
                for(;blk_rq_pos(list_entry(curr2,struct request, queuelist)) != blk_rq_pos( list_ent

                //printk("dispatch1");
                list_del_init(&rq->queuelist);
                elv_dispatch_sort(q, rq);
```

```
                return 1;
        }
        return 0;
}

static void sstf_add_request(struct request_queue *q, struct request *rq)
{
        struct noop_data *nd = q->elevator->elevator_data;
        printk("add\n");

        if(list_empty(&nd->queue))  {
          list_add_tail(&rq->queuelist,&nd->queue);
          if (direction == 1) {
            direction = 2;
          }
          if (direction == 2){
            direction = 1;
          }
          return;
        }
        struct list_head *curr2 = &nd->queue;
        int i = 0;
        int headSector = blk_rq_pos(list_entry(nd->queue.prev, struct request, queuelist));
        int firstSector =blk_rq_pos(list_entry(&nd->queue, struct request, queuelist));
        int nextSector;
        int pos = blk_rq_pos(rq);
        for(;blk_rq_pos(list_entry(curr2,struct request, queuelist)) != blk_rq_pos( list_entry(&(nd-
          if (i == 1) {
            nextSector = blk_rq_pos(list_entry(curr2,struct request, queuelist));
          }
        }
        //i--;
        if (i == 1 && direction == 0){
          list_add_tail(&rq->queuelist,&nd->queue);
          if (headSector > pos) {
            direction = 1;
          }
          else {
            direction = 2;
          }
          return;
        }
        else {
          if (i > 1) {
            if ((firstSector < nextSector) && (direction == 1)){
              direction = 2;
            }
           if ((firstSector > nextSector) && (direction == 2)){
            direction = 1;
           }
          }
```

```
int i = 0;
struct request *rnext=list_entry(nd->queue.next, struct request, queuelist);
struct request *rprev = list_entry(nd->queue.prev, struct request, queuelist);
int next = blk_rq_pos(rnext);
int prev = blk_rq_pos(rprev);

int diff1;
int diff2;
if (direction == 1){
  diff1 = prev-pos;
  diff2 = prev-next;
  if ((diff1 > 0) && ((diff1 < diff2) || (diff2 < 0))) {
    list_add(&rq->queuelist, &nd->queue);
    return;
  }
}
if (direction == 2){
  diff1 = pos-prev;
  diff2 = next-prev;
  if ((diff1 > 0) && ((diff1 < diff2) || (diff2 < 0))) {
    list_add(&rq->queuelist, &nd->queue);
    return;
  }
}
struct list_head *curr1 = nd->queue.next;
struct list_head *curr = nd->queue.next;
struct list_head *last;
for(;blk_rq_pos(list_entry(curr,struct request, queuelist)) != blk_rq_pos( list_entry(&(nd
  //   if (i > 100) {
  //  list_add_tail(&rq->queuelist, &nd->queue);
  //  return;
  //}
  rnext = list_entry((*curr).next, struct request, queuelist);
  rprev = list_entry((*curr).prev, struct request, queuelist);
  next = blk_rq_pos(rnext);
  prev = blk_rq_pos(rprev);
  if (direction == 1){
    diff1 = prev-pos;
    diff2 = prev-next;
    if ((diff1 > 0) && ((diff1 < diff2) || (diff2 < 0))) {
      list_add(&rq->queuelist, curr);
      return;
    }
  }
  if (direction == 2){
    diff1 = pos-prev;
    diff2 = next-prev;
    if ((diff1 > 0) && ((diff1 < diff2) || (diff2 < 0))) {
      list_add(&rq->queuelist, curr);
      return;
    }
```

```
            }
          }

  i = 0;
  curr = last;
  for(;blk_rq_pos(list_entry(curr,struct request, queuelist)) != blk_rq_pos( list_entry(&(nd->queue)
    //if (i > 100) {
    //  list_add_tail(&rq->queuelist, &nd->queue);
    //  return;
    // }
    rnext = list_entry((*curr).next, struct request, queuelist);
    rprev = list_entry((*curr).prev, struct request, queuelist);
    next = blk_rq_pos(rnext);
    prev = blk_rq_pos(rprev);
    if (direction == 1){
      diff1 = pos-prev;
      diff2 = next-prev;
      if ((diff1 > 0) && ((diff1 < diff2) || (diff2 < 0))) {
        list_add(&rq->queuelist, curr);
        return;
      }
    }
    if (direction == 2){
      diff1 = prev-pos;
      diff2 = next-pos;
      if ((diff1 > 0) && ((diff1 < diff2) || (diff2 < 0))) {
        list_add(&rq->queuelist, curr);
        return;
      }
    }
  }
        }
list_add_tail(&rq->queuelist, &nd->queue);
}
//return;
        //    int next = rnext->sector;
        //int prev = rprev->sector;
        //int pos = rq->sector;

        //printk("tail");
        //list_add_tail(&rq->queuelist, &nd->queue);


static struct request *
sstf_former_request(struct request_queue *q, struct request *rq)
{
  //printk("former\n");
        struct noop_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)
{
  //printk("latter\n");
        struct noop_data *nd = q->elevator->elevator_data;

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

static int sstf_init_queue(struct request_queue *q, struct elevator_type *e)
{
        struct noop_data *nd;
        struct elevator_queue *eq;
        //        printk("init\n");
        eq = elevator_alloc(q, e);
        if (!eq)
                return -ENOMEM;

        nd = kmalloc_node(sizeof(*nd), GFP_KERNEL, q->node);
        if (!nd) {
                kobject_put(&eq->kobj);
                return -ENOMEM;
        }
        eq->elevator_data = nd;

        INIT_LIST_HEAD(&nd->queue);

        spin_lock_irq(q->queue_lock);
        q->elevator = eq;
        spin_unlock_irq(q->queue_lock);
        return 0;
}

static void sstf_exit_queue(struct elevator_queue *e)
{
  //printk("exit\n");
        struct noop_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)
{
        return elv_register(&elevator_sstf);
}

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

module_init(sstf_init);
module_exit(sstf_exit);


MODULE_AUTHOR("Jens Axboe");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("SSTF IO scheduler");
```