Digital Systems L2 – Computer Systems

Fang Yuan

School of Electronics Science and Engineering
Nanjing University
Nanjing 210046

2022

Outline

1. DEVICE DRIVERS
   - Introduction to Device Drivers
   - Building and 'Running' a Module
   - Character Device Drivers
   - Debugging Techniques
   - Kernel Timer

Role of Device Drivers

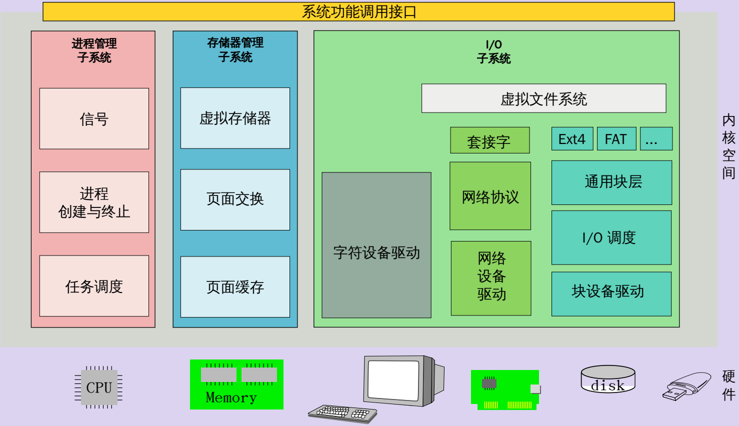
All devices work under the control of certain softwares. These softwares are device drivers. In Linux, drivers can be part of kernel image, or independent code. This code is also called a module.

A flexible driver provides mechanism rather than policy.

- Mechanism What capabilities are to be provided
- Policy How those capabilities can be used

Splitting the kernel

Split View of the kernel



Functions of the OS

- Process management
  - Creating, destroying and scheduling processes
  - Handling the Communication among different processes
  - Handling their connection to the outside world
- Memory management
  Virtual addressing space for all processes on top of the limited available resources.
- File systems
  Structured filesystem on top of unstructured hardware
- Device control
  Device drivers operate all physical devices.
- Networking

Types of Devices and Modules

Modules are part of kernel codes that can be added to or removed from dynamically.

- Character devices
  - A stream of bytes
  - Accessed by means of filesystem nodes
- Block devices – can host a filesystem
- Network interfaces – sending and receiving data packets

Comments to the First Module

```
#include <linux/module.h> // head files for kernel

int init_module(void)    // called when insmod
{
    printk(KERN_EMERG "Hello\n");
        // kernel version of printf
    return 0;
}

void cleanup_module(void) // called when rmmod
{
    printk(KERN_EMERG "module␣removed\n");
}
```

Compiling and Loading a Module

```
# Makefile (!not makefile)
ifneq ($(KERNELRELEASE),)
obj-m          := hello.o
else
KDIR           := /lib/modules/`uname -r`/build
PWD            := $(shell pwd)
default:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
endif
```

Some Macros Unsed in Modules

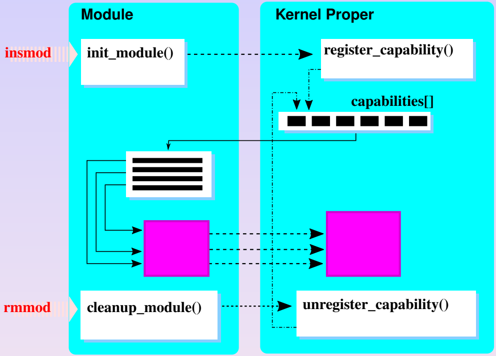
- replace `init_module()` and `cleanup_module()`:

```
#include <linux/module.h>
module_init(user_init_func)
module_exit(user_cleanup_func)
```

- Extra infos

```
MODULE_LICENSE("GPL/LGPL/private")
MODULE_AUTHOR("zhang3 <zhang3@gmail.com>");
MODULE_DESCRIPTION("A simple hello world");
MODULE_VERSION("0:0.99");
```

Kernel Modules vs. Applications



An application performs a single task from beginning to end, whereas a module registers itself in order to serve future requests, and its "main" function(init_module) terminates immediately.

