# Unit 5: Block Device Drivers

Prof. Li-Pin Chang
ESSLab@NCTU

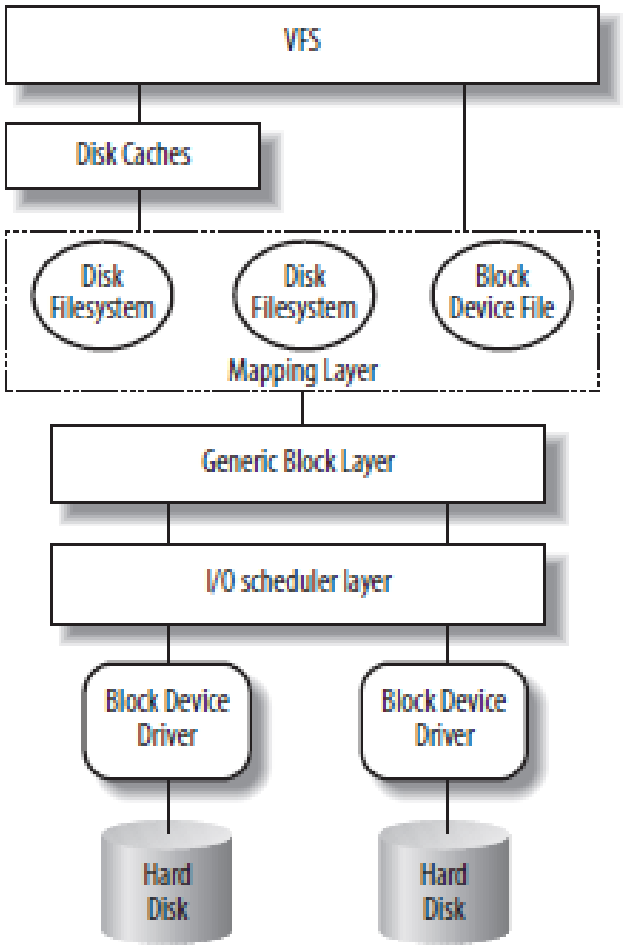
# Block Devices vs. Character Devices

- Block devices
  - High data transfer rates
  - Block access to minimize I/O overhead
  - Data access unit: blocks (512B~4 KB typically)
  - Hard disks, CD-ROM, etc

- Character devices
  - Slow I/O devices
  - Data access unit: bytes
  - Keyboards, mice, TTY, etc

# Related Kernel Structures

- Block devices are abstraction of storage devices such as hard drives, CDROM, SSD, etc
- Disk Cache caches the recently used pages in main memory
- File system hides block device characteristics from applications, and manages data allocations in block devices
  - Mapping (file,ofst) → (dev, ofst)
- I/O scheduler re-order disk requests for efficient access



# Organization of Block Devices

- A disk volume has one or more partitions
  - Disk and partitions are major and minor devices, respectively
  - Sector addresses relative to the beginning of their container (disk or partition)

# Device Nodes

- User space can access block devices via device files (i.e., device nodes) in /dev
  - sda, sdb….
- Device nodes are not regular files, they can be created using “mknod” command
  - E.g., mknod /dev/sda b 8 0 (see below)
  - contains nothing but a pair of major, minor number
- Devices are associated with a major number and a minor number
  - See [1] for a list of pre-defined major numbers
  - For example, sda=(8,0) and sdb=(8,16)
    - sda1=(8,1) and sdb2=(8,18)

[1] https://www.kernel.org/doc/Documentation/devices.txt

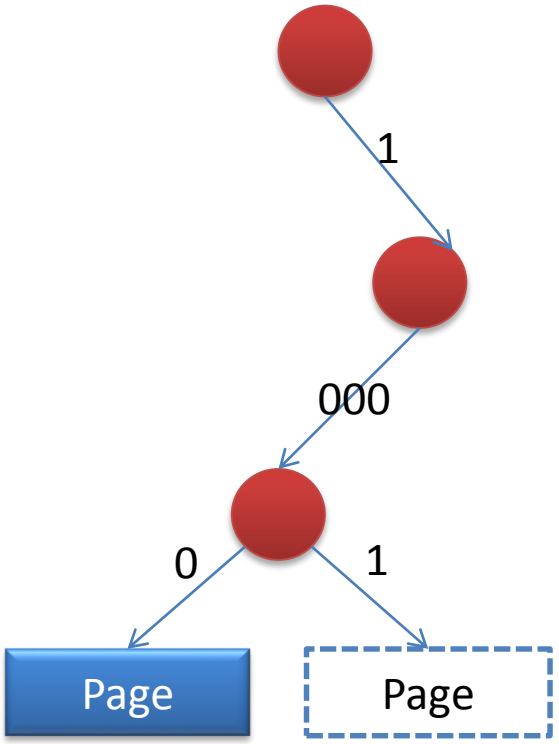
# Ramdisk Concepts

- Ramdisk concept
  - Using VM pages to store data
    - No bulk memory to emulate the “virtual disk space” to avoid double space overhead
    - Pages cannot be swapped out (why?)
    - Pages are allocated with GFP_NOIO
  - Allocating pages on demand
  - Ramdisks: major number of “ram”=1
    - ram0=(1,0)~ram15 (1,15), like fifteen partitions of “ram”

# Ramdisk Concepts

- Pages are indexed in terms of their disk offsets, not their memory addresses
  - Input: a disk offset; output: a page

- E.g., 1 page=4 KB, 1 sector = 512B
  - Reading from sector 129
    - Reading from page 129/8=16=10000b
  - Writing to sector 140
    - Writing to page 140/8=17=10001b
    - There's no such page, allocate it




