BAN CƠ YẾU CHÍNH PHỦ

HỌC VIỆN KỸ THUẬT MẬT MÃ

LẬP TRÌNH DRIVER CHO THIẾT BỊ DẠNG KHỐI

Ngành: Công nghệ thông tin

Chuyên ngành: Kỹ thuật phần mềm nhúng và di động

Mã số: 52.48.02.01

Nội dung

- Kiến thức cơ sở
- Đăng ký
- Các thao tác trên thiết bị
- Xử lý yêu cầu
- Phân tích ví dụ minh hoạ

Kiến thức cơ sở

- Trình điều khiển của thiết bị khối cho phép truy cập đến thiết bị ngẫu nhiên và truyền theo các khối ví dụ như trình điều khiển ổ cứng
- **ramdisk** (vùng dữ liệu trên thiết bị nhớ ngoài nhưng được xử lý như RAM)
- nhân cũng đã chứa các cài đặt ramdisk cơ sở để giảm thiểu độ phức tạp nên trình điều khiển mới thường gọi là **sbull** (Simple Block Utility for Loading Localities)

Kiến thức cơ sở

- Một block là một khối dữ liệu có kích thước xác định bởi nhân hệ điều hành, thường dùng là 4096 byte
  - Kích thước này thường phụ thuộc vào hệ điều hành và kiến trúc sử dụng.
- Trái lại, một sector là một khối nhỏ có kích thước xác định bởi phần cứng
  - Hệ điều hành thường cung cấp và xử lý các sector lô-gic với kích thước 512 byte

Kiến thức cơ sở

- Nếu thiết bị phần cứng sử dụng sector có kích thước vật lý khác, hệ điều hành phải xử lý, điều chỉnh để tương thích, tránh các yêu cầu mà thiết bị vào/ra không thực hiện được
  - Khi đó, nhân sẽ cung cấp **một word là một nhóm các sector 512 byte**
- Do đó, với các hệ thống phần cứng có kích thước sector khác nhau, chúng ta cần điều chỉnh số sector trong một word của nhân cho phù hợp
- Việc này được thực hiện trong trình điều khiển sbull

Đăng ký

- Đăng ký thiết bị khối
- Đăng ký ổ đĩa
- Khởi tạo trong sbull

Đăng ký thiết bị khối

- Bước đầu tiên cho mọi trình điều khiển khối là đăng ký bản thân thiết bị với nhân. Hàm thực hiện tác vụ này là register_blkdev (được khai báo trong <linux/fs.h>):

```
int register_blkdev(unsigned int major, const char *name);
```

  - Tham số major là số hiệu cấp cho thiết bị khi sử dụng, được kết hợp với tên (chứa trong /proc/devices). Nếu major bằng 0, nhân sẽ cấp phát một số major mới và trả về nơi gọi. Một giá trị âm trả về từ register_blkdev cho biết đã có lỗi xảy ra.

- Tương tự, hàm hủy đăng ký có nguyên mẫu như sau:

```
int unregister_blkdev(unsigned int major, const char *name);
```

Đăng ký ổ đĩa

- **register_blkdev** được gọi để lấy số hiệu sử dụng nhưng nó không làm các trình điều khiển sẵn sàng với hệ thống
- Do đó, một giao diện đăng ký tách biệt cần phải sử dụng để quản lý các trình điều khiển cụ thể
- Sử dụng giao diện này sẽ yêu cầu một cặp cấu trúc mới cho biết nơi bắt đầu làm việc với thiết bị

Cấu trúc gendisk

- Cấu trúc gendisk (được khai báo trong <linux/genhd.h>) là một biểu diễn trong nhân của một thiết bị cụ thể. Các trường trong cấu trúc gendisk dưới đây phải được khởi tạo bởi trình điều khiển khối.
  - int major;
  - int first_minor; //chỉ số minor đầu
  - int minors; //số minor
    - ✓ Mô tả số hiệu thiết bị sử dụng bởi ổ đĩa.
    - ✓ Tối thiểu, mỗi ổ đĩa phải có một số minor.
    - ✓ Nếu ổ đĩa được phân chia, cần cấp phát số minor cho mỗi phân vùng.
    - ✓ Giá trị phổ biến của minors là 16 cho phép một thiết bị đầy đủ và 15 phân vùng

Cấu trúc gendisk

- char disk_name [32];
  - Là tên của thiết bị ổ đĩa; được chỉ ra trong `/proc/partitions` và `sysfs`.
- `struct block_device_operations *fops;`
  - Tập các thao tác thiết bị.
- struct request_queue *queue;
  - Cấu trúc này được sử dụng bởi nhân để quản lý các yêu cầu vào/ra cho thiết bị.

