Linux Kernel Development 06

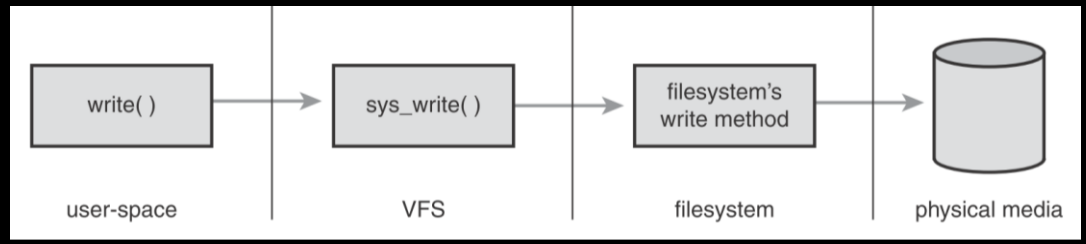
工海系 b06501018 朱紹勳

	Linux Kernel Dev 3rd	Understand Linux Kernel 3rd	
Virtual File System	Chapter 13	Chapter 12	Jserv <Linux核心設計>
Ext2 & Ext3 File system	Chapter 13	Chapter 18	Jserv <Linux核心設計>
I/O & Block	Chapter 14	Chapter 13	
Device Driver & Module	Chapter 17	Chapter 14	Jserv <Linux核心設計>

Virtual File System

Intro of VFS

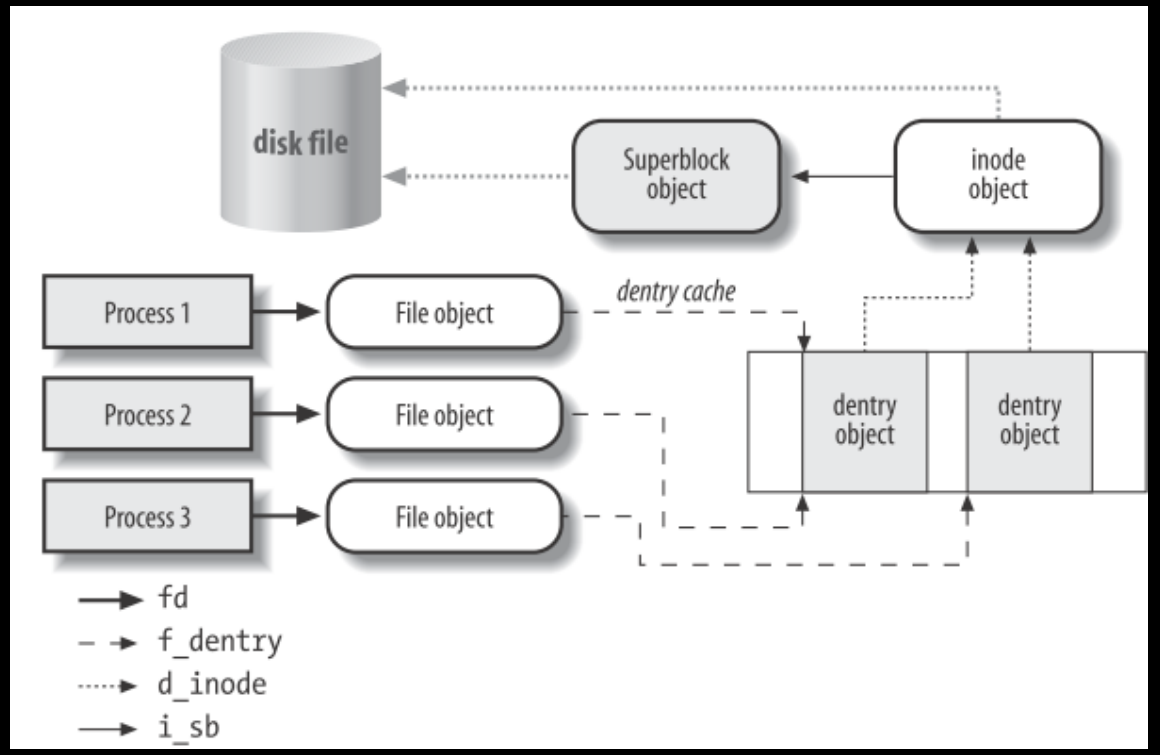
- A subsystem of kernel, work regardless of the file system or underlying physical medium (virtual interface)
- File system Abstraction Layer:
  - Make linux support file systems(actual file system hides implementation details)
  - ex: `ret = write(fd, buf, len);`
- Unix Filesystems
  - Four basic filesystem-related abstractions:
    - Files, Directory Entries, Inode(index node), mount(安裝) point



