



Lecture 4

Loadable Kernel Modules

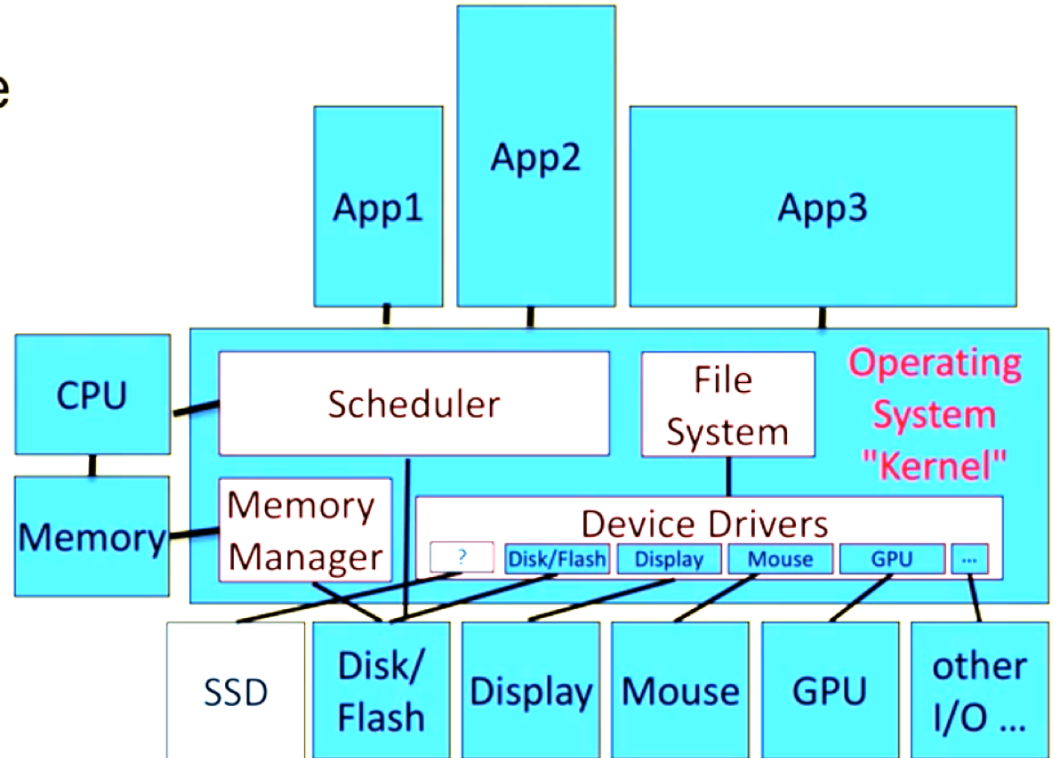
Device have both device-independent and device-dependent code

There is special device driver code associated with each different device connected to the system