Cấu trúc gendisk

- int flags;
  - Một tập các cờ mô tả trạng thái thiết bị.
  - Nếu thiết bị đã được gỡ ra, cờ GENHD_FL_REMOVABLE được thiết lập; với ổ đĩa CD-ROM thì cờ GENHD_FL_CD được thiết lập.
  - Nếu không muốn thông tin phân vùng được hiển thị trong /proc/partitions thì thiết lập cờ GENHD_FL_SUPPRESS_PARTITION_INFO.
- sector_t capacity;
  - Dung lượng của ổ đĩa, tính theo số sector, mỗi sector 512 byte.
- void *private_data;
  - Con trỏ, trỏ đến vùng dữ liệu riêng của trình điều khiển khối

Cấu trúc gendisk

- Nhân cung cấp một tập nhỏ các hàm làm việc với cấu trúc gendisk.
- Cấu trúc gendisk là một cấu trúc được cấp phát động và yêu cầu các xử lý chuyên biệt trong nhân để khởi tạo nó. Các trình điều khiển không thể tự cấp phát cấu trúc này mà phải gọi hàm sau:
  - struct gendisk *alloc_disk(int minors);
    - ✓ Tham số minors cho biết số minor – tương ứng với phân vùng mà ổ đĩa sử dụng. Giá trị này không thay đổi được sau khi hàm thực thi.

Cấu trúc gendisk

- Khi không sử dụng ổ đĩa, có thể giải phóng bằng hàm:
  - void del_gendisk(struct gendisk *gd);
- *Sau khi cấp phát cấu trúc gendisk, muốn nó sẵn sàng với hệ thống, cần phải khởi tạo cho cấu trúc và gọi hàm add_disk*:

```
void add_disk(struct gendisk *gd);
```

  - Chú ý, ngay sau khi gọi hàm add_disk, ổ đĩa đã hoạt động và các phương thức của nó có thể được gọi bất cứ khi nào.
    - ✓Do đó, không nên gọi add_disk khi trình điều khiển chưa hoàn thành việc khởi tạo

Khởi tạo trong sbull

- Trình điều khiển sbull cài đặt một tập các ổ đĩa ảo.
  - Với mỗi ổ đĩa, sbull cấp phát (sử dụng vmalloc) một mảng nhớ và hiệu lực hóa mảng nhớ này thông qua các thao tác trên khối.
  - sbull có thể được kiểm tra bằng việc phân vùng các thiết bị ảo, xây dựng hệ thống file trên nó và gắn vào hệ thống
- Sbull cho phép chỉ ra số hiệu sử dụng (major) ở thời điểm biên dịch hoặc nạp mô-đun. Nếu không được chỉ ra, số hiệu được cấp phát động bằng register_blkdev:

```
sbull_major = register_blkdev(sbull_major, "sbull");
if (sbull_major <= 0) {
  printk(KERN_WARNING "sbull: unable to get major number\n"); return
  -EBUSY;
}
```

Khởi tạo trong sbull

- Giống như các thiết bị ảo, thiết bị sbull được mô tả bằng một cấu trúc nội:

```
struct sbull_dev {
    int size;   /* Device size in sectors */
    u8 *data;   /* The data array */
    short users;         /* How many users */
    short media_change; /* Flag a media change? */
    spinlock_t lock;    /* For mutual exclusion */
    struct request_queue *queue;         /* The device request queue */
    struct gendisk *gd; /* The gendisk structure */
    struct timer_list timer;    /* For simulated media changes */
} dev;
```

Khởi tạo trong sbull

- Các bước cụ thể sau được yêu cầu để khởi tạo cấu trúc trên và làm cho thiết bị kết hợp với nó sẵn sàng thực hiện trong hệ thống.
- Việc khởi tạo và cấp phát bộ nhớ như sau:

```
memset (dev, 0, sizeof (struct sbull_dev));
dev->size = nsectors*hardsect_size;
dev->data = vmalloc(dev->size);
if (dev->data == NULL) {
  printk (KERN_NOTICE "vmalloc failure.\n");
  return;
}
spin_lock_init(&dev->lock);
```

Khởi tạo trong sbull

- Tiếp theo cần cấp phát và khởi tạo một cấu trúc spinlock trước khi cấp phát hàng đợi yêu cầu.
  - dev->queue = blk_init_queue(sbull_request, &dev->lock);
  - Trong đó, sbull_request là hàm yêu cầu – sử dụng để thực thi các yêu cầu đọc và ghi khối.
  - Khi cấp phát hàng đợi yêu cầu, cần cung cấp một spinlock để điều khiển việc truy xuất hàng đợi này.