  - A filesystem is a hierarchical storage of data addressing to a specific structure
  - The mount point is called namespace in global hierarchy
  - Files are organized in directories (just like folder, which are actually normal file in Unix)
  - The metadata of file is store in a separate data structure called inode
  - The control information of filesystem is store in superblock

Data Structure of VFS & its object

- The structures contain both data and pointers, methods.
- Superblock : mounted file system
  - `super_operations`, ex: `write_inode()` and `sync_fs()`
- Inode : file
  - `inode_operations`, ex: `create()` and `link()`
- Dentry : a component of path
  - `dentry_operations obj`, ex: `d_compare()` and `d_delete()`
- File : an open file as associated with a process
  - `file_operations obj`, ex: `read()` and `write()`



The Superblock Object

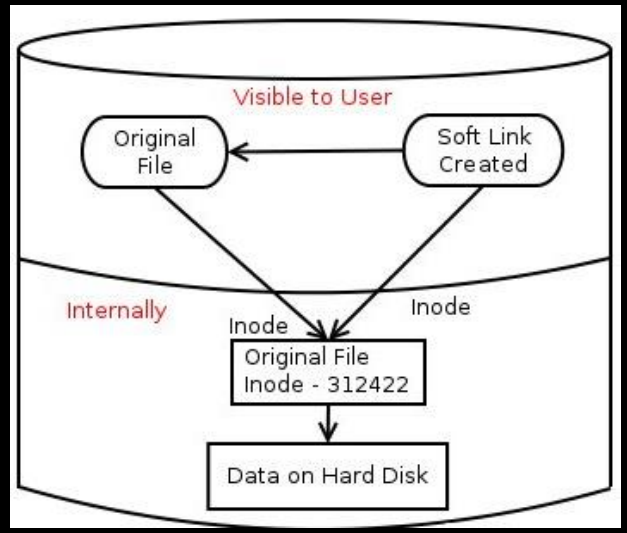
- Store information describing that specific file system, special sector on disk
- Defined in `<linux/fs.h>`, operations inplement in `<fs/super.c>`
- Initialize using `alloc_super()`
- C: `sb->s_op->write_super(sb);` vs. C++: `sb.write_super();`
- Operations:
  - Inode
    - `alloc_inode()`, `destroy_inode()`, `dirty_inode()`, `write_inode()`, `delete_inode()`, `drop_inode()`, `clear_inode()`…
  - Super
    - `put_super()`, `write_super()`
  - Filesystem
    - `sync_fs()`, `write_super_lockfs()`, `unlockfs()`, `statfs()`, `remount_fs()`

