Block device driver

Register a block device driver in the kernel

First, declare a constant for the size of our device:

```
#define MYBLK_SIZE_SECT (4 * 1024 * 2)
```

Then, declare a few global variables to store the major number that we will allocate, and the struct gendisk pointer:

```
static struct gendisk *disk;
static unsigned char *data;
```

Create an empty request() operation:

```
static void myblk_request(struct request_queue *q)
{
}
```

Declare a block_device_operations structure with no operation, specifying only the block device owner:

```
static struct block_device_operations myblk_ops = {
        .owner = THIS_MODULE,
};
```

In the module initialization function, register the major number, allocate and initialize the gendisk structure, create the queue and add the disk to the system:

```
major = register_blkdev(0, "myblk");
if (major < 0)
{
      printk(KERN_ERR "Couldn't register major, %d\n", major);
      return major;
}

disk = alloc_disk(1);
if (! disk)
{
      printk(KERN_ERR "Couldn't get a gendisk structure\n");
      unregister_blkdev(major, "myblk");
      return -ENOMEM;
}

disk->major = major;
disk->first_minor = 0;
disk->minors = 1;
disk->fops = & myblk_ops;
strncpy(disk->disk_name, "testblk", sizeof(disk->disk_name));
set_capacity(disk, MYBLK_SIZE_SECT);

disk->queue = blk_init_queue(myblk_request, NULL);
if (! disk->queue)
{
      printk(KERN_ERR "Couldn't allocate a queue\n");
      put_disk(disk);
      unregister_blkdev(major, "myblk");
```

```
      return -ENOMEM;
}

add_disk(disk);
```

In the module cleanup function, the necessary cleanup is done:

```
del_gendisk(disk);
blk_cleanup_queue(disk->queue);
put_disk(disk);
unregister_blkdev(major, "myblk");
```

Handle I/O requests

To handle the I/O requests, it will be nice to have the definition of a constant for the size of a sector in the kernel:

```
#define KERNEL_SECTOR_SIZE 512
```

We also define a global pointer that will contain the address of the memory area used to store the contents of the ramdisk:

```
static unsigned char *data;
```

In the module initialization function, we allocate the area of memory. This must be done before the disk is added to the system using add_disk(), because as soon as add_disk() is called, I/O requests might be made on the block device:

```
data = vmalloc(MYBLK_SIZE_SECT * KERNEL_SECTOR_SIZE);
if (! data)
{
      printk(KERN_ERR "Couldn't allocate memory for the device");
      return -ENOMEM;
}
```

Of course, the error handling of the other initialization steps must be fixed accordingly.

In the module cleanup function, we don't forget to free this memory, after the disk has been removed from the system:

```
vfree(data);
```

Now, we only have the `request()` operation to implement, in the simplest possible way. We loop over all requests using `elv_next_request()` and for each of them, we make the memory copy in the right direction depending on the request type (when `rq_data_dir()` is `TRUE`, it means that the request is a write request), and notify the completion of the request using `__blk_end_request()`.

```
static void myblk_request(struct request_queue *q)
{
        struct request *rq;

        while ((rq = elv_next_request(q)) != NULL)
        {
                if (rq_data_dir(rq))
                        memcpy(data + rq->sector * KERNEL_SECTOR_SIZE, rq->buffer,
                               rq->current_nr_sectors * KERNEL_SECTOR_SIZE);
                else
                        memcpy(rq->buffer, data + rq->sector * KERNEL_SECTOR_SIZE,
```

```
                              rq->current_nr_sectors * KERNEL_SECTOR_SIZE);

                __blk_end_request(rq, 0, rq->current_nr_sectors << 9);
        }
}
```

Asynchronous operation

We declare a linked list and a spinlock protecting this list against concurrent accesses:

```
static LIST_HEAD(req_list);
static DEFINE_SPINLOCK(req_list_lock);
```

We declare a timer that will trigger the execution of a function every second:

```
static struct timer_list req_timer;
```

In the initialization function, we initialize the timer and register it in the kernel:

```
        init_timer(& req_timer);
        req_timer.function = myblk_timer_func;
        req_timer.expires  = jiffies + HZ;
        add_timer(& req_timer);
```

In the cleanup function, we unregister the timer and make sure that it is not running anymore:

```
        del_timer_sync(& req_timer);
```

We modify the `request()` operation so that requests are simply dequeued from the request queue and added to our own linked list. The field `queuelist` of the request structure can be used for our own purposes once the request is dequeued from the request queue. So, we use this `queuelist` field to link each request in our linked list:

```
static void myblk_request(struct request_queue *q)
{
        struct request *rq;

        while ((rq = elv_next_request(q)) != NULL)
        {
                blkdev_dequeue_request(rq);

                spin_lock(& req_list_lock);
                list_add_tail(& rq->queuelist, & req_list);
                spin_unlock(& req_list_lock);
        }
}
```

Finally, we implement the function executed every second by the timer. This function loops over the list of requests until it is empty. For each request, it loops over each segment and make the necessary memory copies, and notifies the completion of the request. Finally, it rearms the timer so that the function gets called again at the next second:

```
static void myblk_timer_func(unsigned long d)
{
        struct request *rq;

        while (! list_empty(& req_list))
        {
                struct bio_vec *bvec;
```

```
                struct req_iterator iter;

                spin_lock(& req_list_lock);
                rq = list_entry(req_list.next, struct request, queuelist);
                list_del_init(&rq->queuelist);
                spin_unlock(& req_list_lock);

                rq_for_each_segment(bvec, rq, iter)
                {
                        if (rq_data_dir(rq))
                                memcpy(data + rq->sector * KERNEL_SECTOR_SIZE,
                                       page_address(bvec->bv_page) + bvec->bv_offset,
                                       bvec->bv_len);
                        else
                                memcpy(page_address(bvec->bv_page) + bvec->bv_offset,
                                       data + rq->sector * KERNEL_SECTOR_SIZE,
                                       bvec->bv_len);

                        rq->sector += bvec->bv_len / KERNEL_SECTOR_SIZE;
                }

                blk_end_request(rq, 0, rq->nr_sectors << 9);
        }

        req_timer.expires = jiffies + HZ;
        add_timer(& req_timer);
}
```

Complete source code

A complete solution can be found on
http://free-electrons.com/labs/solutions/linux/block/pyedur/myblk-v2.c