Khởi tạo trong sbull

- Sau khi hàng đợi yêu cầu và bộ nhớ thiết bị đã có, chúng ta có thể cấp phát, khởi tạo và cài đặt cấu trúc gendisk tương ứng thông qua đoạn mã dưới đây.

```
dev->gd = alloc_disk(SBULL_MINORS);
if (! dev->gd) {
    printk (KERN_NOTICE "alloc_disk failure\n");
    goto out_vfree;
}
dev->gd->major = sbull_major;
dev->gd->first_minor = which*SBULL_MINORS;
dev->gd->fops = &sbull_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
snprintf (dev->gd->disk_name, 32, "sbull%c", which + 'a');
set_capacity(dev->gd,                nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
```

  - Trong đó, SBULL_MINORS là số lượng minor mà mỗi thiết bị sbull hỗ trợ

Chú ý về kích thước sector

- Như đề cập trước, nhân Linux thường coi mọi ổ đĩa là một mảng tuyến tính các sector 512 byte
- Do đó, thách thức lớn đặt ra khi kích thước của sector phần cứng khác 512 byte.
- Để giải quyết vấn đề này, thiết bị sbull xuất ra tham số sử dụng để thay đổi kích thước sector "hardware"
- Thực chất đây là việc ánh xạ tập sector phần cứng trong yêu cầu thành tập sector lô-gic (có kích thước bằng kích thước sector của nhân).

Chú ý về kích thước sector

- Kỹ thuật co giãn này được thực hiện thông qua 2 hàm dưới đây.
  - hardsect_size là kích thước của sector phần cứng trong yêu cầu và dung lượng thiết bị sbull được thiết lập trong cấu trúc gendisk theo hàm thứ hai, với hằng số sử dụng để co giãn sector nhân với sector phần cứng với kích thước tùy ý.
  - blk_queue_hardsect_size(dev->queue, hardsect_size);
  - set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));

Các thao tác trên thiết bị

- Các phương thức mở và giải phóng
- Hỗ trợ các phương tiện di động
- Phương thức ioctl

Các thao tác trên thiết bị khối

- các thiết bị khối cũng sử dụng cấu trúc **block_device_operations** để cài đặt, cung cấp các thao tác sẵn sàng làm việc với hệ thống.
  - Cấu trúc này được khai báo trong <linux/fs.h>. Sau đây là các trường chính trong cấu trúc này:

```
int (*open)(struct inode *inode, struct file *filp);
int (*release)(struct inode *inode, struct file *filp);
```

    - ✓Tương tự như trình điều khiển ký tự, các hàm này được gọi khi thiết bị được mở và đóng.

Các phương thức mở và giải phóng

- Phương thức open cũng tương tự như trong trình điều khiển thiết bị ký tự; nó lấy các con trỏ kiểu cấu trúc inode và file hợp lệ
- Khi inode trỏ tới một thiết bị khối, trường i_bdev->bd_disk chứa một con trỏ đến cấu trúc gendisk tương ứng; con trỏ này có thể được sử dụng để lấy các cấu trúc dữ liệu nội của trình điều khiển cho thiết bị

Các phương thức mở và giải phóng

- minh họa một phương thức open cơ bản

```
static int sbull_open(struct inode *inode, struct file *filp)
{
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;
    del_timer_sync(&dev->timer);
    filp->private_data = dev;
    spin_lock(&dev->lock);
    if (! dev->users)
        check_disk_change(inode->i_bdev);
    dev->users++;
    spin_unlock(&dev->lock);
    return 0;
}
```