A simplified radix tree

# Radix Tree

- *radix_tree_insert*(&brd->brd_pages, idx, page)
  - Insert a page indexed by idx to the tree brd_pages
- *radix_tree_lookup*(&brd->brd_pages, idx)
  - Return the page indexed idx
- *radix_tree_delete*(&brd->brd_pages, idx)
  - Delete the page indexed by idx, return the deleted page

# Block Device Initialization

- Linux/drivers/block/brd.c
- Fill the pointers to functions in the structure *block_device_operations*, including
  - An ioctl handler
  - A direct access handler (for XIP)
  - No read/write handlers?

```
static const struct block_device_operations brd_fops = {
        .owner =        THIS_MODULE,
        .locked_ioctl =         brd_ioctl,
#ifdef CONFIG_BLK_DEV_XIP
        .direct_access =    brd_direct_access,
#endif
};
```

# Block Device Initialization

```
static int __init brd_init(void)
{
        int i, nr;
        unsigned long range;
        struct brd_device *brd, *next;

        ...

        if (register_blkdev(RAMDISK_MAJOR, "ramdisk"))
                return -EIO;

        for (i = 0; i < nr; i++) {
                brd = brd_alloc(i);
                if (!brd)
                        goto ↓out_free;
                list_add_tail(&brd->brd_list, &brd_devices);
        }

        list_for_each_entry(brd, &brd_devices, brd_list)
                add_disk(brd->brd_disk);

        blk_register_region(MKDEV(RAMDISK_MAJOR, 0), range,
                            THIS_MODULE, brd_probe, NULL, NULL);

        printk(KERN_INFO "brd: module loaded\n");
        return 0;

out_free:
        ...

} ? end brd_init ?
```

Register the device major/minor number

# Block Device Initialization

- *brd_alloc*()
  - Create a request queue
  - Tell the scheduler that no dispatch sorting is necessary, just pass requests in
    - *blk_queue_make_request(brd->*brd_queue, brd_make_request);
    - Bypassing *elv_dispatch_sort*() in disk schedulers
  - Register block device operations
    - *brd_fops*
    - Only ioctl, the rest are left to default

- Called 16 times for creating ram0~ram15

# Remark: Kernel Memory Allocation

- *kmalloc*()
  - Allocating physically contiguous memory (for I/O)
  - prone to allocation failure
- *vmalloc*()
  - The allocated memory is logically contiguous but may not be physically contiguous
- *get_free_pages*()
  - Allocate memory in terms of page
  - Allocated pages are not yet mapped to kernel memory space

# Remark: GFP Flags

- Used by get_free_pages and kmalloc()
- *GFP_ATOMIC*
  - No sleep, no I/O
  - For interrupt handlers
- *GFP_NOIO*
  - May sleep but no I/O
  - Used by storage-related drivers
- *GFP_KERNEL*
  - May sleep and may I/O
  - General use
  - Storage-related drivers can use it in the init/exit routines

```
static struct brd_device *brd_alloc(int i)
{
        struct brd_device *brd;
        struct gendisk *disk;

        ...
        INIT_RADIX_TREE(&brd->brd_pages, GFP_ATOMIC);

        brd->brd_queue = blk_alloc_queue(GFP_KERNEL);
        if (!brd->brd_queue)
                goto out_free_dev;
        blk_queue_make_request(brd->brd_queue, brd_make_request);
        blk_queue_ordered(brd->brd_queue, QUEUE_ORDERED_TAG, NULL);
        blk_queue_max_sectors(brd->brd_queue, 1024);
        blk_queue_bounce_limit(brd->brd_queue, BLK_BOUNCE_ANY);

        disk = brd->brd_disk = alloc_disk(1 << part_shift);
        if (!disk)
                goto out_free_queue;
        disk->major             = RAMDISK_MAJOR;
        disk->first_minor       = i << part_shift;
        disk->fops              = &brd_fops;
        disk->private_data      = brd;
        disk->queue             = brd->brd_queue;
        disk->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
        sprintf(disk->disk_name, "ram%d", i);
        set_capacity(disk, rd_size * 2);

        return brd;
        ...
} ? end brd_alloc ?
```

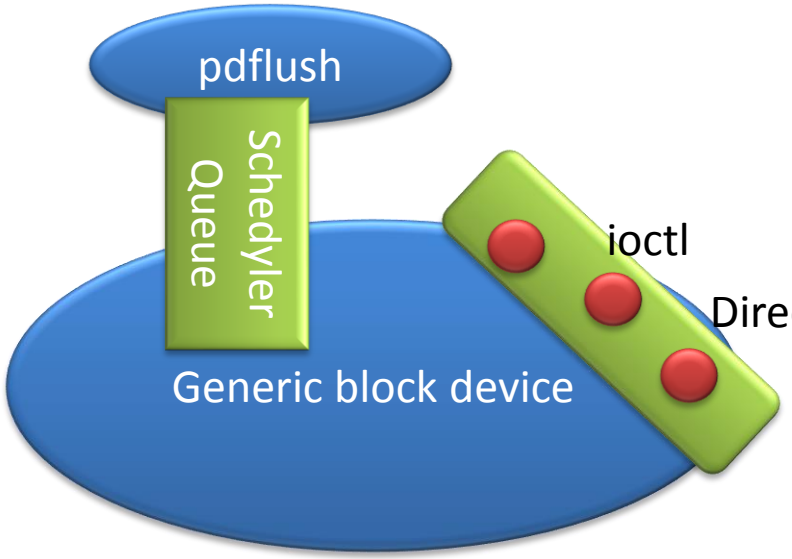
Register a "make_request" handler, bypassing the scheduler

