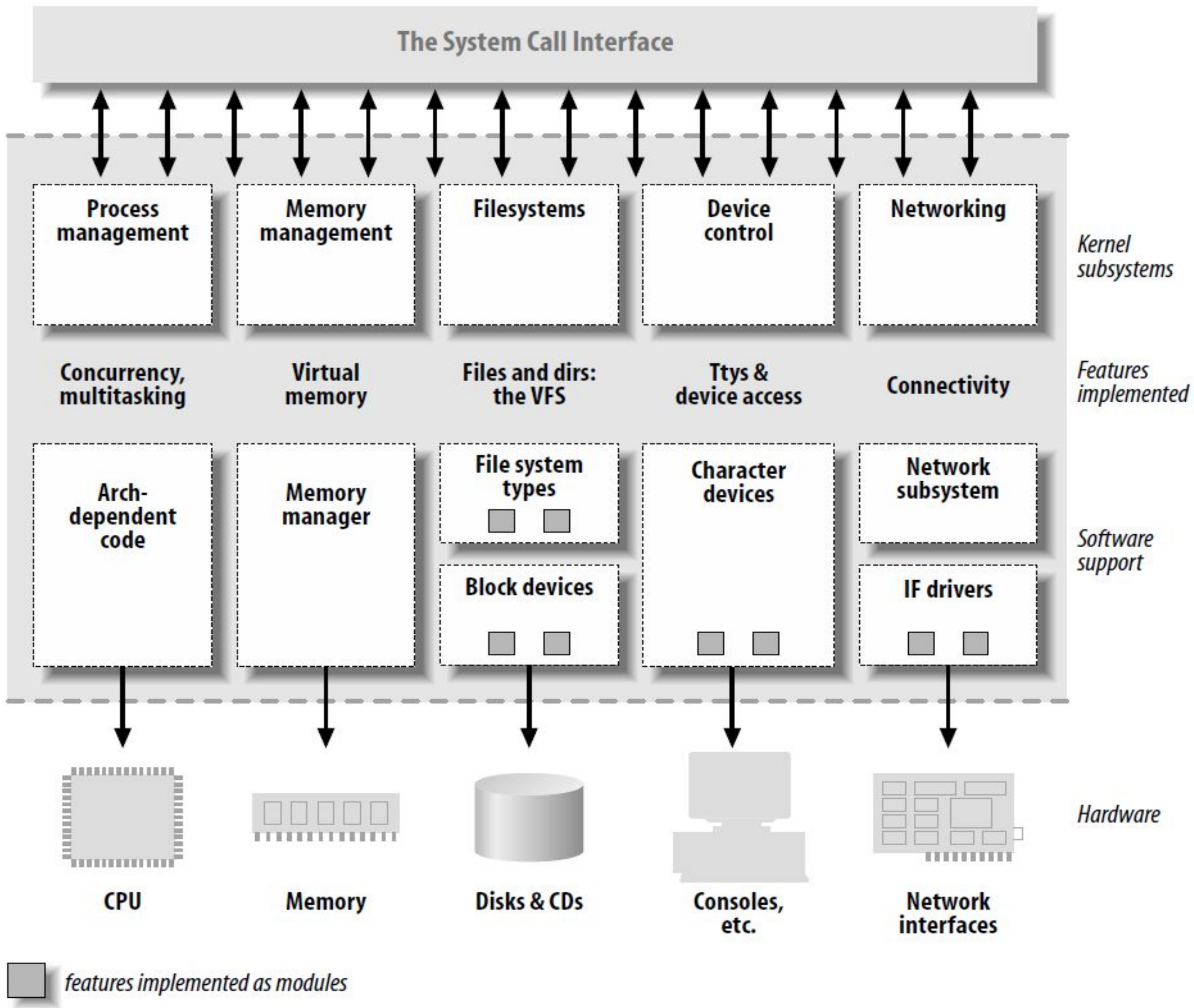
