Overview

- This Lecture
  - Character devices
  - Source: ULK ch12, ch13, ELDD ch5

Device drivers

- Device drivers are divided into different classes in Linux
  - Char
  - Block
  - Network
  - tty
- Different interfaces are provided by Linux for each class of device drivers
- Drivers are uniquely identified by major and minor numbers
  - Use *ls -l /dev* to find out information of each devices
  - Consult *Documentation/devices.txt* under linux src

File abstraction

- Each file has a name (path) in its directory
  - dentry
- Each file has an inode pointed by dentry
- A file system has a superblock
  - A superblock has maps for data blks and inodes
- When a file is opened, the kernel uses a *file* data structure to hold info of the file
  - *f_op*, *f_dentry*
  - The file abstraction is just a wrapper. File operations like read/write are delegated to *f_op*

```
struct file {
union {
        struct list_head        fu_list;
        struct rcu_head         fu_rcuhead;
    } f_u;
    struct path             f_path;
#define f_dentry        f_path.dentry
#define f_vfsmnt        f_path.mnt
    const struct file_operations    *f_op;
    atomic_long_t           f_count;
    unsigned int            f_flags;
    fmode_t                 f_mode;
    loff_t                  f_pos;
    struct fown_struct      f_owner;
    const struct cred       *f_cred;
    struct file_ra_state    f_ra;
 u64                    f_version;
#ifdef CONFIG_SECURITY
    void                    *f_security;
#endif
 /* needed for tty driver, and maybe others */
    void                    *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c  */
    struct list_head        f_ep_links;
    spinlock_t              f_ep_lock;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space    *f_mapping;
#ifdef CONFIG_DEBUG_WRITECOUNT
    unsigned long f_mnt_write_state;
#endif
};
```

Registration of drivers

- A driver needs to be registered with the following function
  - int register_chrdev_region(dev_t first,unsigned int count, char *name);
- If the major number of the device is not known, Linux can dynamically allocate one with the function
  - int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
- Auxiliary macros
  - MKDEV(int major, int minor); MAJOR(dev_t dev); MINOR(dev_t dev);

How to know the major number?

- When a major number is dynamically allocated, we need to find out the allocated number
  - From */proc/devices* the device name and major number will be found after the device driver get registered with alloc_chrdev_region
  - Note that a driver module should always remember to free the major number using unregister_chrdev_region when it is unloaded. The nodes created under */dev* should be removed as well.
- A script can be created to automatically find out the major number, and then make the device nodes under */dev* using *mknod*
- Alternatively you can use */sys* to register your device and allow *udev* daemon to generate an entry for you under */dev*, as in our *temp* module.

File operations

- A set of file operations is associated with a char device driver
  - ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  - Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with -EINVAL (“Invalid argument”). A nonnegative return value represents the number of bytes successfully read(the return value is a “signed size” type, usually the native integer type for the target platform).

File operations (cont.)

- ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
- Sends data to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if nonnegative, represents the number of bytes successfully written.
- int (*open) (struct inode *, struct file *);
- Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is NULL, opening the device always succeeds, but your driver is not notified.

File operations (cont.)

- int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
- The ioctl system call offers a way to issue device-specific commands(such as formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few ioctl commands are recognized by the kernel without referring to the fops table. If the device doesn't provide an ioctl method, the system call returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl for device").
- int (*release) (struct inode *, struct file *);
- This operation is invoked when the file structure is being released. Like open, release can be NULL.

File operations (cont.)

- There are more functions in the structure *file_operations*, but we are interested in the above five functions in our char device driver
  - Refer to ULK ch12 pp.473-474 for more details
- A device driver may not need to provide all functions defined in the file operations

The file structure

- When a function in file operations is called, a pointer to a file structure may be provided as an argument
- The file structure represents an opened file
  - Created when *open* is called at user space
  - Released when the last *close* is called
- The important fields (refer to ULK pp.471)
  - f_mode, f_pos, f_flag, f_op, private_data, f_dentry (for inode structure)