Các phương thức mở và giải phóng

- phương thức release; phương thức này sẽ giảm bộ đếm người dùng.

```
static int sbull_release(struct inode *inode, struct file *filp){
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;
    spin_lock(&dev->lock);
    dev->users--;
    if (!dev->users) {
        dev->timer.expires = jiffies + INVALIDATE_DELAY;
        add_timer(&dev->timer);
    }
    spin_unlock(&dev->lock);
    return 0;
}
```

Các thao tác trên thiết bị khối

- `int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);`
  - Phương thức này cài đặt lời gọi hệ thống ioctl, cho phép bắt các yêu cầu chuẩn; hầu hết các phương thức ioctl trong trình điều khiển khối được cài đặt ngắn gọn.
- `int (*media_changed) (struct gendisk *gd);`
  - Phương thức này được gọi khi nhân kiểm tra xem người dùng có thay đổi phương tiện trong trình điều khiển; trả về giá trị khác 0 nếu có. Tham số **cấu trúc gendisk** cho biết cách thức nhân biểu diễn một ổ đĩa đơn.
- `int (*revalidate_disk) (struct gendisk *gd);`
  - Phương thức revalidate_disk được gọi để xử lý sự kiện thay đổi thiết bị đa phương tiện; nó báo cho trình điều khiển biết để thực hiện công việc đảm bảo thiết bị mới sẵn sàng sử dụng

Hỗ trợ các phương tiện di động

- Cấu trúc block_device_operations bao gồm hai phương thức để hỗ trợ phương tiện lưu động. Nếu đang viết trình điều khiển cho một thiết bị không thể điều khiển được, ta có thể bỏ qua các phương pháp này một cách an toàn.
- Phương thức media_changed được gọi (từ check_disk_change) để xem liệu phương tiện đã được thay đổi hay chưa; nó sẽ trả về một giá trị khác không nếu chưa thay đổi. Việc cài đặt sbull rất đơn giản; nó truy vấn một cờ đã được thiết lập nếu bộ đếm thời gian loại bỏ phương tiện đã hết hạn:

```
int sbull_media_changed(struct gendisk *gd){
    struct sbull_dev *dev = gd->private_data;
    return dev->media_change;
}
```

Hỗ trợ các phương tiện di động

- Phương thức revalidate được gọi sau khi thay đổi phương tiện; công việc của nó là làm bất cứ điều gì được yêu cầu để chuẩn bị trình điều khiển cho các hoạt động trên phương tiện mới, nếu có.
- Sau khi gọi revalidate, nhân sẽ đọc bảng phân vùng và có thể bắt đầu làm việc với thiết bị. Việc cài đặt sbull chỉ đơn giản là đặt lại cờ media_change và xóa bộ nhớ thiết bị để mô phỏng việc chèn một đĩa trống.

```
int sbull_revalidate(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;
    if (dev->media_change) {
        dev->media_change = 0;
        memset (dev->data, 0, dev->size);
    }
    return 0;
}
```

Phương thức ioctl

- Các thiết bị khối có thể cung cấp phương thức ioctl để thực hiện các chức năng điều khiển thiết bị. Tuy nhiên, mã hệ thống con mức cao hơn chặn một số lệnh ioctl trước khi trình điều khiển thấy chúng. Trong thực tế, một trình điều khiển khối hiện đại có thể không phải cài đặt nhiều lệnh ioctl.
- Phương thức sbull ioctl thường chỉ xử lý một lệnh như một yêu cầu đến các thành phần trong thiết bị.

Phương thức ioctl

```
int sbull_ioctl (struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg){
    long size;
    struct hd_geometry geo;
    struct sbull_dev *dev = filp->private_data;
    switch(cmd) {
        case HDIO_GETGEO:
        size = dev->size*(hardsect_size/KERNEL_sectOR_SIZE);
        geo.cylinders = (size & ~0x3f) >> 6;
        geo.heads = 4;
        geo.sectors = 16;
        geo.start = 4;
        if (copy_to_user((void _ _user *) arg, &geo, sizeof(geo)))
            return -EFAULT;
        return 0;
    }
    return -ENOTTY; /* unknown command */
}
```

Xử lý yêu cầu

- Cốt lõi của mỗi trình điều khiển khối là phương thức *request*
  - Phương thức này này là nơi công việc thực sự được thực hiện hoặc ít nhất là bắt đầu; tất cả phần còn lại là trên nền hàm này
  - Hiệu suất của trình điều khiển đĩa có thể là một phần quan trọng trong toàn bộ hiệu năng của hệ thống.
- Do đó, hệ thống con khối của nhân đã được viết hướng đến hiệu năng; có thể để cho phép trình điều khiển tận dụng tối đa các thiết bị mà nó điều khiển