1 req = 1024 sectors max

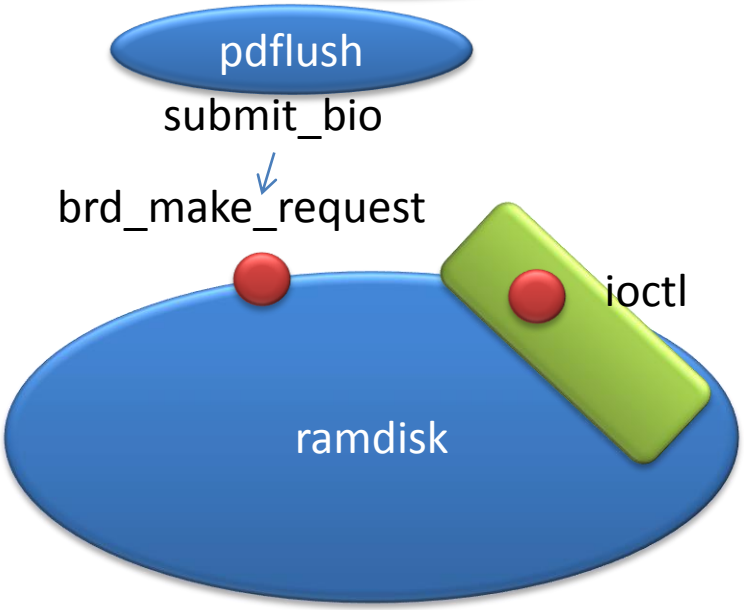
Means "never bounce"

Note this

# Block Device Interfaces




- In generic block device interface, the disk scheduler will sort the request queue




- ramdisk registers only two handlers and leave the rest to default.
- Does not use the request queue

# Handling Read and Write

- *brd_make_request*() will be called when the upper layers submit a *bio* to the ramdisk driver
  - The I/O scheduler is bypassed
  - Processing READ (or READA) and WRITE commands
  - The *bio* structure describes an I/O operation on a block device
    - An I/O operation may involve multiple pages that are not physically continuous
    - A *bio* contains multiple segments (vectors), each of which describes a set of continuous pages
  - Calls *brd_do_bvec*() to handle one *bio_vec* at a time

```
static int brd_make_request(struct request_queue *q, struct bio *bio)
{
        struct block_device *bdev = bio->bi_bdev;
        struct brd_device *brd = bdev->bd_disk->private_data;
        int rw;
        struct bio_vec *bvec;
        sector_t sector;
        int i;
        int err = -EIO;

        sector = bio->bi_sector;
        if (sector + (bio->bi_size >> SECTOR_SHIFT) >
                                        get_capacity(bdev->bd_disk))
                goto out;

        rw = bio_rw(bio);
        if (rw == READA)
                rw = READ;

        bio_for_each_segment(bvec, bio, i) {
                unsigned int len = bvec->bv_len;
                err = brd_do_bvec(brd, bvec->bv_page, len,
                                bvec->bv_offset, rw, sector);
                if (err)
                        break;
                sector += len >> SECTOR_SHIFT;
        }

out:
        bio_endio(bio, err);

        return 0;
}
```

Vector ~= segment

# Handling Read and Write

- Structure of *bio*

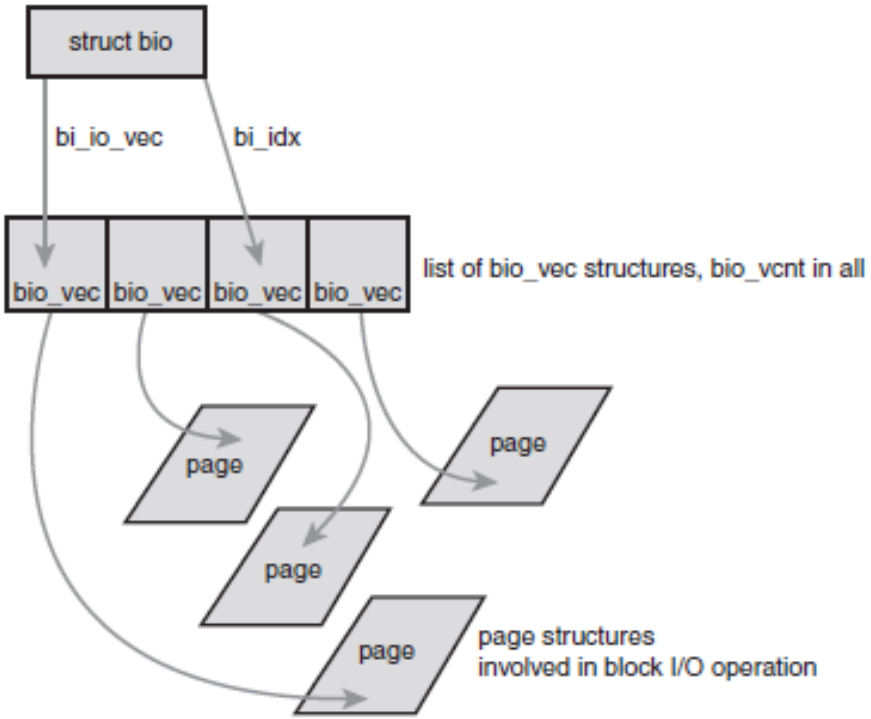



Figure 14.2 Relationship between **struct bio**, **struct bio_vec**, and **struct page**.

bio_vec:
(page #, byte len, byte offset)