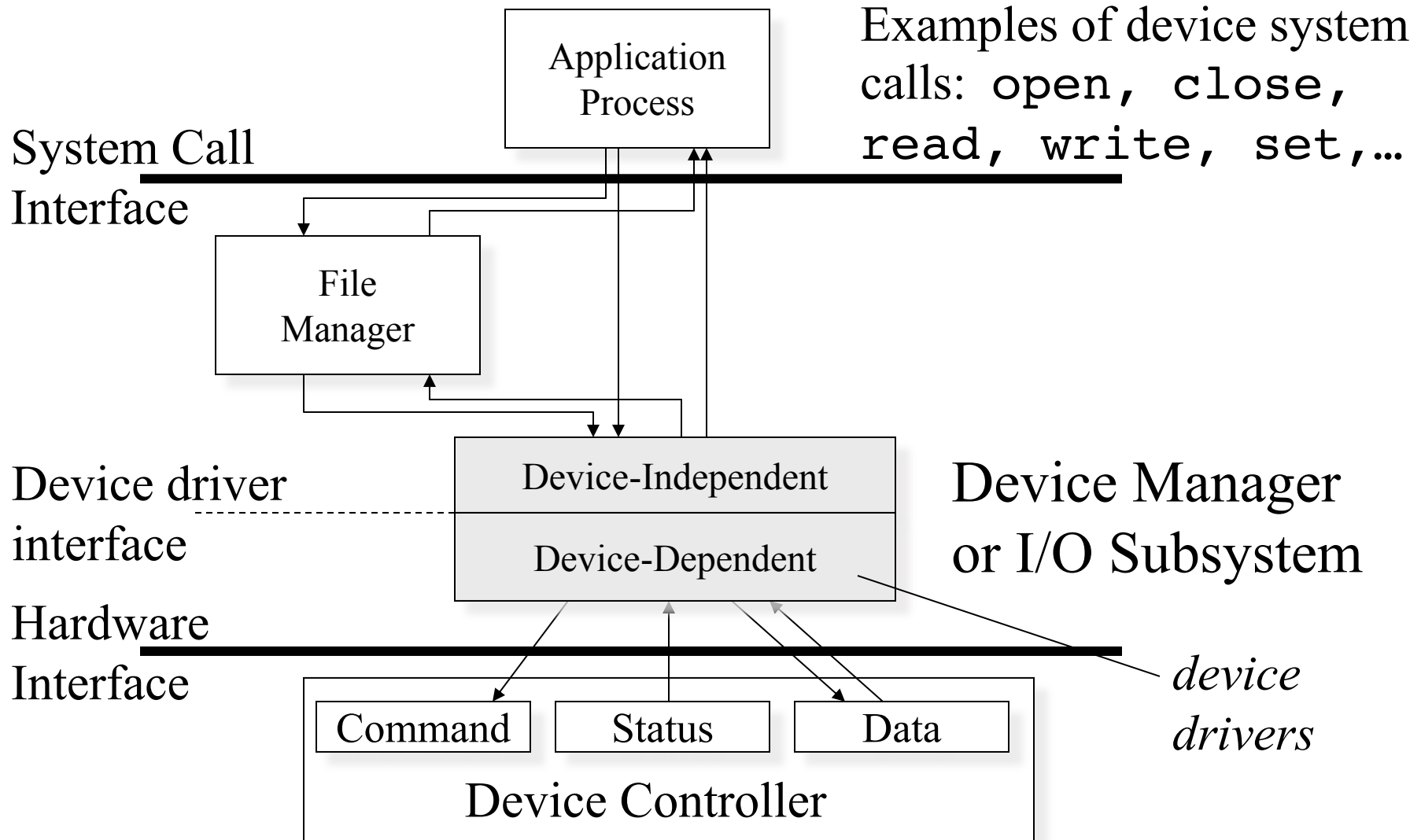
1. *Device-Independent* API

2. *Device-Dependent* driver code

3. *Device Controller*



Device Management Organization



Device Independent Part

- A set of system calls that an application program can use to invoke I/O operations
- A particular device will respond to only a subset of these system calls
 - A keyboard does not respond to *write()* system call
- POSIX set: *open()*, *close()*, *read()*, *write()*, *lseek()* and *ioctl()*

Device Independent Function Call

Trap Table

<code>func_i(...)</code>

```
dev_func_i(devID, ...) {  
    // Processing common to all devices  
    ...  
    switch(devID) {  
        case dev0: dev0_func_i(...);  
                    break;  
        case dev1: dev1_func_i(...);  
                    break;  
        ...  
        case devM: devM_func_i(...);  
                    break;  
    };  
    // Processing common to all devices  
    ...  
}
```



Adding a New Device

- Write device-specific functions for each I/O system call
- For each I/O system call, add a new *case* clause to the *switch* statement in device independent function call



Trap Table

<code>func_i(...)</code>

```
dev_func_i(devID, ...) {  
    // Processing common to all devices  
    ...  
    switch(devID) {  
        case dev0: dev0_func_i(...);  
                    break;  
        case dev1: dev1_func_i(...);  
                    break;  
        ...  
        case devM: devM_func_i(...);  
                    break;  
        case devNew: devNew_func_i(...);  
        break;  
    };  
    // Processing common to all devices  
    ...  
}
```



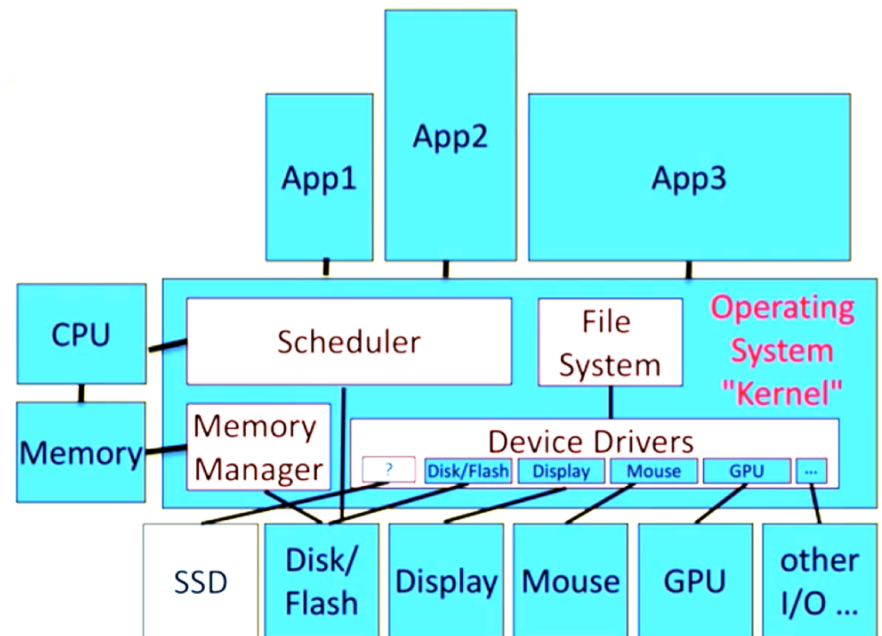
Adding a New Device

- After updating all `dev_func_*`(...) in the kernel, compile the kernel

Problem: Need to recompile the kernel, every time a new device or a new driver is added