Register Resources

I/O ports

```
int check_region(unsigned long start,
    unsigned long len);
struct resource *request_region(
    unsigned long start,
    unsigned long len, char *name);
void release_region(unsigned long start,
    unsigned long len);
```

Passing Arguments Kernel Module

In command line:
insmod hello.ko val=3600

In module code:

```
#include <linux/moduleparam.h>
...
module_param(val, int, 0666);
```

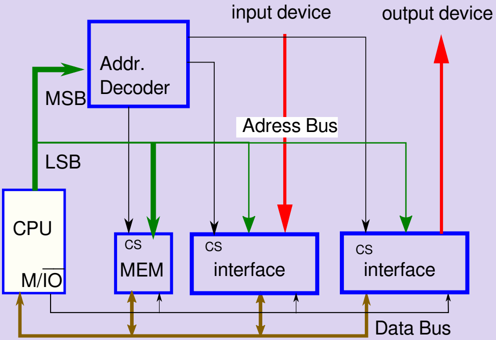
available variable types: byte, short, ushort, int, uint, long, ulong, charp, bool, invbool.

permission types:

S_IWUSR S_IRGRP
S_IRUSR S_IWGRP
S_IXUSR S_IXGRP

Different Memmory Spaces

Memory-mapped IO and Port-mapped IO



CISC computers use different instructions (produce $M/\overline{IO}$) to access memory or IO.

Register Resources

I/O memory

```
int check_mem_region(unsigned long start,
    unsigned long len);
int request_mem_region(unsigned long start,
    unsigned long len, char *name);
int release_mem_region(unsigned long start,
    unsigned long len);
```

Register a Device

Register

```c
int register_chrdev(unsigned int major,
    const char *name,
    struct file_operations *fops);
```

- major: a specified major number, or dynamic allocation(major=0)
- name:device filename, should be created by mknod
- fops: a global pointer to the struct file_operations

Major and minor numbers

- Char devices are accessed through names in the filesystem.
- The major number identifies the driver associated with the device.
- The minor number is used only by the driver specified by the major number.
- (with or without devfs)

File Operations

Each field in the structure file_operations must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations. An example:

```
struct file_operations scull_fops = {
        lseek: scull_llseek,
        read: scull_read,
        write: scull_write,
        ioctl: scull_ioctl,
        open: scull_open,
        release: scull_release,
};
```

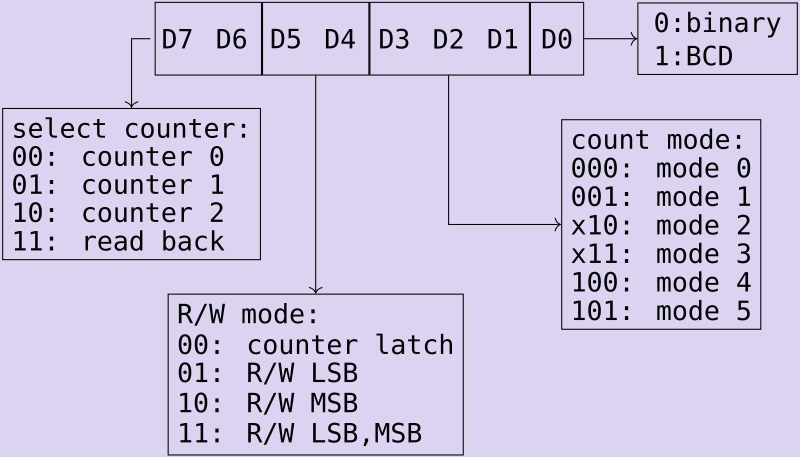
What Happens When File is Opening

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

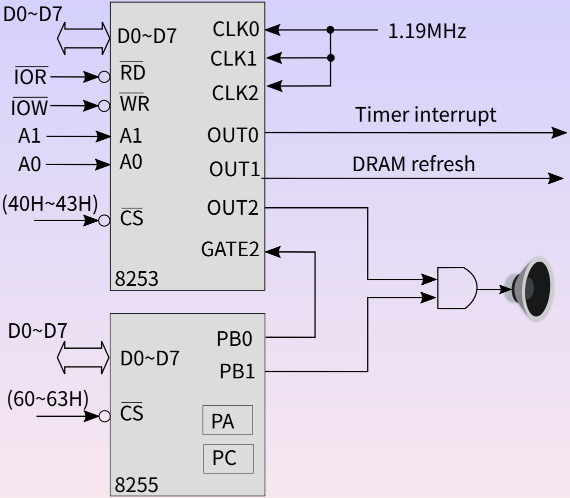
If this entry is NULL, open() is always succeeds without notified.
Device number `node->i_rdev` in include/linux/kdev_t.h

```
struct file {
    ......
    struct path       f_path;
    unsigned int      f_flags;
    fmode_t           f_mode;
    loff_t            f_pos;
    void              *private_data;
    ......
}
```