# bio (block I/O) Structure

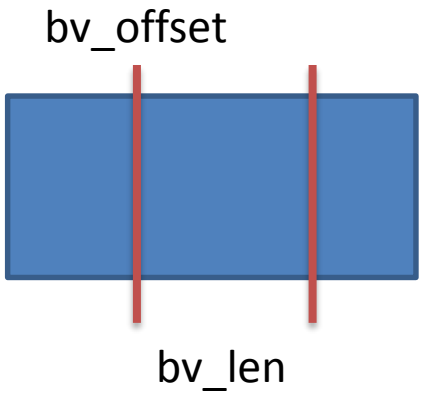
```
struct bio {
        sector_t                bi_sector;      /* associated sector on disk */
        struct bio              *bi_next;       /* list of requests */
        struct block_device     *bi_bdev;       /* associated block device */
        unsigned long           bi_flags;       /* status and command flags */
        unsigned long           bi_rw;          /* read or write? */
        unsigned short          bi_vcnt;        /* number of bio_vecs off */
        unsigned short          bi_idx;         /* current index in bi_io_vec */
        unsigned short          bi_phys_segments; /* number of segments */
        unsigned int            bi_size;        /* I/O count */
        unsigned int            bi_seg_front_size; /* size of first segment */
        unsigned int            bi_seg_back_size;  /* size of last segment */
        unsigned int            bi_max_vecs;    /* maximum bio_vecs possible */
        unsigned int            bi_comp_cpu;    /* completion CPU */
        atomic_t                bi_cnt;         /* usage counter */
        struct bio_vec          *bi_io_vec;     /* bio_vec list */
        bio_end_io_t            *bi_end_io;     /* I/O completion method */
        void                    *bi_private;    /* owner-private method */
        bio_destructor_t        *bi_destructor; /* destructor method */
        struct bio_vec          bi_inline_vecs[0]; /* inline bio vectors */
};
```


bi_io_vec[0…bi_vcnt-1]

# bio Vector

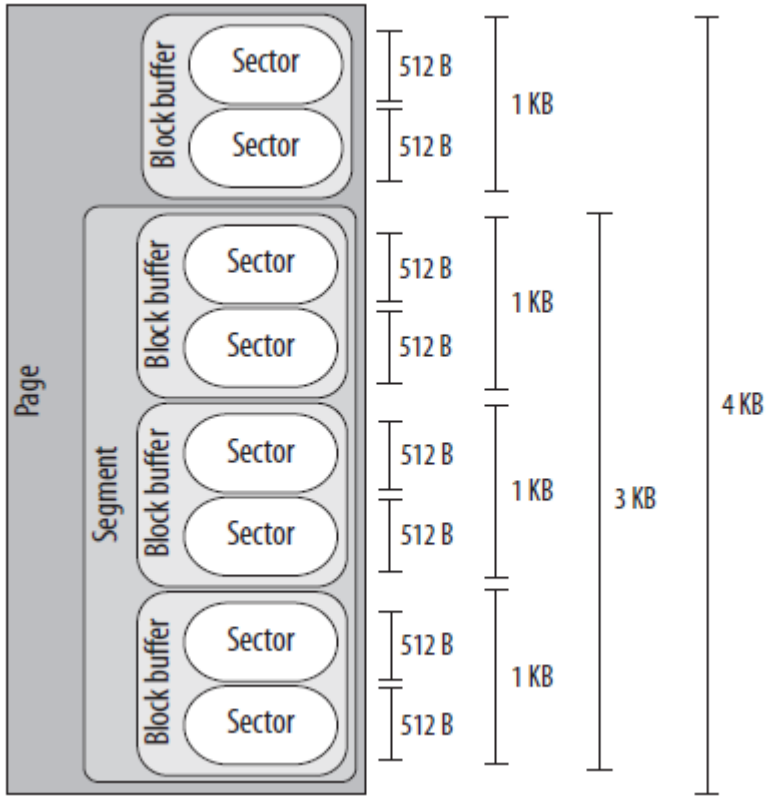
*bv_page* must be mapped to kernel memory before reading/writing

```
struct bio_vec {
        /* pointer to the physical page on which this buffer resides */
        struct page     *bv_page;

        /* the length in bytes of this buffer */
        unsigned int    bv_len;

        /* the byte offset within the page where the buffer resides */
        unsigned int    bv_offset;
};
```



# Pages, Segments, Buffers, and Sectors

- **Pages**: high-level operations, close to the virtual memory system
  - Used by page cache
  - Size subject to MMU
- **Segments**: vector I/O units
- **Block buffers**: disk block buffer heads
  - File system allocation unit size
  - Size <= page size
  - Size subject to file system
- **Sectors**: disk access units
  - Size subject to the physical device



# Pages, Segments, Buffers, and Sectors

- Sectors (2^n bytes)
  - Device hardware access units; 512B for HDD and 2048 B for CDROM
- Blocks (2^n bytes)
  - VFS (file system) allocation units; usually 1KB ~ 4KB; cannot be larger than a page (1 KB, 2 KB, or 4 KB in ext4 file system)
- Segments
  - Partial or whole pages
- Pages (2^n bytes)
  - Virtual memory management units; 4 KB for x86 machines