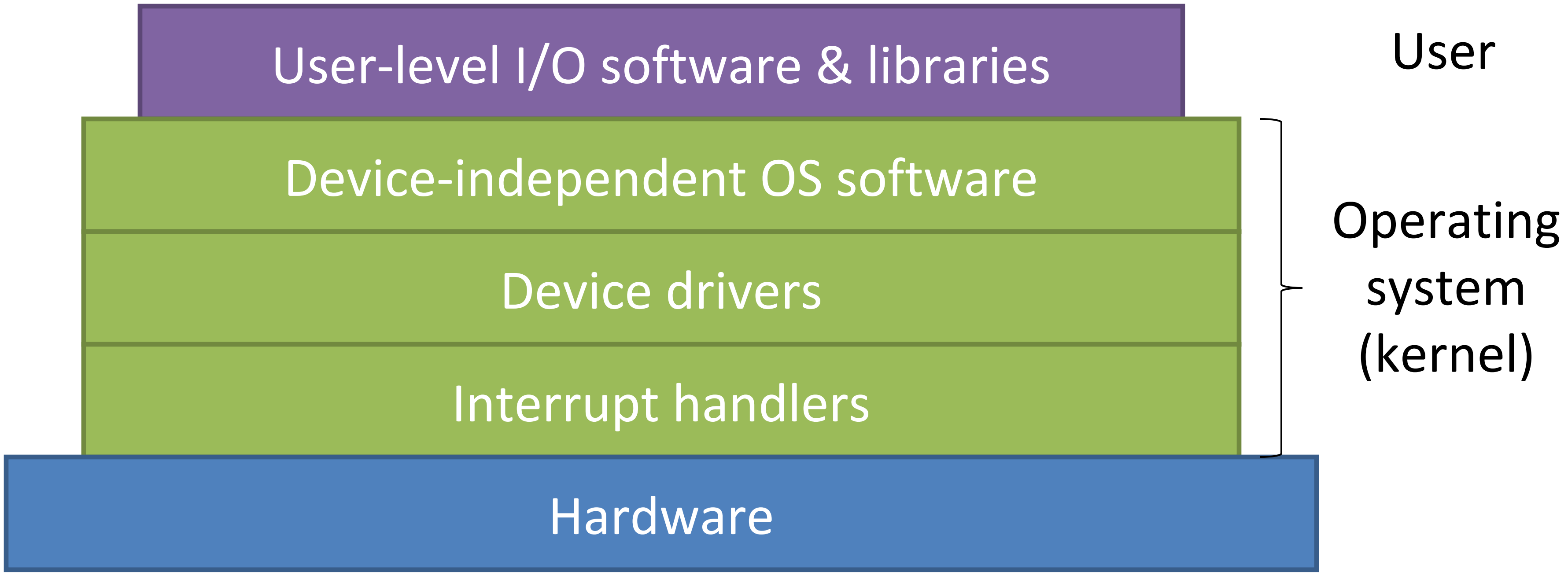
Device Drivers