The Inode Object

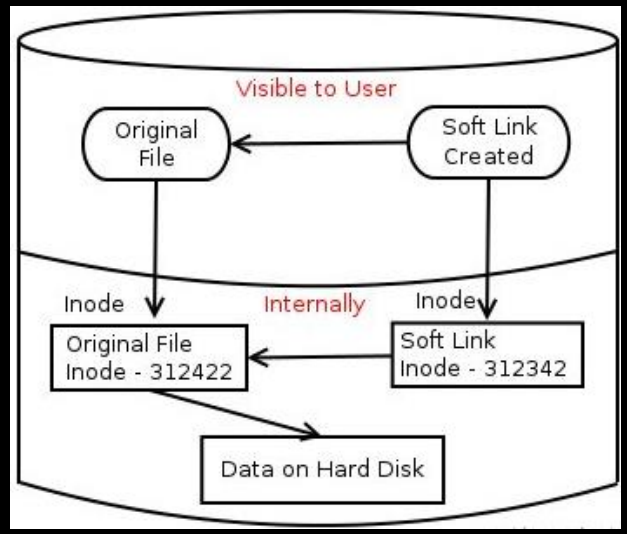
- Information needed by the kernel to manipulate a file or directory
- Defined in `<linux/fs.h>`
- Operations:
  - File
    - `create()`, `lookup()`, `rename()`
  - Directory
    - `mkdir()`, `rmdir()`
  - Link
    - `link()`, `unlink()`, `symlink()`, `readlink()`, `follow_link()`, `put_link()`
  - Attribute
    - `setattr()`, `getattr()`, `listxattr()`, `removexattr()`

Links

- Inode contain many information about file, but no file name!
- Hard Link
  - Different filename corresponding to same inode -> Hard Link
  - Just like deep copy, delete will not effect others existance
  - Same file size, same inode number
  - Cannot cross filesystem, cannot link to dentry



- Soft(symbolic) link, short cut
  - File that contains a reference to another file or directory (inode) in the form of path
  - Just like pointer. Soft link is useless if original file is deleted.
  - Smaller file size, different inode number



The Dentry(目錄項) Object

- All dentry are specialize file. Has no corresponding disk data structure
- A valid dentry has three states: used, unused, or negative.
    - d_inode points to an associated inode
    - d_inode points to an inode
    - d_inode is NULL
- Dentry cache
    - Lists of “used” dentries, A doubly linked “LRU”, hash table
- Operation:
    - `d_revalidate(), d_hash(), d_compare(), d_delete(), d_release(), d_iput()`

The File Object

- Represent a file opened by a process (in memory)
- Multiple process can open same file → multiple file objects in existence for the same file
- Operations:
  - ioctl:

    ```
    ioctl(), unlocked_ioctl(), compat_ioctl()
    ```

  - sync:

    ```
    fsync(), aio_fsync(), fasync()
    ```

  - Other:

    ```
    llseek(), read(), aio_read(), write(), aio_write()
    readdir(), poll(),
    open(), flush(), release(), mmap(), readv(), writev(), sendfile(),
    sendpage(), get_unmapped_area()
    ```

Data Structures Attributes related to FS

- `struct file_system_type` : describe fFS type, ext2, ext3…; `<linux/fs.h>`
  - `get_sb()` : read superblock from disk; `kill_sb()`
  - locks
- `struct vfsmount` : instance to mount the FS; `<linux/mount.h>`

Data Structures Attributes related to Process

- `file_struct` : pointed by file entry, store per process info; `<linux/fdtable.h>`
- `fs_struct` : pointed by file desc, store FS, process info; `<linux/fs_strut.h>`
- `namespace` : pointed by mmt_namespace; <linux/namespace.h>

Intro of FileSystem

- Ext2(extended filesystem)
  - Max. file size : 2TB, Max volume size : 4TB, Max file name : 255 char
  - Old, may be the most widely used linux filesystem
  - may need enormous time to find the breaking file to recover
- Ext3
  - Add Journaling feature
    - When there is update, it will write information to journal at first, then update filesystem
    - It can optimize the recovery time when facing kernel break down
- Ext4
  - Add check sum to improve filesystem reliability
- Btrfs
  - Copy-On-Write, Snapshot
- etc…

I/O & Block

Intro of Block

- Random access fixed-size of data from HW, ex: disk, flash memory, character device
- The smallest addressable unit on a block device is a sector(2^n bytes)
- Kernel performs all disk operations in terms of **blocks**
- The size of block:
  - n times of sector, < size of page(kernel)