# Handling Read and Write

- *brd_do_bvec*()
  - *copy_to_brd_setup*(): allocate the written pages on demand.
    - May sleep, because it calls *alloc_page(gfp_flags*) with gfp_flags = GFP_NOIO | __GFP_ZERO
    - But must not cause I/O (page fault) to avoid deadlocks
  - *copy_from_brd*() for READ and *copy_to_brd*() for WRITE
    - Find the requested page in the radix tree
    - Copy data between the page in the radix tree and the page associated with the *bio_vec*

```
static int brd_do_bvec(struct brd_device *brd, struct page *page,
                       unsigned int len, unsigned int off, int rw,
                       sector_t sector)
{
        void *mem;
        int err = 0;

        if (rw ! = READ) {
                err = copy_to_brd_setup(brd, sector, len);
                if (err)
                        goto ↓out;
        }

        mem = kmap_atomic(page, KM_USER0);
        if (rw == READ) {
                copy_from_brd(mem + off, brd, sector, len);
                flush_dcache_page(page);
        } else {
                flush_dcache_page(page);
                copy_to_brd(brd, mem + off, sector, len);
        }
        kunmap_atomic(mem, KM_USER0);

out:
        return err;
} ? end brd_do_bvec ?
```

```
static void copy_from_brd(void *dst, struct brd_device *brd,
                sector_t sector, size_t n)
{
        struct page *page;
        void *src;
        unsigned int offset = (sector & (PAGE_SECTORS-1)) << SECTOR_SHIFT;
        size_t copy;

        copy = min_t(size_t, n, PAGE_SIZE - offset);
        page = brd_lookup_page(brd, sector);
        if (page) {
                src = kmap_atomic(page, KM_USER1);
                memcpy(dst, src + offset, copy);
                kunmap_atomic(src, KM_USER1);
        } else
                memset(dst, 0, copy);

        if (copy < n) {
                dst += copy;
                sector += copy >> SECTOR_SHIFT;
                copy = n - copy;
                page = brd_lookup_page(brd, sector);
                if (page) {
                        src = kmap_atomic(page, KM_USER1);
                        memcpy(dst, src, copy);
                        kunmap_atomic(src, KM_USER1);
                } else
                        memset(dst, 0, copy);
        }
} ? end copy_from_brd ?
```

- Neither the user-land pages nor the cached pages are mapped to the kernel space yet. Thus, *kmap_atomic*() is called for the two types of pages

```
static void copy_to_brd(struct brd_device *brd, const void *src,
                        sector_t sector, size_t n)
{
        struct page *page;
        void *dst;
        unsigned int offset = (sector & (PAGE_SECTORS-1)) << SECTOR_SHIFT;
        size_t copy;

        copy = min_t(size_t, n, PAGE_SIZE - offset);
        page = brd_lookup_page(brd, sector);
        BUG_ON(!page);

        dst = kmap_atomic(page, KM_USER1);
        memcpy(dst + offset, src, copy);
        kunmap_atomic(dst, KM_USER1);

        if (copy < n) {
                src += copy;
                sector += copy >> SECTOR_SHIFT;
                copy = n - copy;
                page = brd_lookup_page(brd, sector);
                BUG_ON(!page);

                dst = kmap_atomic(page, KM_USER1);
                memcpy(dst, src, copy);
                kunmap_atomic(dst, KM_USER1);
        }
} ? end copy_to_brd ?
```

# Ioctl

- A special channel between user programs kernel drivers
  - for passing special control commands other than the standard block commands
    - E.g., set device speed, get disk geometry, and turn on/off special device functionalities
  - See [1] for a list of pre-defined ioctl codes
  - You can implement the handlers for your own ioctl codes in your driver
    - Do not conflict with the existing ioctl codes!

[1] Linux/Documentation/ioctl/ioctl-number.txt

```
static int brd_ioctl(struct block_device *bdev, fmode_t mode,
                     unsigned int cmd, unsigned long arg)
{
        int error;
        struct brd_device *brd = bdev->bd_disk->private_data;

        if (cmd != BLKFLSBUF)
                return -ENOTTY;

        /*
         * ram device BLKFLSBUF has special semantics, we want to actually
         * release and destroy the ramdisk data.
         */
        mutex_lock(&bdev->bd_mutex);
        error = -EBUSY;
        if (bdev->bd_openers <= 1) {
                /*
                 * Invalidate the cache first, so it isn't written
                 * back to the device.
                 *
                 * Another thread might instantiate more buffercache here,
                 * but there is not much we can do to close that race.
                 */
                invalidate_bh_lrus();
                truncate_inode_pages(bdev->bd_inode->i_mapping, 0);
                brd_free_pages(brd);
                error = 0;
        }
        mutex_unlock(&bdev->bd_mutex);

        return error;
} ? end brd_ioctl ?
```

Ramdisk currently support only BLKFLSBUF to release all the allocated pages

# ioctl Codes

Linux/Documentation/ioctl/ioctl-number.txt

```
Code Seq#(hex) Include File Comments
========================================================
0x00 00-1F linux/fs.h conflict!
0x00 00-1F scsi/scsi_ioctl.h conflict!
0x00 00-1F linux/fb.h conflict!
0x00 00-1F linux/wavefront.h conflict!
0x02 all linux/fd.h
0x03 all linux/hdreg.h
0x04 D2-DC linux/umsdos_fs.h Dead since 2.6.11, but don't reuse these.
0x06 all linux/lp.h
0x09 all linux/raid/md_u.h
0x10 00-0F drivers/char/s390/vmcp.h
0x12 all linux/fs.h
linux/blkpg.h
```

Linux/include/linux/fs.h

```
/* the read-only stuff doesn't really belong here, but any other place is
   probably as bad and I don't want to create yet another include file. */

#define BLKROSET   _IO(0x12,93)   /* set device read-only (0 = read-write) */
#define BLKROGET   _IO(0x12,94)   /* get read-only status (0 = read_write) */
#define BLKRRPART  _IO(0x12,95)   /* re-read partition table */
#define BLKGETSIZE _IO(0x12,96)   /* return device size /512 (long *arg) */
#define BLKFLSBUF  _IO(0x12,97)   /* flush buffer cache */
#define BLKRASET   _IO(0x12,98)   /* set read ahead for block device */
#define BLKRAGET   _IO(0x12,99)   /* get current read ahead setting */
#define BLKFRASET  _IO(0x12,100)/* set filesystem (mm/filemap.c) read-ahead */
#define BLKFRAGET  _IO(0x12,101)/* get filesystem (mm/filemap.c) read-ahead */
#define BLKSECTSET _IO(0x12,102)/* set max sectors per request (ll_rw_blk.c) */
#define BLKSECTGET _IO(0x12,103)/* get max sectors per request (ll_rw_blk.c) */
#define BLKSSZGET  _IO(0x12,104)/* get block device sector size */
```