CS449 Spring 2016

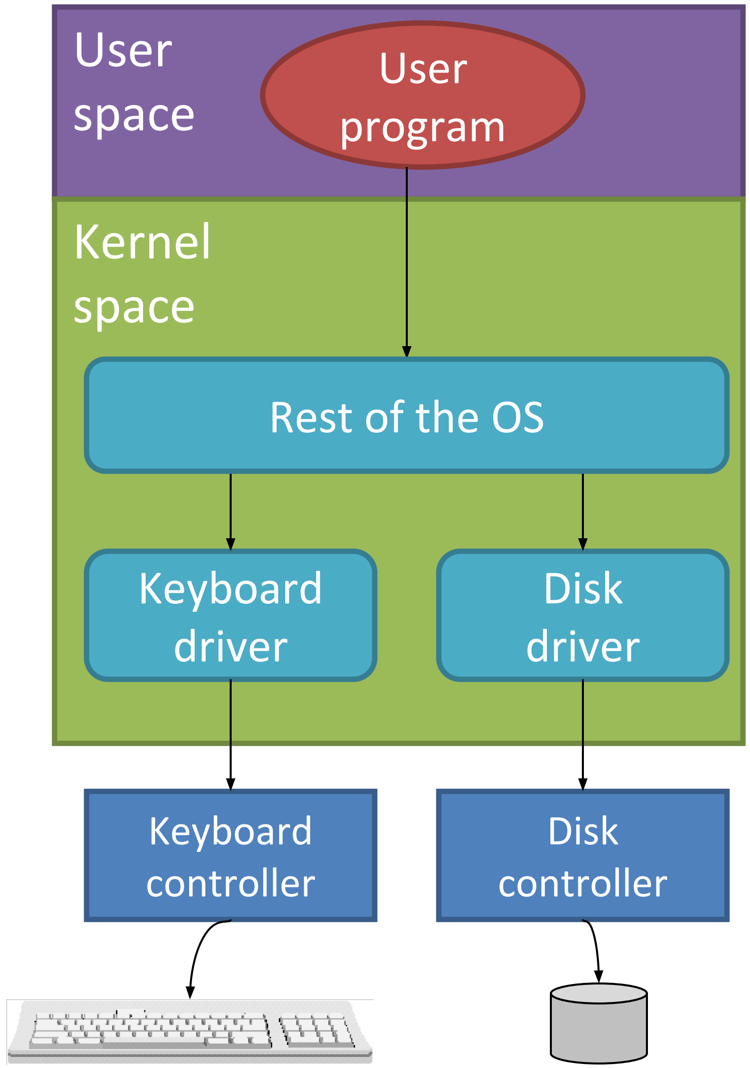
Abstraction via the OS



Software Layers



Device Drivers



Types of Devices

- **Block Devices**
  - A device that stores data in fixed-sized blocks, each uniquely addressed, and can be randomly accessed
  - E.g., Disks, Flash Drives
- **Character Devices**
  - Device that delivers or accepts a stream of characters
  - E.g., Keyboard, mouse, terminal

Mechanism vs. Policy

- Mechanism – What capabilities are provided
  - E.g. Hard disk driver: exposes the disk as a continuous array of data blocks
- Policy – How to use those capabilities
  - E.g. Data blocks can be organized using a file system, or they can be used as a raw block device
  - E.g. Data blocks may only be accessible by certain users
- Drivers should be flexible by only providing mechanisms not policies

Sysfs

- Mounted on `/sys/`

- Contains files that provide information about devices: whether it's powered on, the vendor name and model, what bus the device is plugged into, etc. It's of interest to applications that manage devices

- Exports information about devices and drivers to userspace

- Can configure aspects of device

Devfs

- Mounted on `/dev/`

- Contains files that allow programs to access the devices themselves: write data to a serial port, read a hard disk, etc. It's of interest to applications that access devices

- Character and block devices exposed via the filesystem

- `/dev/` typically contains “files” that represent the different devices on a system

- `/dev/console` – the console
- `/dev/fd/` - a process’s open file descriptors

Device Drivers in Linux

- Can be compiled into the kernel
- Can be loaded dynamically as Modules

Hello World Module

```c
#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);
module_exit(hello_exit);
```

Why printk?

- The kernel does not have access to libraries
- Can’t use printf or many other standard functions (FILE stuff, strtok, etc.)

- Modules are linked against the kernel only
- Kernel provides useful set of common functions like strcpy, strcat, etc.

MODULE_LICENSE

- Informs the kernel what license the module source code is under
- Affects which symbols (functions, variables, etc.) it may access in the kernel

- A GPL-licensed module can access everything
- Certain (or not specifying one) module license will “taint” the kernel

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.
  MODPOST
  CC /home/ldd3/src/misc-modules/hello.mod.o
  LD [M] /home/ldd3/src/misc-modules/hello.ko
  make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