Phương thức xử lý yêu cầu

- Nguyên mẫu của phương thức (hàm) xử lý yêu cầu trong trình điều khiển thiết bị khối như sau:

```
void request(request_queue_t *queue);
```

- Hàm này được gọi bất cứ khi nào nhân yêu cầu trình điều khiển xử lý các thao tác đọc, ghi và các thao tác khác trên thiết bị.
- Hàm này không cần phải hoàn thành tất cả các yêu cầu trên hàng đợi trước khi trả về; nó có thể không hoàn thành bất kỳ một trong số các yêu cầu cho thiết bị thực.
- Tuy nhiên, nó phải bắt đầu với những yêu cầu đó và đảm bảo rằng tất cả các yêu cầu được xử lý bởi trình điều khiển.

Phương thức xử lý yêu cầu

- Mỗi thiết bị có một hàng đợi yêu cầu. Điều này là do việc chuyển dữ liệu thực tế đến từ một đĩa có thể diễn ra cách xa thời điểm nhân yêu cầu. sbull thực hiện hàng đợi như sau:

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

- Do đó, khi hàng đợi yêu cầu được tạp, hàm request sẽ kết hợp với nó.
- Chúng ta cũng cần cung cấp biến spinlock (khóa vòng) như một phần của tiến trình tạo hàng đợi.
- Bất cứ khi nào hàm request được gọi, biến này sẽ được duy trì bởi nhân.

Hàng đợi yêu cầu

Các hàm xử lý hàng đợi

- Chỉ có một tập nhỏ các hàm xử lý các yêu cầu trên hàng đợi mà trình điều khiển phải xét đến. Chúng ta cần phải duy trì hàng đợi trước khi gọi các hàm này. Hàm trả về yêu cầu tiếp theo sẽ xử lý là hàm elv_next_request:

```
struct request *elv_next_request(request_queue_t *queue);
```

- Để loại bỏ một yêu cầu ra khỏi hàng đợi, ta gọi hàm blkdev_dequeue_request:

```
void blkdev_dequeue_request(struct request *req);
```

- Nếu trình điều khiển muốn thao tác trên nhiều yêu cầu trong cùng một hàng đợi, cần phải không xếp hàng cho các yêu cầu này (dequeue). Các yêu cầu không xếp hàng có thể được tái xếp hàng trong hàng đợi. Để không xếp hàng một nhóm các yêu cầu, ta gọi hàm sau:

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

Hàng đợi yêu cầu

- **Các hàm điều khiển hàng đợi**
- Tầng khối xuất ra một tập các hàm được sử dụng bởi trình điều khiển để điều khiển các thao tác trên hàng đợi yêu cầu. Các hàm này bao gồm:

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

- Khi thiết bị gặp trạng thái không thể xử lý được một số lệnh, có thể gọi hàm blk_stop_queue để báo cho tầng khối (block layer).
  - Sau khi gọi hàm này, hàm request sẽ không thể được gọi cho đến khi ta gọi hàm blk_start_queue.

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

- Hàm này báo cho nhân biết địa chỉ vật lý cao nhất mà thiết bị có thể thực thi theo DMA.

Phân tích một yêu cầu

- Mỗi cấu trúc request đại diện cho một yêu cầu vào/ra theo khối mặc dù nó có thể được hình thành thông qua việc sáp nhập một số yêu cầu độc lập ở mức cao hơn.
- Các sector được chuyển cho bất kỳ yêu cầu cụ thể nào có thể được phân phối trên toàn bộ nhớ chính mặc dù chúng luôn tương ứng với một tập hợp sector liên tiếp trên thiết bị khối.
- Yêu cầu được biểu diễn dưới dạng một tập hợp các phân đoạn, mỗi phân đoạn tương ứng với một bộ đệm trong bộ nhớ.
- Nhân có thể tham gia nhiều yêu cầu liên quan đến các sector lân cận trên đĩa nhưng nó không bao giờ kết hợp các thao tác đọc và ghi trong một cấu trúc request