# ioctl

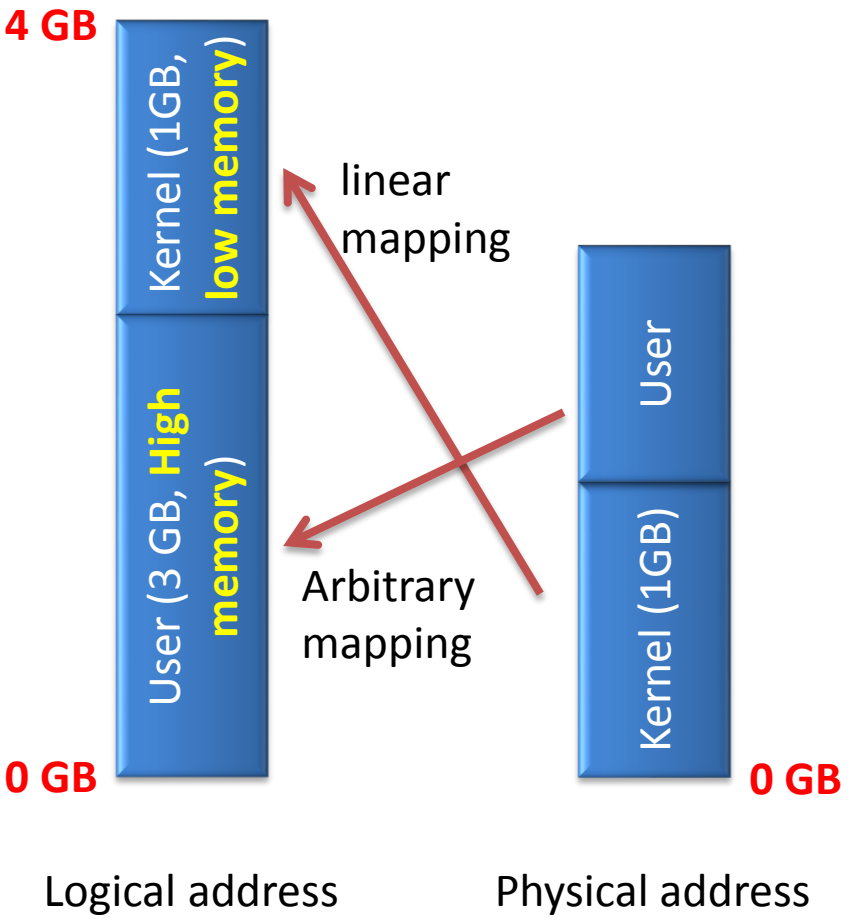
- How to pass ioctl codes from user programs to the ramdisk driver?
  - first open the device node using fd=*open*(“/dev/ram0”,O_RDWR);
  - then, call *ioctl*(fd,BLKFLSHBUF,0)
  - BLKFLSHBUF must be the same integer in the user program and in the kernel driver

# ioctl Buffers

- User space can pass a buffer to driver
  - *ioctl*(fd, CMD, (void *)buffer)
- Kernel driver calls *get_user*() and *put_user*() to get variables in the user-land buffer
  - static int *brd_ioctl*(… , unsigned long arg)
  - *get_user*(&data,(char *)arg)

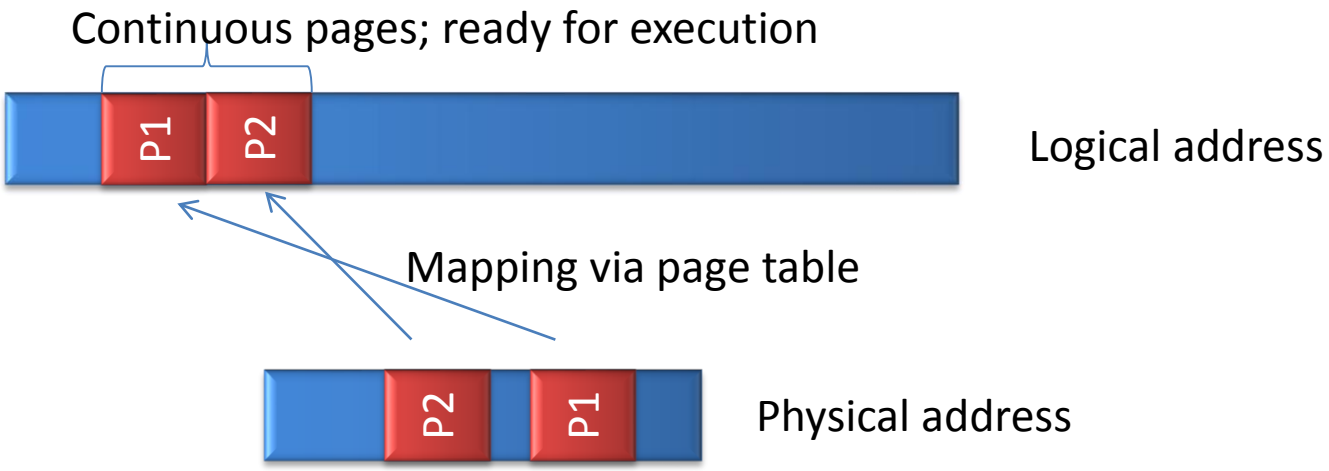
# Bounce Buffers and High Memory

- Legacy devices cannot perform DMA on pages in high memory
  - DMA in ISA devices uses 24 bits
- Drivers of legacy devices can create "bounce buffers" in low memory as an intermediate buffer
  - Extra data copy required
- Kernel code cannot access high-memory pages, unless these pages are mapped to kernel space first
  - *kmap*, *kunmap* for permeant mapping
  - *kmap_atomic* and *kunmap_atomic* for temporarily mapping