The inode structure

- The inode structure contains a great deal of information about a file
- For device drivers, only two fields are of interest
  - dev_t i_rdev;
    - For inodes that represent device files, this field contains the actual device number.
  - struct cdev *i_cdev;
    - struct cdev is the kernel's internal structure that represents char devices; this field contains a pointer to that structure when the inode refers to a char device file.
- Two macros are provided to get device numbers.
  - unsigned int iminor(struct inode *inode);
  - unsigned int imajor(struct inode *inode);

```
struct inode {
        struct hlist_node       i_hash;
        struct list_head        i_list;
        struct list_head        i_sb_list;
        struct list_head        i_dentry;
        unsigned long           i_ino;
        atomic_t                i_count;
        unsigned int            i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        dev_t                   i_rdev;
        u64                     i_version;
        loff_t                  i_size;
#ifdef __NEED_I_SIZE_ORDERED
        seqcount_t              i_size_seqcount;
#endif
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        unsigned int            i_blkbits;
        blkcnt_t                i_blocks;
        unsigned short          i_bytes;
        umode_t                 i_mode;
        spinlock_t              i_lock;
        struct mutex            i_mutex;
        struct rw_semaphore     i_alloc_sem;
        const struct inode_operations   *i_op;
        const struct file_operations    *i_fop;
        struct super_block      *i_sb;
        struct file_lock        *i_flock;
        struct address_space    *i_mapping;
        struct address_space    i_data;
        struct list_head        i_devices;
        union {
                struct pipe_inode_info  *i_pipe;
                struct block_device     *i_bdev;
                struct cdev             *i_cdev;
        };
        int                     i_cindex;
        __u32                   i_generation;
        unsigned long           i_state;
        unsigned long           dirtied_when;
        unsigned int            i_flags;
        atomic_t                i_writecount;
        void                    *i_private;
}
```

Char device internal structure

- An internal structure is used to represent a char device
  - The structure can be allocated and initialized with
    - struct cdev *my_cdev = cdev_alloc();
    - my_cdev->ops = &my_fops;
  - Or initialized with
    - void cdev_init(struct cdev *cdev, struct file_operations *fops);
  - And initialize the owner field
    - my_cdev->cdev.owner = THIS_MODULE

Char device registration

- Once the internal structure is set up, the device should be registered to the kernel with
  - int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
- When the module is unloaded, the device should be de-registered with
  - void cdev_del(struct cdev *dev);

open

- *open* should perform the following tasks
  - Check which device (minor number) is being opened
  - Check for device-specific errors (such as device-not-ready or similar hardware problems)
  - Initialize the device if it is being opened for the first time
  - Update the f_op pointer, if necessary
  - Allocate and fill any data structure to be put in *filp->private_data*
  - How to know a file is open write-only or read-only?
    - if((filp->f_flags & O_ACCMODE) == O_WRONLY)
    - if((filp->f_flags & O_ACCMODE) == O_RDONLY)

release

- *release* should perform the following tasks
  - Deallocate anything that open allocated in *filp->private_data*
  - Shut down the device on last close

read and write

- The *read* and *write* methods both perform a similar task, that is, copying data from and to application buffer.
  - ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t *offp);
  - ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp);
  - For both methods, *filp* is the file pointer and *count* is the size of the requested data transfer. The *buff* argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. Finally, *offp* is a pointer to a ”long offset type” object that is the current file position. The return value is a “signed size type”

Copying between user and kernel

- Two functions are frequently used by device drivers
  - unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);
  - unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);
- Why not directly copy data to/from user space?

Return value of read

- The return value is interpreted as below by the application program
  - If the value equals the count argument passed to the read system call, the requested number of bytes has been transferred. This is the optimal case.
  - If the value is positive, but smaller than count, only part of the data has been transferred. This may happen for a number of reasons, depending on the device. Most often, the application program retries the read.
  - If the value is 0, end-of-file was reached (and no data was read).
  - A negative value means there was an error. The value specifies what the error was (refer to <linux/errno.h>).

Return value of write

- The return value is interpreted as below by the application program
  - If the value equals count, the requested number of bytes has been transferred.
  - If the value is positive, but smaller than count, only part of the data has been transferred. The program will most likely retry writing the rest of the data.
  - If the value is 0, nothing was written. This result is not an error, and there is no reason to return an error code. Once again, the standard library retries the call to write.
  - A negative value means there was an error. The value specifies what the error was (refer to <linux/errno.h>).

OS organization

- There are many ways to structure an OS
  - Monolithic
  - Microkernel
  - Exokernel
- OS approach for protection and isolation
  - Virtualize resources (CPU, memory)
    - Simulate a dedicated CPU and memory space
  - Abstract other resources (storage, network)
    - A sharable abstraction layer over hardware

Monolithic

- Traditional approach used in Linux/Unix
  - More composable using loadable modules
  - Successful approach
- Three layers
  - H/W, kernel, user
- Kernel is a big program
  - Process management, MM, FS, network, I/O
  - With full access privilege over H/W
- Pros: fast, subsystems cooperate easily, convenient (for hiding or exposing functions)
- Cons: complex, buggy, no isolation

Microkernel

- Keep the kernel small
  - Many traditional kernel services such as VM and FS become user-space server processes
  - Philosophy: use IPC and user-space servers to split OS subsystems
    - For any new function of OS, make a new server and talk to it with IPC
- Four layers
  - H/W, kernel, server processes, applications
  - Servers: VM, FS, TCP/IP, Print, Display

Microkernel (cont.)