Intel8254 Control word

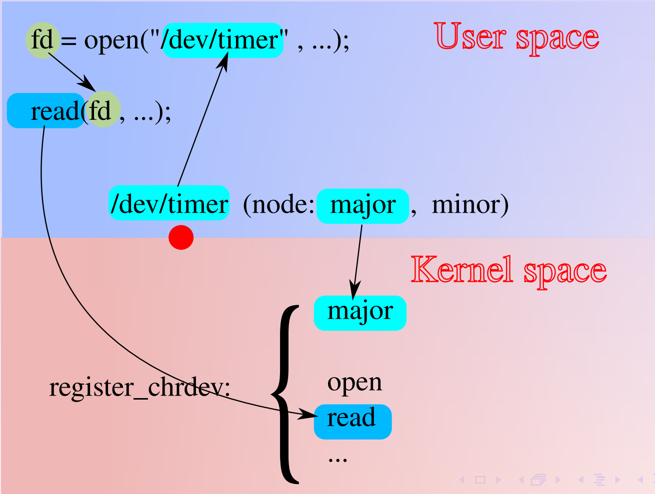


Speaker in PC



Device Node

A device node connects between user space and kernel.



Unregister a Device

Removing a driver from the system

Unregister

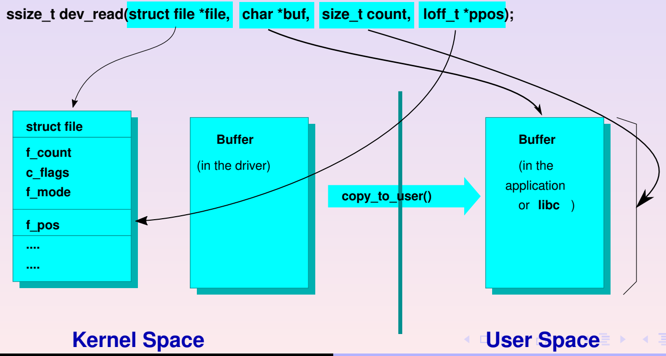
```
void unregister_chrdev(unsigned int major,
    const char *name);
```

You also need to remove the device files.
Create and remove device node manually, or using udev.

Memory Usage

```
unsigned long copy_to_user(void *to,
    const void *from, unsigned long count);

unsigned long copy_from_user(void *to,
    const void *from, unsigned long count);
```



ioctl

Special Operations on I/O Devices

- Most devices can perform operations beyond simple data transfers
- User space must often be able to request operations other than read/write.
- These operations are usually supported via the ioctl (I/O Control) method.

I/O Control Methods

About _IO macros (linux/ioctl.h)

- _IOC_TYPEBITS, 8 bits, just choose a number(magic number)
- _IOC_NRBITS, 8 bits, sequential number
- _IOC_DIRBITS, 2 bits

_IO	an ioctl with no parameters
_IOW	an ioctl with write parameters
_IOR	an ioctl with read parameters
_IOWR	an ioctl with both w/r parameters

- _IOC_SIZEBITS, 14 bits

I/O Memory Address

Unlike X86, RISC machines have memory mapped I/O. I/O addresses of memory must be converted into virtual address before use.

```
#include <linux/io.h>
void *ioremap(unsigned long phys_addr,
    unsigned long size);

void iounmap(void * addr);
```

Block I/O

Read/write I/O device may block when data is unavailable. The process must hang up in order to do other tasks until data transferring is available, then wake up to finish read/write.

```
#include <linux/wait.h>

DECLARE_WAIT_QUEUE_HEAD(wq);
...
wait_event_interruptible(wait_queue_head_t queue,
    int condition);
...
void wake_up_interruptible(wait_queue_head_t *queue
```

PROCFS

The /proc is a special, software-created filesystem that is used by the kernel to export information to user space.

Create proc file

```
#include <linux/proc_fs.h>
struct proc_dir_entry *proc_create(
            const char *name,
            umode_t mode,
            struct proc_dir_entry *parent,
            const struct file_operations *proc_fops
```

The default proc file `*name` is under /proc when `*parent` is NULL.

PROCFS

Each file under /proc is tied to a kernel function that generates the file's "contents" on-the-fly when the file is read.

Remove proc file

```
void remove_proc_entry(const char *,
                       struct proc_dir_entry *);
```