# XIP: eXecute In Place

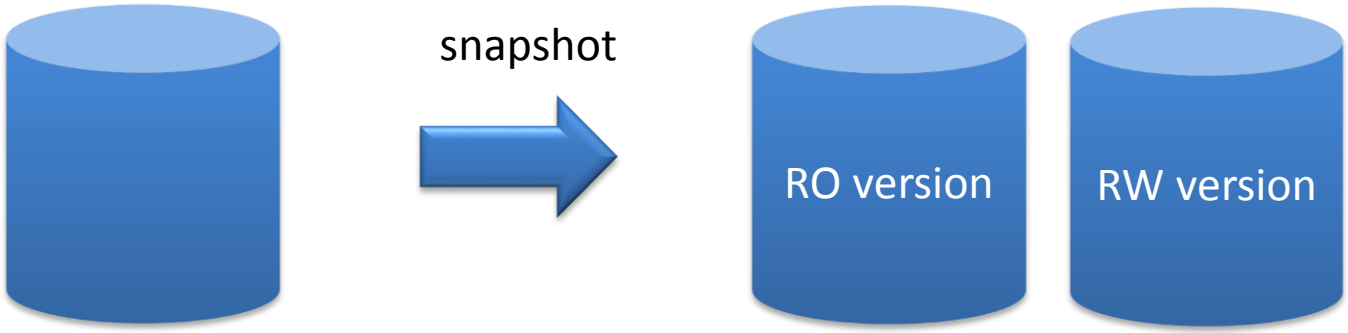
- For block devices whose storage space is already a part of the CPU address space
  - E.g., NOR flash and ramdisk
- The OS modify the page table and maps the “disk blocks” of the executable into the process logical space
  - Saves the cost of page copying and avoid double page allocation



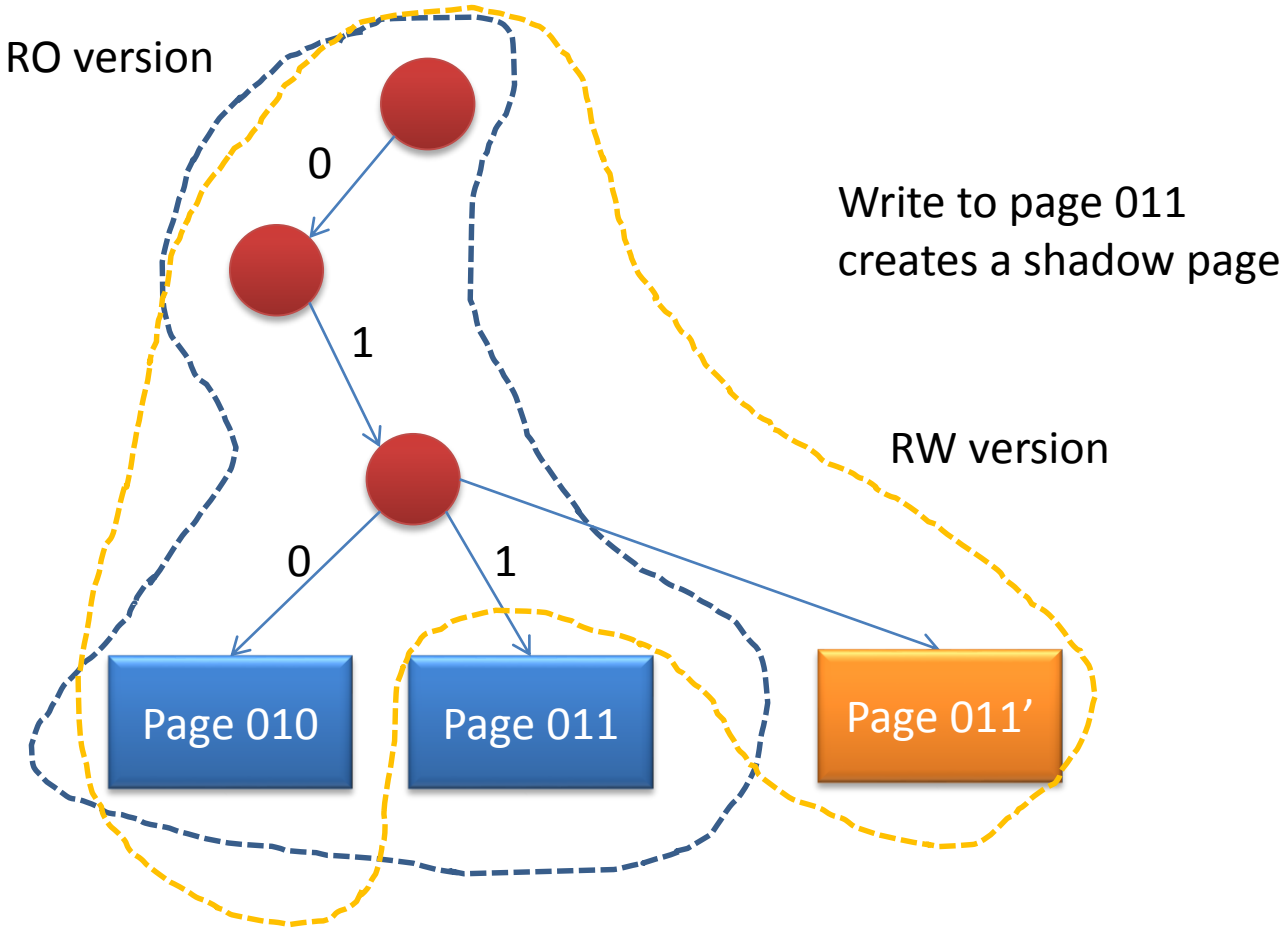
# Lab 8: Ramdisk snapshot

# Lab Objective

- Adding the support of snapshot in the ramdisk driver
- Disk snapshot is very useful for disaster recovery and system backup
  - Taking a snapshot on a disk creates a read-only version and a read-write version of the disk
  - Copy on write to save memory usage