Cấu trúc bio

- Khi nhân, ở dạng hệ thống tệp tin, hệ thống con bộ nhớ ảo hoặc cuộc gọi hệ thống, quyết định một tập hợp các khối được chuyển đến/đi từ một thiết bị vào/ra khối, nó kết hợp một cấu trúc bio để mô tả hoạt động đó
  - Cấu trúc đó được xử lý với đoạn mã vào/ra khối, hợp nhất nó thành một cấu trúc request hiện có, nếu cần, sẽ tạo ra một cấu trúc mới
  - Cấu trúc bio chứa mọi thứ mà trình điều khiển khối cần để thực thi yêu cầu mà không cần tham chiếu đến tiến trình trong không gian người dùng

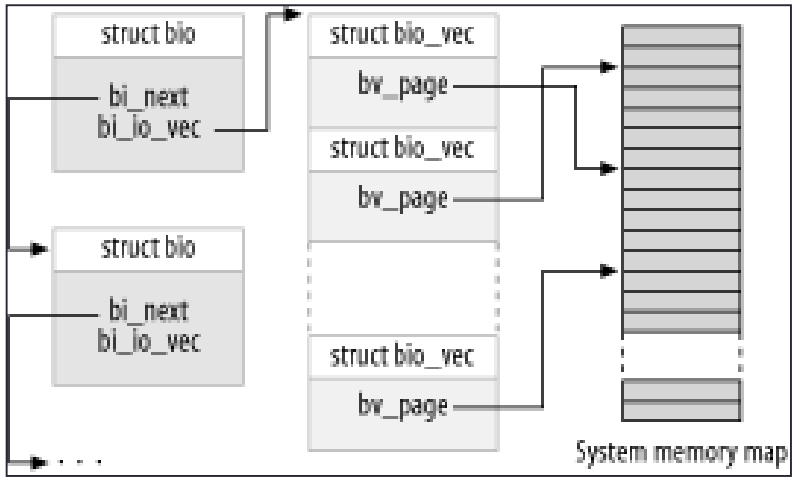
Cấu trúc bio

- Cấu trúc bio, được định nghĩa trong <linux/bio.h>, chứa một số trường có thể được sử dụng cho các tác giả trình điều khiển:
  - sector_t bi_sector;
    - ✓ Sector 512 byte đầu tiên được chuyển cho cấu trúc bio.
  - unsigned int bi_size;
    - ✓ Kích thước dữ liệu được chuyển, tính theo byte.
  - unsigned long bi_flags;
    - ✓ Tập các cờ mô tả bio; bit thấp nhất được thiết lập nếu là một yêu cầu ghi. macro bio_data_dir(bio) nên được sử dụng thay cho giám sát trên các cờ này.
  - unsigned short bio_phys_segments;
  - unsigned short bio_hw_segments;
    - ✓ Số lượng các đoạn vật lý chưa cấu trúc bio; và số lượng đoạn phần cứng sau khi ánh xạ DMA được thiết lập

Cấu trúc bio

- Lõi của cấu trúc bio, còn được gọi là bi_io_vec được xây dựng theo cấu dưới

```
struct bio_vec {
        struct page     *bv_page;
        unsigned int    bv_len;
        unsigned int    bv_offset;
};
```