- **Buffers and Buffer Heads**
  - Each buffer is associated with exactly one block
  - buffer_head is the descriptor of buffer, defined in <linux/buffer_head.h>

```
struct buffer_head {
        unsigned long b_state;       /* buffer state flags */
        struct buffer_head *b_this_page; /* list of page's buffers */
        struct page *b_page;         /* associated page */
        sector_t b_blocknr;          /* starting block number */
        size_t b_size;               /* size of mapping */
        char *b_data;                /* pointer to data within the page */
        struct block_device *b_bdev; /* associated block device */
        bh_end_io_t *b_end_io;       /* I/O completion */
        void *b_private;             /* reserved for b_end_io */
        struct list_head b_assoc_buffers;  /* associated mappings */
        struct address_space *b_assoc_map; /* associated address space */
        atomic_t b_count;            /* use count */
};
```

```
get_bh() // increase buf reference count
put_bh() // decrease buf reference count
```

The physical block on disk :

logical block(b_bdev)

The physical page in memory :

b_data ~ (b_data + b_size)

Status Flag	Meaning
BH_Uptodate	Buffer contains valid data.
BH_Dirty	Buffer is dirty. (The contents of the buffer are newer than the contents of the block on disk and therefore the buffer must eventually be written back to disk.)
BH_Lock	Buffer is undergoing disk I/O and is locked to prevent concurrent access.
BH_Req	Buffer is involved in an I/O request.
BH_Mapped	Buffer is a valid buffer mapped to an on-disk block.
BH_New	Buffer is newly mapped via `get_block()` and not yet accessed.
BH_Async_Read	Buffer is undergoing asynchronous read I/O via `end_buffer_async_read()`.
BH_Async_Write	Buffer is undergoing asynchronous write I/O via `end_buffer_async_write()`.
BH_Delay	Buffer does not yet have an associated on-disk block (delayed allocation).
BH_Boundary	Buffer forms the boundary of contiguous blocks—the next block is discontinuous.
BH_Write_EIO	Buffer incurred an I/O error on write.
BH_Ordered	Ordered write.
BH_Eopnotsupp	Buffer incurred a "not supported" error.
BH_Unwritten	Space for the buffer has been allocated on disk but the actual data has not yet been written out.
BH_Quiet	Suppress errors for this buffer.

Currently, kernel work directly with pages and address spaces instead of buffers

- **bio structure(New method):**
  - Represent the segment of kernel block I/O
  - defined in <linux/bio.h>
  - I/O vector
    - bi_io_vec field points to an array of bio_vec structures
    - bio_vec : <page, offset, len>
    - Each block I/O request is represented by a bio structure
      - Each request is composed of one or more blocks
      - block s are store in bio_vec, which also represent the actual segment's location in a physical page in memory