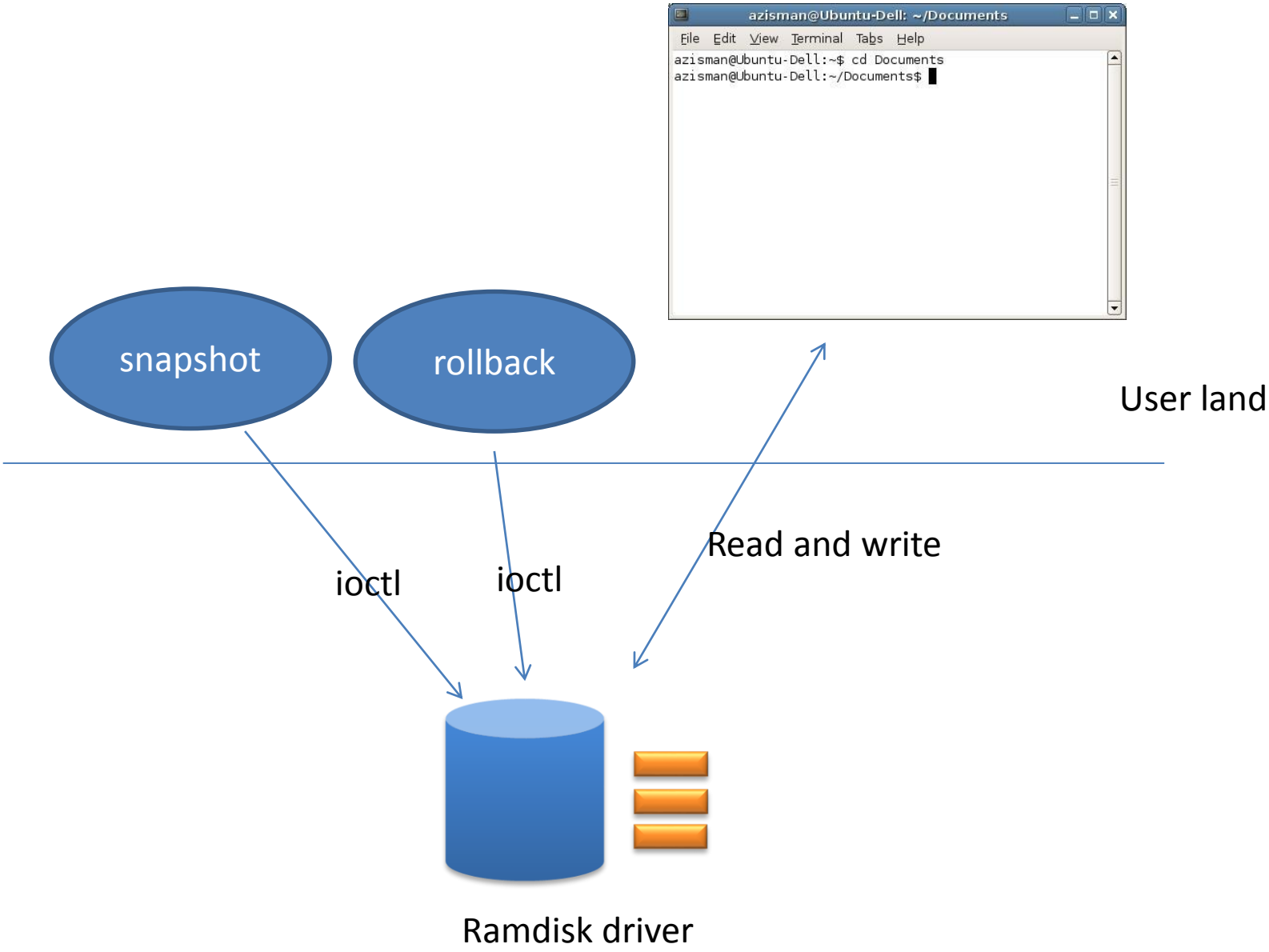


# Shadow Pages



# Snapshot and Rollback

- Implement two basic commands
  - Snapshot: taking a ramdisk snapshot
  - Rollback: delete the snapshot and revert to the old disk image
- Coding efforts
  - Write two user programs calls *ioctl*() to communicate with the ramdisk driver; one for snapshot and the other for rollback
  - Add the corresponding ioctl handlers in the ramdisk driver
  - Handle creation and deletion of shadow pages
  - Revise read and write operations


azisman@Ubuntu-Dell: ~/Documents
File Edit View Terminal Tabs Help
azisman@Ubuntu-Dell:~$ cd Documents
azisman@Ubuntu-Dell:~/Documents$
snapshot
rollback
User land
ioctl
ioctl
Read and write
Ramdisk driver

# References


- Robert love, “Linux Kernel Development 3rd Edition,” 2010
- Daniel B. Bovet et al., “Understanding the Linux Kernel 3rd Edition,” 2005
- Jonathan Corbet et al., “Linux Device Drivers 3rd Edition,” 2005
  - Ch16, http://lwn.net/Kernel/LDD3/
- Linux kernel source tree 2.6.34