Device have both device-independent and device-dependent code

- Need a way to **Dynamically add** new code into the OS kernel when new device needs to be supported
- Load *Device-Dependent* driver code into kernel
- Only load the device drivers as needed
- No kernel recompilation for changes in the device driver



Loadable Kernel Modules

- LKM is an object file that contains code to extend a running kernel
- Windows (kernel-mode driver), Linux (LKM), OS X (Kernel extension: kext), VmWorks
- LKMs can be loaded and unloaded from kernel on demand at runtime

LKMs

- Offer an easy way to extend the functionality of the kernel **without having to rebuild or recompile the kernel again**
- Simple and efficient way to create programs that reside in the kernel and run in privileged mode
- **Most of the drivers are written as LKMs**
- What's out there in the kernel? See `/lib/modules` for the all the LKMs
- `lsmod`: lists all kernel modules that are already loaded



How to write a kernel module?

- Kernel Modules are written in the C programming language.
- You must have a Linux kernel source tree to build your module.
- You must be running the **same kernel version** you built your module with to run it.
- Linux kernel object: **.ko** extension



Kernel Module: Basics

- A kernel module file has several typical components:
 - `MODULE_AUTHOR("your name")`
 - `MODULE_LICENSE("GPL")`
 - The license must be an open source license (GPL, BSD, etc.) or you will “taint” your kernel.
 - Tainted kernel loses many abilities to run other open source modules and capabilities.



Kernel Module: Key Operations

- `int init_module(void)`
 - Called when the kernel loads your module.
 - Initialize all your stuff here.
 - Return 0 if all went well, negative if something blew up.
- Typically, `init_module()` either
 - registers a handler for something with the kernel,
 - or replaces one of the kernel functions with its own code (usually code to do something and then call the original function)

Kernel Module: Key Operations

- `void cleanup_module(void)`
 - Called when the kernel unloads your module.
 - Free all your resources here.



Hello World Example

```
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_AUTHOR("Awesome Developer");
MODULE_LICENSE("GPL");

int init_module(void)
{
    printk(KERN_ALERT "Hello world: I am a developer in CS3753
                                speaking from the Kernel");
    return 0;
}
```


Hello World Example

```
void cleanup_module(void)
{
    printk(KERN_ALERT "Goodbye from a developer in CS3753,
                        I am exiting the Kernel");
}
```



Building Your Kernel Module

- Accompany your module with a 1-line GNU Makefile:
 - `obj-m += hello.o`
 - Assumes file name is “hello.c”
- Run the magic make command:
 - `make -C <kernel-src> M=`pwd` modules`
 - Produces: `hello.ko`
- Assumes current directory is the module source.



obj-\$(CONFIG_FOO) += foo.o

- Good definitions are the main part (heart) of the kbuild Makefile.
- The most simple kbuild makefile contains one line:
obj-\$(CONFIG_FOO) += foo.o
- This tell **kbuild** that there is one object in that directory named foo.o. foo.o will be built from foo.c or foo.S.
- \$(CONFIG_FOO) evaluates to either y (for built-in) or m (for module). If CONFIG_FOO is neither y nor m, then the file will not be compiled nor linked.



Loading Your Kernel Module: *insmod*

- Use *insmod* to manually load your kernel module

sudo insmod helloworld.ko

- *insmod* makes an *init_module* system call to load the LKM into kernel memory
- *init_module* system call invokes the LKM's initialization routine (also called *init_module*) right after it loads the LKM
- The LKM author sets up the initialization routine to call a kernel function that registers the subroutines that the LKM contains



Where is our Hello World message

- Dmesg
- /var/log/system.log

Unloading Your Kernel Module

- Use *rmmod* command

```
rmmod hello.ko
```

- Should print the Goodbye message



Kernel Module Dependencies: *modprobe*

- insmod/rmmod can be cumbersome...
 - You must manually enforce inter-module dependencies.
- *modprobe* automatically manages dependent modules
 - Copy hello.ko into /lib/modules/<version>
 - Run depmod
 - modprobe hello / modprobe -r hello
- Dependent modules are automatically loaded/unloaded.



- *depmod* creates a Makefile-like dependency file, based on the symbols it finds in the set of modules mentioned on the command line or from the directories specified in the configuration file
- This dependency file is later used by *modprobe* to automatically load the correct module or stack of modules



modinfo command

- .ko files contain an additional .modinfo section where additional information about the module is kept
 - Filename, license, dependencies, ...
- modinfo command retrieves that information
 - `modinfo hello.ko`