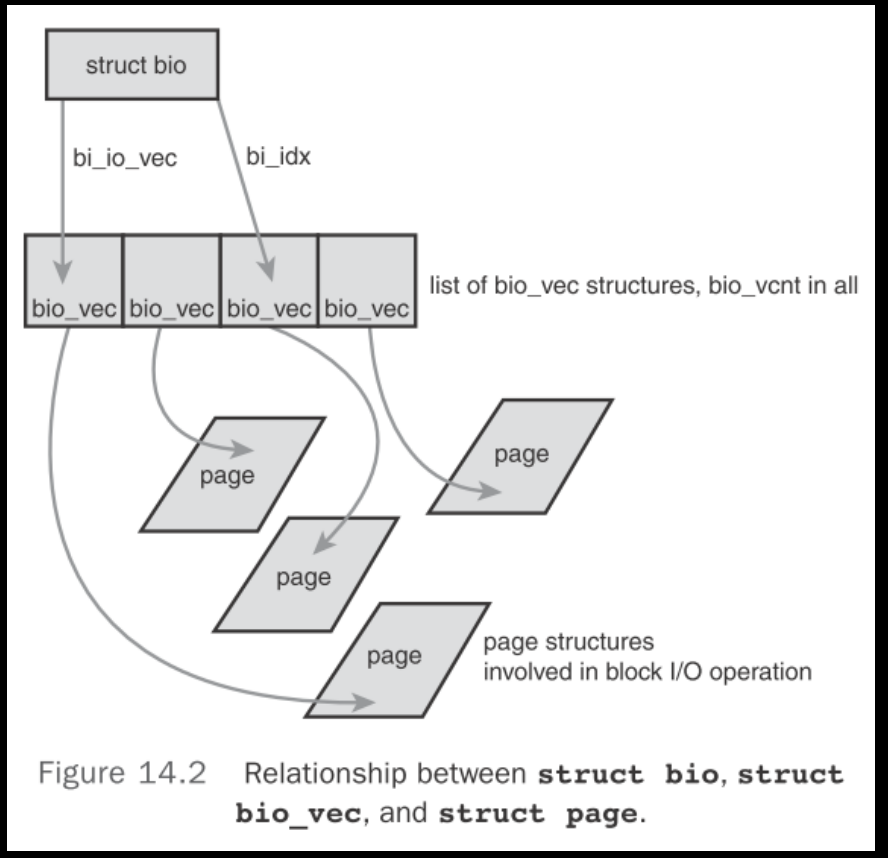


Figure 14.2 Relationship between **struct bio**, **struct bio_vec**, and **struct page**.

- **Benefits of bio struct**
  - (1) easily represent high memory (2) can represent both normal page and direct I/O (3) easy to use scatter-gather (vectored) block I/O operations (4) lightweight

- **Request Queues**
  - Block devices maintain request queues to store their pending block I/O requests
  - The structure request_queue is defined in <linux/blkdev.h>

I/O Schedulers

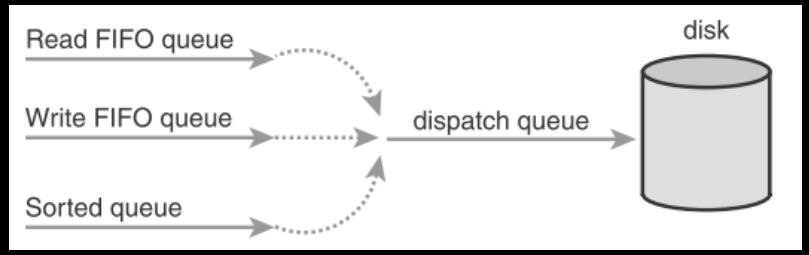
- Disk I/O is one of the slowest operation in computer
- Merging & Sorting optimization

4	2	1	3	2	1	4	3

4	4	1	1	2	2	3	3
1	1	2	2	3	3	4	4

  - Merge: combine multiple request of same block I/O
  - Sort: To make disk head move in one direction
- The Linus Elevator `<block/elevator.c>`
  1. Merge old and new request if they have same disk I/O section
  2. If a request in the queue is too old, the new insert to tail, prevent starvation
  3. If a suitable location to insert, insert there, maintain order. If not, insert to tail
- 1. Deadline I/O Scheduler `<block/deadline-iosched.c.>`

  Prevent write-starving-read problem (application need to wait for read operation)
  - Read latency is important to the performance of system, Read expire = 500 ms, Write expire = 5ms
  - Reducing request starvation comes at a cost to global throughput
  1. If both read and write, elevator chooses the “read” direction
  2. checks the deadline queue with direction, if time expire, enqueue dispatch queue; if not, sorted queue



- 2. The Anticipatory(預測) I/O Scheduler `<block/as-iosched.c>`
  - Originated from Deadline, aims to provide good read latency, while maintain good global throughput.
  - Add *anticipation heuristic*
  - Performs well across most workloads