Makefile

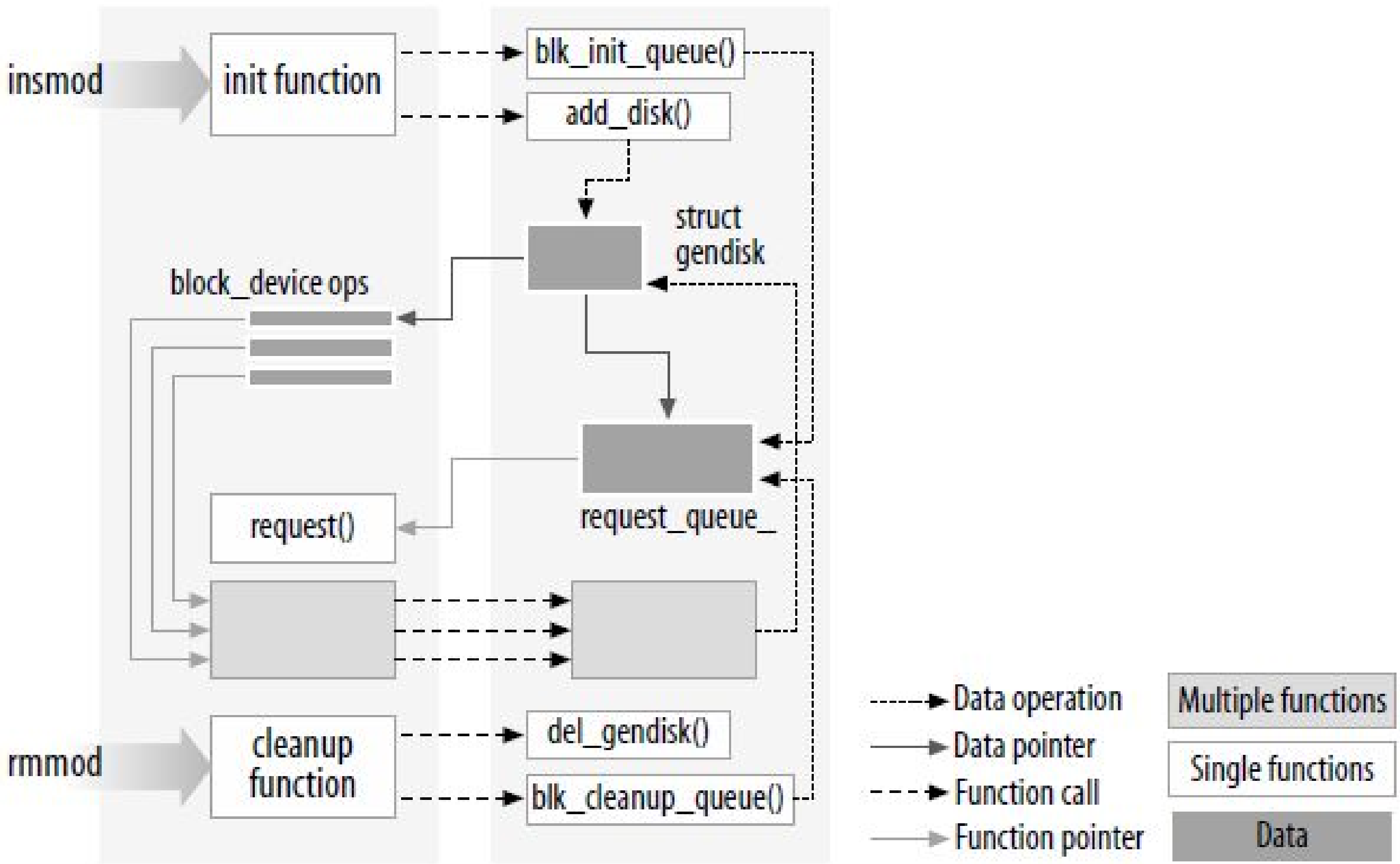
```
obj-m := hello_dev.o

KDIR  := /u/SysLab/shared/linux-2.6.23.1
PWD   := $(shell pwd)

default:
        $(MAKE) -C $(KDIR) M=$(PWD) modules
```

Module Helper Programs

- `insmod` – loads a module
- `rmmod` – unloads a module
- `lsmod` – lists what modules are loaded

- `modprobe` – loads a module checking dependencies

Loading & Unloading a Module



Hello World with Read function

Download full source from:

http://www.linuxdevcenter.com/2007/07/05/examples/hello_dev.c

```c
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/module.h>
#include <asm/uaccess.h>

/*
* hello_read is the function called when a process calls read() on
* /dev/hello. It writes "Hello, world!" to the buffer passed in the
* read() call.
*/

static ssize_t hello_read(struct file * file, char * buf, size_t count, loff_t *ppos)
{
	char *hello_str = "Hello, world!\n";
	int len = strlen(hello_str); /* Don't include the null byte. */
	/*
	 * We only support reading the whole string at once.
	 */
	if (count < len)
		return -EINVAL;
	/*
	 * If file position is non-zero, then assume the string has
	 * been read and indicate there is no more data to be read.
	 */
	if (*ppos != 0)
		return 0;
	/*
	 * Besides copying the string to the user provided buffer,
	 * this function also checks that the user has permission to
	 * write to the buffer, that it is mapped, etc.
	 */
	if (copy_to_user(buf, hello_str, len))
		return -EINVAL;
	/*
	 * Tell the user how much data we wrote.
	 */
	*ppos = len;

	return len;
}
```

```
/*
* The only file operation we care about is read.
*/

static const struct file_operations hello_fops = {
	.owner		= THIS_MODULE,
	.read		= hello_read,
};

static struct miscdevice hello_dev = {
	/*
	 * We don't care what minor number we end up with, so tell the
	 * kernel to just pick one.
	 */
	MISC_DYNAMIC_MINOR,
	/*
	 * Name ourselves /dev/hello.
	 */
	"hello",
	/*
	 * What functions to call when a program performs file
	 * operations on the device.
	 */
	&hello_fops
};
```

```
static int __init hello_init(void)
{
	int ret;

	/*
	 * Create the "hello" device in the /sys/class/misc directory.
	 * Udev will automatically create the /dev/hello device using
	 * the default rules.
	 */
	ret = misc_register(&hello_dev);
	if (ret)
		printk(KERN_ERR "Unable to register \"Hello, world!\" misc device\n");

	return ret;
}

module_init(hello_init);

static void __exit hello_exit(void)
{
	misc_deregister(&hello_dev);
}

module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Valerie Henson <val@nmt.edu>");
MODULE_DESCRIPTION("\"Hello, world!\" minimal module");
MODULE_VERSION("dev");
```

