Software Layers

- User-level I/O software & libraries
- Device-independent OS software
- Device drivers
- Interrupt handlers
- Hardware

Operating system (kernel)

User
Device Drivers

User space

User program

Kernel space

Rest of the OS

Keyboard driver

Disk driver

Keyboard controller

Disk controller
Device Drivers in Linux

• Can be compiled into the kernel
  – Using static linking just like any binary

• Can be loaded dynamically as Modules
  – Using dynamic linking just like link loader
  – Except...
    • Modules are linked to kernel (not applications)
    • Modules are loaded into kernel space (not user space)
Device Drivers as Plug-in Modules
Hello World Module

#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init); //macro to specify ‘init’ function
module_exit(hello_exit); //macro to specify ‘exit’ function
Why printk?

- Modules do not have access to user libraries
  - Modules are linked to the kernel
  - Kernel executes by itself w/o relying on user libraries
- Can’t use printf and many other standard functions provided by C library (libc.so)
  - Printk is a kernel function similar to printf
  - But implemented very differently (w/o write system call)
- Kernel provides own versions for other common functions like strcpy, strcat, etc.
Building & Running

\% make

make[1]: Entering directory `/usr/src/linux-2.6.10'
   CC [M] /home/ldd3/src/misc-modules/hello.o
   Building modules, stage 2.
   [sic]
\% su

root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#

- insmod/rmmod calls module ‘init’/’exit’ functions
- Need root privileges to insmod/rmmod since you are modifying kernel
Module Helper Programs

- **insmod** – loads a module to memory
  - Dynamically links a module to kernel
  - Calls module ‘init’ function to initialize device
- **rmmod** – unloads a module from memory
  - Calls module ‘exit’ function to perform clean up
- **lsmod** – lists what modules are loaded
- Need root privileges since kernel is modified
Types of Devices in Linux

• **Block Devices**
  – Device that can host a file system by storing data in randomly accessible fixed-sized blocks
  – E.g. Disks, Flash Drives

• **Character Devices**
  – Device that delivers/accepts a stream of characters
  – E.g. Keyboard, mouse, terminal
Devfs and Device Files

- Devfs (mounted on /dev/) contains device files:
  ```shell
  thoth $ ls -l /proc/self/fd
  total 0
  lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 0 -> /dev/pts/0
  lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 1 -> /dev/pts/0
  lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 2 -> /dev/pts/0
  lr-x------ 1 wahn UNKNOWN1 64 Sep 11 13:37 3 -> /proc/16970/fd
  thoth $ ls -l /dev/pts/0
  crw------- 1 wahn tty 136, 7 Sep 11 11:37 /dev/pts/0
  ```
- All read/write system calls to stdin, stdout, stderr routed to pseudo terminal device file /dev/pts/0
- The “c” in “crw” in /dev/pts/0 permissions means it is a character device. If “b”, block device.
Sysfs and Configuration Files

- Sysfs (mounted on /sys/) contains configuration files
  
  ```bash
  thoth $ cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq
  1999998
  thoth $ cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_min_freq
  249999
  ```

- /sys/devices/system/cpu: all CPUs available to system
- Shows CPU 0 can operate between max frequency 1999998 KHz (2 GHz) and min frequency 249999 KHz (250 MHz)
How Device Drivers Interface with OS

- A well-defined interface for each type. E.g.
  - Block devices have an interface suitable for storage media
  - Char devices have an interface suitable for stream-oriented devices

- This interface consists of two parts:
  - Functions that the device driver implements (that the OS can call)
  - Functions that the OS implements (that the device driver can call)

- All device types have interfaces to:
  - Register device: OS calls device driver ‘init’ function on ‘insmod’
  - Unregister device: OS calls device driver ‘exit’ function on ‘rmmod’
  - Service OS requests made in response to user system calls
  - Service interrupts coming from HW device
Block Device Driver
Block Device Init Function

static void disk_request(struct request_queue *q) { ... }
static struct block_device_operations disk_fops = { ... };
static int __init disk_init(void)
{
    /* 1. Ask kernel to create queue for disk_request */
    struct request_queue *disk_queue = blk_init_queue(disk_request, &lock);
    /* 2. Ask kernel to allocate disk structure */
    struct gendisk *disk = alloc_disk(DISK_MINORS);
    /* 3. Register block device operations to disk */
    disk->fops = &disk_fops;
    /* 4. Attach request queue to disk */
    disk->queue = disk_queue;
    /* 5. Make disk available to the system */
    add_disk(disk);
}

• Red: Interface implemented by device driver
• Blue: Interface implemented by OS
disk_fops struct

struct block_device_operations {
    struct module *owner;
    int (*open)(struct inode*, struct file*);
    int (*ioctl)(struct inode*, struct file*, unsigned int, unsigned long);
    int (*release)(struct inode*, struct file*);
    ...
};

- Contains function pointers passed to OS when driver is loaded
- OS calls functions on corresponding system call
- Note: no entries for read and write system calls
  - Handled by disk_request () associated with the request queue
disk_request() function

static void disk_request(struct request_queue *q);

• OS calls disk_request() after enqueuing a batch of read/write requests from a series of read/write system calls

• Device driver dequeues requests from q and services them
  – Performance of many storage media is sensitive to order of processing requests (e.g. amount of arm movement in a hard disk)
  – Servicing batches of requests allows device driver to reorder them to best suit the characteristics of device
Character Device Operations

```c
struct file_operations {
    struct module *owner;
    int (*open)(struct inode*, struct file*);
    ssize_t (*read)(struct file*, char*, size_t, loff_t*);
    ssize_t (*write)(struct file*, const char*, size_t, loff_t*);
    int (*release)(struct inode*, struct file*);
    ...
};
```

- Note the presence of read and write system calls
  - Read/write system calls forwarded directly to driver
    (No need to batch together requests as in block devices)
- Different Device classes (block, char, ...) have different interfaces
User Space Drivers

• User space drivers: Device drivers where most of the code is run in user space
  – Small kernel module forwards all requests to a user program that implements the requests
  – Typically used for drivers that do not (or rarely) need to communicate with hardware

• FUSE – Filesystem in User Space
  – Useful for implementing virtual file systems (e.g. by communicating with cloud storage)
- **FUSE**

  - **hello:** User-written file system program using libfuse
  - FUSE kernel module forwards all requests to hello
User Space Drivers

• Advantages?
  – Full Standard C Library can be linked in
  – Can use a conventional debugger like GDB
  – Problems with driver will not crash entire system

• Disadvantages
  – Hard to deal with hardware interrupts efficiently
    • Needs to go through user/kernel boundary every time
    • Copying to shared memory, changing privilege etc.
Pitfall 1: Error Handling

```c
int __init my_init_function(void)
{
    int err;
    /* registration takes a pointer and a name */
    err = register_this(ptr1, "driver");
    if (err) goto fail_this;
    err = register_that(ptr2, "driver");
    if (err) goto fail_that;
    err = register_those(ptr3, "driver");
    if (err) goto fail_those;
    return 0; /* success */

    fail_those: unregister_that(ptr2, "driver");
    fail_that: unregister_this(ptr1, "driver");
    fail_this: return err; /* propagate the error */
}
```

• Always remember to clean up state before returning on error!
Pitfall 2: Race Conditions

• Race: situation where two tasks “race” to complete resulting in illegal interleaving of operations
  – Multiple applications on multiple CPUs may attempt to access your driver simultaneously
  – Your driver maybe in the middle of handling a system call when an interrupt happens
• Use synchronization appropriately! – we will talk about this on the lecture on synchronization