- 3. The Complete Fair Queuing I/O Scheduler `<block/cfq-iosched.c>`
  - designed for *specialized workloads*
  - Every process's I/O has it own queue (merge & sort)
  - Use time slice RR queue, plucking requests from each queue before continue next round
  - Assuring that each process receives a fair slice of the disk's bandwidth

- 4. The Noop I/O Scheduler `<block/noop-iosched.c>`
  - No sorting & seek-prevention anymore, only merging
  - Use in random-Access device

Device Driver & Module

Intro of Device

- Three types of device: Block(blkdev), Character(cdev), Network(Ethernet)

Intro of Module

- Linux is “monolithic”, however, the kernel is modular
  - Support dynamic insertion and removal of code from itself at runtime
  - All data, entry, exit point are group in a binary image file, so called Module

- Difference between kernel module & Application:
  - Module is initialize at first, then driven by event
  - Need to manually release all initialized things when exit
  - error handler routine: kernel will kill current process
  - A module runs in kernel space, while applications run in user space

- Structure of module program
  - Entry: `int hello_init(void)`
    - `printk()`, operate like standard C, but can use without C library, use in kernel.
    - registered by `module_init()`, macro
      return 0 after initialization is complete
    - register resources, initialize hardware, allocate data structures…
    - If file is compile statically into kernel image
      Init function will store in the kernel image and run on kernel boot.
  - Exit: `void hello_exit(void)`
    - registered by `module_exit()`
  - `MODULE_LICENSE()`
    - “GPL” : GNU General Public License
    - “Proprietary” : Private(default)

```c
// hello.c – The Hello, World! Kernel Module
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

/*
* hello_init – the init function, called when the module is loaded.
* Returns zero if successfully loaded, nonzero otherwise.
*/
static int hello_init(void)
{
    printk(KERN_ALERT “I bear a charmed life.\n”);
    return 0;
}
/*
* hello_exit – the exit function, called when the module is removed.
*/
static void hello_exit(void)
{
    printk(KERN_ALERT “Out, out, brief candle!\n”);
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE(“GPL”);
MODULE_AUTHOR(“Shakespeare”);
MODULE_DESCRIPTION(“A Hello, World Module”);
```

- Build Kernel
  - kbuild build system
    - Leave code in the Linux kernel Source Tree
      - put under /drivers/block/, or /driver/char/…etc, or even a new directory
      - add
        ```
        obj-{CONFIG} := xxx.o                          // xxx.ko
        xxx-objs := xxx-main.o xxx-line.o
        ```
        in the makefile under device/…/Makefile
    - Leave the code externally
      - build a makefile in source code directory,
        ```
        obj-{CONFIG} := xxx.o                          // complie xxx.c to xxx.ko
        xxx-objs := xxx-main.o xxx-line.o              // xxx-main.c and xxx-lick.c link to .ko
        ```
      - Also need to tell how to find the kernel source files and base Makefile
        ```
        make -C /kernel/source/location SUBDIRS=$PWD modules
        ```
- Install Module
  - Compiled modules are installed into /lib/modules/version/kernel/driver/type/…

    `make modules_install` need to run as root
- Dependencies
  - Most Linux distributions generate the mapping automatically after power on

    `depmod` in root

- Load/ remove Module
  - `insmod module.ko / rmmod module` `// superuser`
  - for more intelligent, use `modprobe module [ module parameters ]`

- Module Parameters
  - enabling drivers to declare parameters :

```
static int allow_live_bait = 1;              /* default to on, may be global var*/
module_param(allow_live_bait, bool, 0644);  /* a Boolean type */
```

- Exported Symbols
  - When modules are loaded, they are dynamically linked into the kernel
  - After the function is declared, follow by `EXPORT_SYMBOL()`
  - ex:

```
/*
* get_pirate_beard_color - return the color of the current pirate's beard.
* @pirate is a pointer to a pirate structure
* the color is defined in <linux/beard_colors.h>.
*/
int get_pirate_beard_color(struct pirate *p)
{
    return p->beard.color;
}
EXPORT_SYMBOL(get_pirate_beard_color);
```