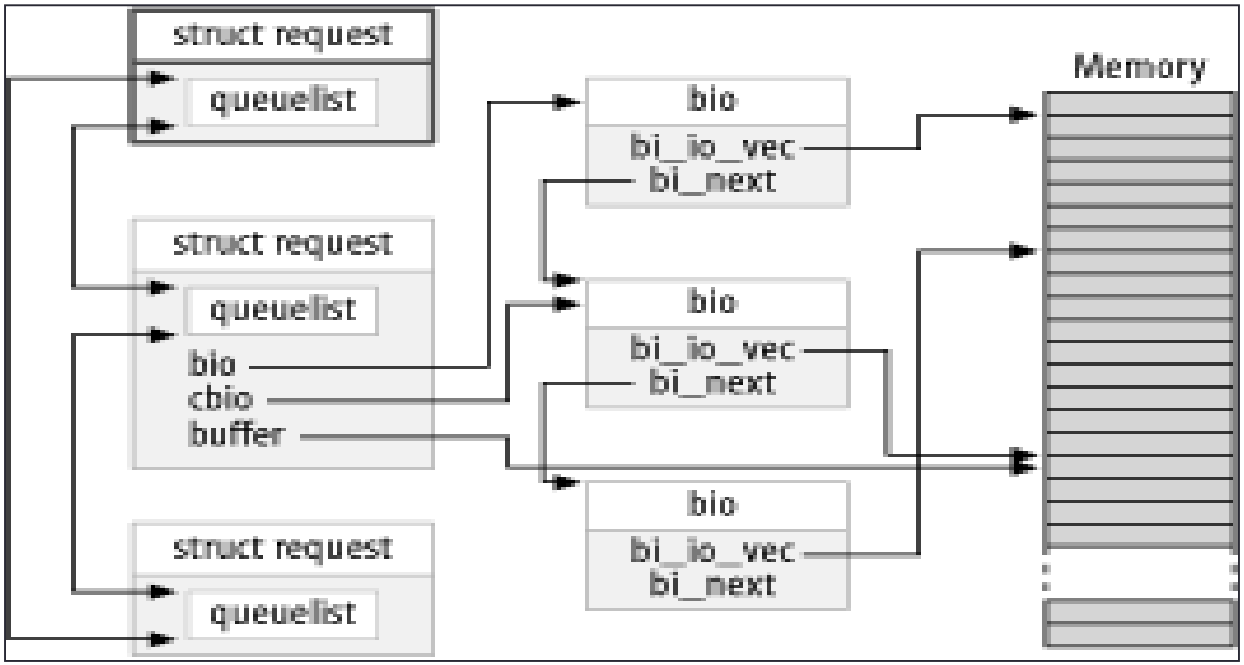
Các trường trong cấu trúc request

- `sector_t hard_sector;`
- `unsigned long hard_nr_sectors;`
- `unsigned int hard_cur_sectors;`
  - Các trường sử dụng để theo dõi các sectors mà trình điều khiển đã hoàn thành thao tác trên đó chưa.
    - ✓ Sector đầu tiên không được chuyển chứa trong hard_sector, tổng số sector chưa chuyền chứa trong hard_nr_sectors và số sector còn lại trong cấu trúc bio hiện tại là hard_cur_sectors
    - ✓ Các trường này chỉ được sử dụng bên trong hệ thống con khối, trình điều khiển không thao tác trên nó.
- struct bio *bio;
  - bio là một danh sách liên kết các cấu trúc bio cho yêu cầu hiện tại. Ta không thể truy cập trực tiếp trường này mà phải thông qua hàm use rq_for_each_bio.

Các trường trong cấu trúc request

- char *buffer;
  - Con trỏ đến vùng đệm sử dụng để chuyển dữ liệu.
- unsigned short nr_phys_segments;
  - Số các đoạn phân biệt xuất hiện của yêu cầu trong bộ nhớ vật lý sau khi đã hiệu chỉnh, sáp nhập các trang.
- struct list_head queuelist;
  - Cấu trúc danh sách liên kết
  - Sử dụng để liên kết yêu cầu với hàng đợi yêu cầu

Các trường trong cấu trúc request



Các hàm hoàn thiện yêu cầu

- Khi thiết bị hoàn thành chuyển một số hoặc toàn bộ các sector trong một yêu cầu vào/ra, nó phải báo cho hệ thống con biết bằng hàm:

```
int end_that_request_first(struct request *req, int success, int count);
```

- Hàm này báo cho đoạn mã khối biết trình điều khiển đã kết thúc chuyển số sector là count.
  - Nếu chuyển thành công, success là 1; trái lại, truyền 0.
  - Giá trị trả về của hàm trên cho biết mọi sector trong yêu cầu đã được chuyển hay chưa. Trả về 0 nghĩa là mọi sector đã được chuyển và yêu cầu được xử lý thành công

Các hàm hoàn thiện yêu cầu

- Cần phải bỏ yêu cầu ra khỏi hàng đợi với hàm blkdev_dequeue_request:
  - `void end_that_request_last(struct request *req);`
- end_that_request_last báo cho mọi tiến trình, hệ thống khác đang chờ trên yêu cầu này về trạng thái hoàn thành của yêu cầu; nó phải được gọi với một khóa hàng đợi được duy trì.
- Đoạn mã minh họa:

```
void end_request(struct request *req, int uptodate){
    if (!end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
        add_disk_randomness(req->rq_disk);
        blkdev_dequeue_request(req);
        end_that_request_last(req);
}}
```

Phân tích ví dụ

- https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html

HỎI - ĐÁP