Test the Read function

Then, we're ready to compile and load the module:

```
$ cd hello_proc
$ make
$ sudo insmod ./hello_proc.ko
```

Now, there is a file named /proc/hello_world that will produce "Hello, world!" when read:

```
$ cat /proc/hello_world
Hello, world!
```

Device Operation Callbacks

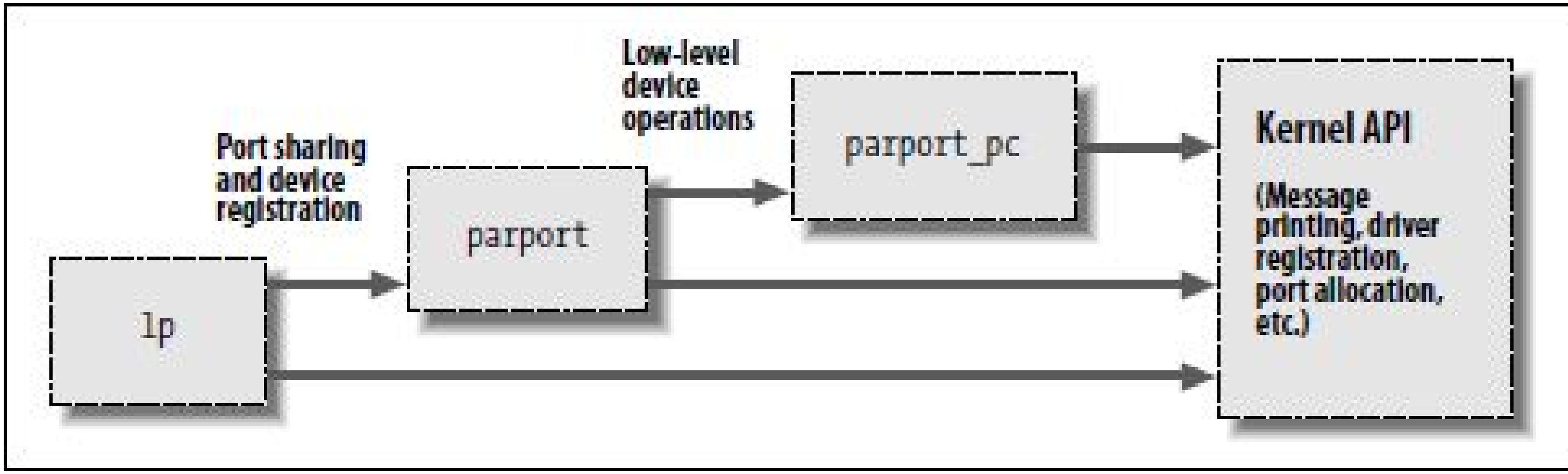
```
struct file_operations {
	struct module *owner;
	int (*open) (struct inode *, struct file *);
	ssize_t (*read) (struct file *, char *, size_t, loff_t *);
	ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
	...
};
```

- Struct containing callbacks passed to OS when driver is loaded
- OS does callbacks on functions on corresponding system call

Error Handling

```
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 */
}
```

Driver Stacking



Things not to do in the kernel

- Do not stack allocate big arrays
  - Kernel stack is small, maybe only a single page (4KB)
  - You should use kmalloc to allocate heap space

- Do not leave memory unfreed
  - Will stay around forever until the next reboot!

- Do not do floating point arithmetic
  - Context switch into the kernel does not save floating point registers

Race Conditions

- The kernel will make calls into your module while your initialization function is still running

- Multiple applications will attempt to access your driver simultaneously

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
  - Can be swapped to disk using virtual memory

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