Interrupt handling

Register and unregister an interrupt handler:

- ```
  int request_irq(unsigned int irq
        irqreturn_t (*handler)(int , void *,
  struct pt_regs*)
        unsigned long flags,
        const char *dev_name,
        void *dev_id);
  ```
- `void free_irq(int irq, void *dev_id);`
  ```

Implementing a Handler

- 'Fast' interrupt (SA_INTERRUPT) handlers are executed with interrupts disabled on the current processor.
- SA_SHIRQ signals that the interrupt can be shared between devices.
- Interrupt handlers need to finish up quickly and not keep interrupts blocked for long. The interrupt handler is splitted into two halves:
  - The top half is the route that actually responds to the interrupt.
  - The bottom half is scheduled by the top half to be executed later at a safer time, with all interrupts enabled.

Printing

printk loglevel (linux/kernel.h)

- KERN_EMERG: emergency messages(crash)
- KERN_ALERT: requiring immediate action
- KERN_CRIT: critical conditions(serious hardware or software failures)
- KERN_ERR: error conditions
- KERN_WARNING: warnings
- KERN_NOTICE: security-related conditions
- KERN_INFO: informational messages
- KERN_DEBUG: debugging messages.

printk slows down the system noticeably, as printing causes a disk operation.

Debuggers and Related Tools

- gdb
  gdb /usr/src/linux/vmlinux /proc/kcore
  compiling the kernel with debugging support (-g) results a huge vmlinux file.
- kdb – The kernel debugger
- The integrated kernel debugger patch
- The kgdb patch
- Kernel crash dump analyzers
- The user-mode Linux port
- The Linux trace toolkit
- Dynamic probes

Timing

- Understanding kernel timing
  Timer interrupt, which is the mechanism the kernel uses to keep track of time intervals. The interval of timer interrupts is set by the value HZ.
- Using the jiffies counter(modified during timer interrupt)
- Processor-specific registers, rdtsc(low32, high32) (x86)
- Delaying operation for a specified amount of time: udelay(), mdelay()
- Scheduling asynchronous functions to happen after a specified time lapse

Delay in Kernel

- udelay()、mdelay() and even ndelay() are software loop (busy-wait functions).
- while (jiffies < j) ;
- wait_event_timeout(jit_delay*HZ)

Knowing the Current Time

Calendar date and time

```
/* time in user space */
#include <sys/time.h>

int gettimeofday(struct timeval *tv,
        struct timezone *tz);
int settimeofday(const struct timeval *tv,
        const struct timezone *tz);

/* time in kernel space */
do_gettimeofday(struct timeval *tv);
```

Knowing the Current Time

Epoch Time

```
/* Number of seconds since
   1970-01-01 00:00:00 +0000 (UTC) */

time_t time(time_t *tloc);
```

Summary

- Kernel Layout
- Kernel Modules
- Char Device Drivers
- PROCFS
- Kernel Timing

Function List 1

```
#include <linux/module.h>

int init_module(void);
void cleanup_module(void);

module_init(x)
modile_exit(x)

MODULE_LICENSE(string)
MODULE_AUTHOR(string)
MODULE_DESCRIPTION(string)
MODULE_VERSION(string)
```

Function List 2

```
#include <linux/fs.h>

struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *,
    ssize_t (*write) (struct file *, const char __u
    unsigned int (*poll) (struct file *, struct pol
    long (*unlocked_ioctl) (struct file *, unsigned
    long (*compat_ioctl) (struct file *, unsigned i
    int (*mmap) (struct file *, struct vm_area_stru
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    ...;
};
```

Function List 3

```
#include <linux/fs.h>

int register_chrdev(int major, char *name,
                    struct file_operations *);

/* extract major and minor */
MAJOR (kdev_t dev);
MINOR (kdev_t dev);
MKDEV (int major, int minor);

void unregister_chrdev(int major, char *name);
```

Function List 4

```
#include <linux/slab.h>

void *kmalloc(size_t size, gfp_t flags);
void kfree(const void *);

#include <linux/types.h>
typedef enum {
    GFP_KERNEL,
    GFP_ATOMIC,
    __GFP_HIGHMEM,
    __GFP_HIGH
} gfp_t;
```

Function List 5

```c
#include <linux/uaccess.h>

unsigned long copy_to_user(void *to,
                           const void *from,
                           unsigned long n);
unsigned long copy_from_user(void *to,
                             const void *from,
                             unsigned long n);

#include <linux/io.h>
void *ioremap (unsigned long phys_addr,
               unsigned long size);
void iounmap(void *addr);
```

Function List 6

```
#include <linux/proc_fs.h>
struct proc_dir_entry *proc_create(
        const char *name,
        umode_t mode,
        struct proc_dir_entry *parent ,
        const struct file_operations *proc_fops);

void remove_proc_entry(
        const char *,
        struct proc_dir_entry *);
```

Function List 7

```
/* Platform Specific, may only on X86 available */
#include <linux/io.h>

unsigned char inb(unsigned long addr);
void outb(unsigned short value,
          unsigned long addr);
void insb(unsigned long addr,
          void *buffer,
          unsigned int count);
void outsb(unsigned long addr,
           const void *buffer,
           unsigned int count);
```

Suffix b, w, l specify 8bits, 16bits or 32bits port access. s for string I/O.

Function List 7

Userspace port mapped IO (priviledge required!)

```
#include <sys/io.h>

int ioperm(unsigned long from,
           unsigned long num,
           int turn_on);

unsigned char inb(unsigned long addr);
...
```