- How it works?
  - Servers have privileged access to some H/W
  - Applications request service using IPC
  - Kernel's main job is to provide fast/secure IPC
- Pros
  - Simple/efficient kernel, sub-systems isolated, enforced modularity
- Cons
  - Harder cross-sub-system optimization, lots of IPC overheads may slowdown overall OS

Exokernel

- Philosophy
  - Eliminate all abstractions, let applications do what it wants
- System consists of
  - H/W, kernel, environments, libOS, app.
- How it works?
  - Do not provide addr. space, virt. cpu, FS, TCP
  - Instead, app has control over phys pages, addr mappings, clock interrupts, disk, net
  - Let app build nice address space if it wants
  - Give aggressive apps much more flexibility

Exokernel (cont.)

- Challenges
  - How to share cpu, mem, etc if they are exposed directly to apps?
  - How to implement security/isolation despite apps having low-level control?
  - How to multiplex w/o understanding e.g. disk (file system), TCP/IP packets?

Exokernel (cont.)

- Example: memory
  - Resources are phy. pages, va->pa mappings
  - App. ask kernel with the following API
    - pa = AllocPage()
    - DeallocPage(pa)
    - TLBwr(va, pa)
  - And the kernel-app upcalls
    - PageFault(va)
    - PleaseReleaseAPage()

Exokernel (cont.)

- How to use phy pages fairly and protect the app memory?
  - Ensure app only creates mappings to phys pages it owns
  - Track what environment owns what phys pages
  - Decide which app to ask to give up a phys page when system runs out
  - App has to decide which of its pages to give away
  - Is fair sharing guaranteed?

Exokernel (cont.)

- Example: shared memory
  - Two processes want to share memory
    - Process a:
      - pa = AllocPage()
      - put 0x5000 -> pa in private page table
      - PageFault(0x5000) upcall -> TLBwr(0x5000, pa)
      - give pa to process b via exokernel
    - process b:
      - put 0x6000 -> pa in private page table
  - Note that app is not allowed to directly do this in traditional “virtual address space”

Exokernel (cont.)

- Cool example in database app
  - The problem on traditional OS:
    - Assume an OS with demand-paging to/from disk
    - if DB caches some disk data, and OS needs a phys page, OS may page-out a DB page holding a cached disk block
    - But that's a waste of time: if DB knew, it could release phys page without writing, and later read it back from DB file (not swapping area)

Exokernel (cont.)

- Cool example in database app on exokernel
  - If exokernel needs phys memory for some other application, exokernel sends the DB a PleaseReleaseAPage() upcall
  - DB picks a clean page, calls DeallocPage(pa)
  - Or DB picks a dirty page, writes to disk, then DeallocPage(pa)
  - If it is a clean page, it saves the time to write to disk, which has to be done in traditional OS with swapping.

Exokernel (cont.)

- Example: CPU
  - How to expose CPU to app?
    - Kernel tells app when it is taking away CPU
    - Kernel tells app when it gives CPU to app
    - If app is running and timer interrupt causes end of slice, CPU jumps from app into kernel
    - Kernel then jumps back into app via PleaseYield() upcall
    - App saves state (registers, EIP, etc)
    - App calls Yield() to agree to yield
    - When kernel decides to resume app, kernel jumps into app at Resume() upcall
    - App restores those saved registers and EIP

Exokernel (cont.)

- Cool example with exokernel CPU management
  - Suppose time quantum ends in the middle of
    - Lock();
    - …
    - Unlock();
  - If the app holds the lock while suspended, maybe other apps can't make any progress
  - Luckily the PleaseYield() upcall can be used to complete the critical section if needed

Exokernel (cont.)

- Example: fast RPC (remote procedure call) with direct CPU management
  - On traditional OS, pipes or sockets are used
    - write(), read(), read(), write() are used to cost at least 8 kernel/user crossings
  - On exokernel, only 4 kernel/user crossings
    - Yield() can take a target process argument
    - Almost a direct jump to an instruction in target process
    - Kernel allows entries at approved locations in target
    - Kernel leaves regs alone, so can contain arguments
    - Target app uses Yield() to return results