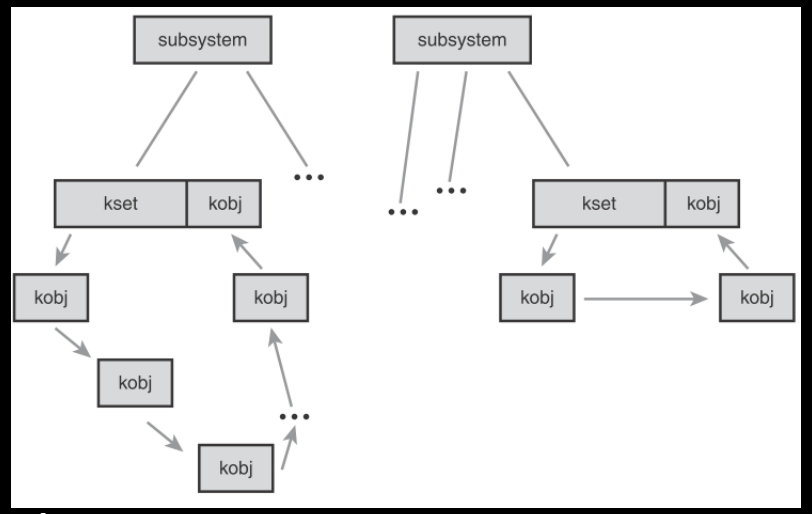
The Device Model

- Unified device model : code reuse, reference count, device tree traverse
  - `kobject,` defined in `<linux/kobject>, struct kobject`

    kobject add object properties, ex: reference count, parent, name…
  - `ktype,` defined in `<linux/kobject>, struct kobject_type`

    describe default behavior and attribute of a family of kobjects.
  - `kset,` defined in `<linux/kobject>, struct kset`

    collection of kobject, just like a container



- Relationship between kobj, ktype, kset
  - kobjects are usually embedded in other structures
  - embedded kobject acts as a base class for a group of kobjects
  - ksets aggregate together related kobjects
  - kobjects with identical ktypes may grouped into different ksets

Manipulate with kobject

- Driver writers do not have to deal with kobjects directly
- Declaration & Initialization

```
struct kobject *kobj;
kobj = kmalloc(sizeof (*kobj), GFP_KERNEL);
if (!kobj)
    return -ENOMEM;
memset(kobj, 0, sizeof (*kobj));
kobj→kset = my_kset;
kobject_init(kobj, my_ktype);
```

or

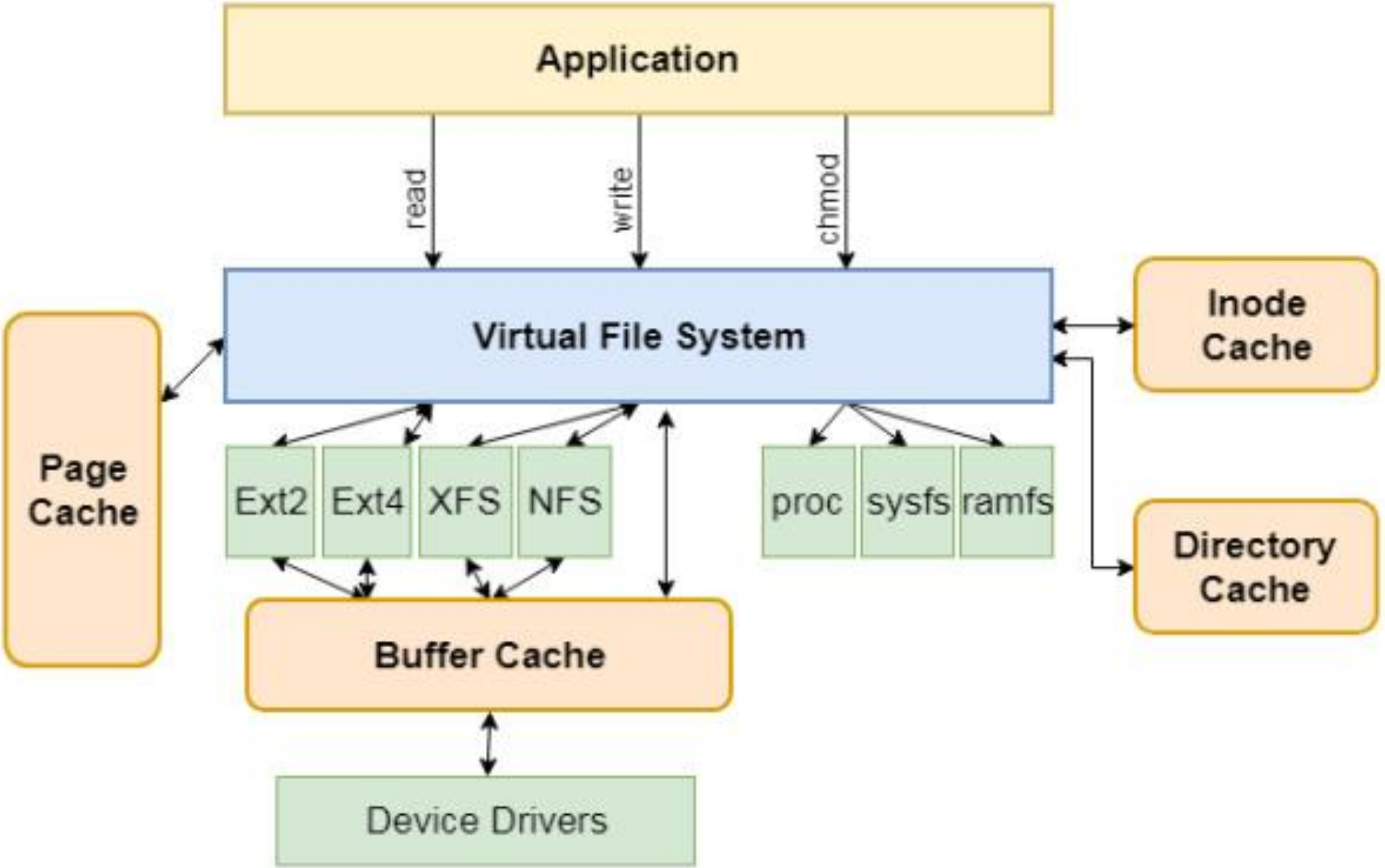
```
struct kobject *kobj;

kobj = kobject_create();
if (!kobj)
    return -ENOMEM;
```

Reference Count

- If reference count is nonzero, the object exist in memory (pinned)
- defined in <lib/kref.c>
  - `kobject_get(struct kobject *kobj); // reference count ++`
  - `kobject_put(struct kobject *kobj); // reference count --`
    - kref structure
    - kref_init()
    - kref_get()
    - kref_put()

sysfs

- Psudo FS, mount under /sys. Provide hierarchical view of kobject for userspace to access kernel data structure
- Adding and Removing kobjects from sysfs
  - `int kobject_add(*kobj, *parent, *fmt, ...);`
  - `struct kobject *kobject_create_and_add(*name, *parent);`
  - `void kobject_del(*kobj);`
- Adding Files to sysfs
  - Default Attribute
    - struct attribute : *name, *owner, mode
  - Create & Destory Attributes
    - `int sysfs_create_file(*kobj, *attr);`
    - `int sysfs_create_link(*kobj, *target, *name);`
    - `void sysfs_remove_file(*kobj, *attr);`
    - `void sysfs_remove_link(*kobj, *name);`
  - sysfs Conventions


Application
read
write
chmod
Virtual File System
Page Cache
Inode Cache
Directory Cache
Ext2
Ext4
XFS
NFS
proc
sysfs
ramfs
Buffer Cache
Device Drivers