ENGR 315 Course Takeover

Introduction to Linux kernel modules

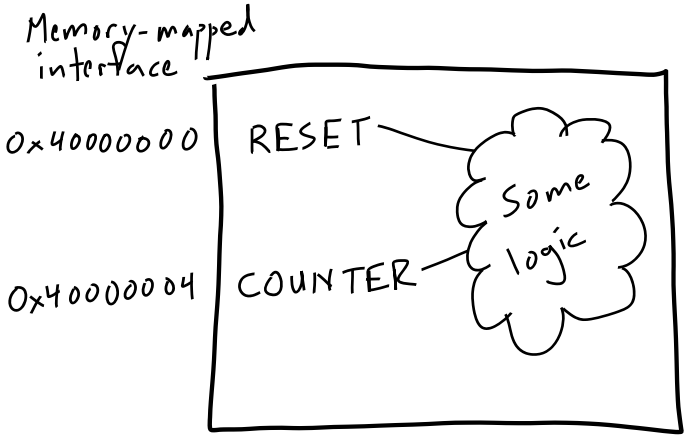
Austin Roach
ahroach@iu.edu

The hardware block

Popcount

~~Bitcount~~ block:

- Reset register at 0x40000000
- Counter register at 0x40000004
  - Write: input 32-bits at a time into counter
  - Read: receive count result



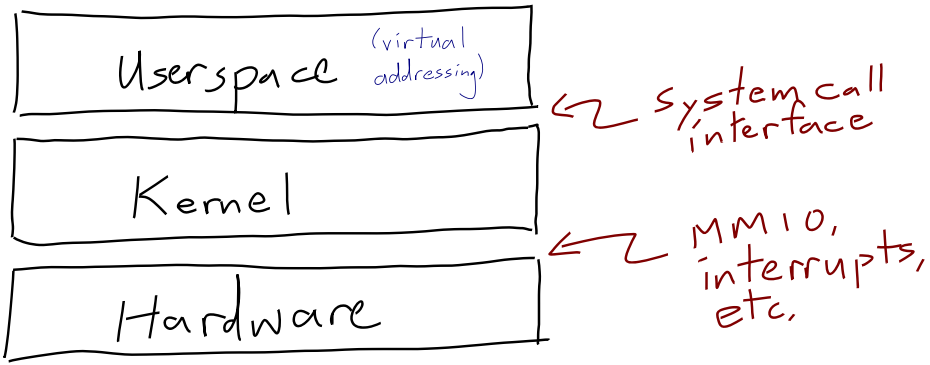
Back in the bare-metal days...

```c
#include <stdio.h>
#include <stdint.h>

int main()
{
        // Reset the device
        *(volatile uint32_t *)0x40000000 = 1;
        // Send in some data
        *(volatile uint32_t *)0x40000004 = 0xaaaa5555;
        // Print the number of ones
        printf("Result: %u\n", *(volatile uint32_t *)0x40000004);

        return 0;
}
```

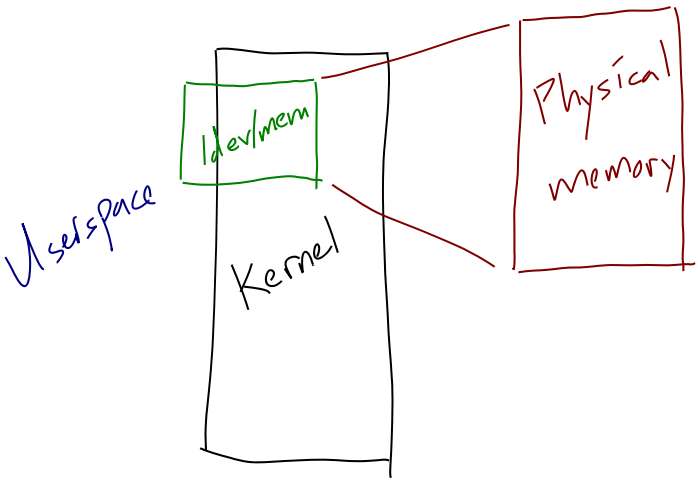
But when we run this on Linux...

```
xilinx@pynq:/tmp$ ./bare-metal-test
Segmentation fault (core dumped)

xilinx@pynq:/tmp$ gdb ./bare_metal_test
(gdb) run
Starting program: /tmp/bare_metal_test
Program received signal SIGSEGV, Segmentation fault.
0x00400516 in main ()
(gdb) x/i $pc
=> 0x400516 <main+10>: str r2, [r3, #0]
(gdb) info reg
...
r2             0x1 1
r3             0x40000000 1073741824
```

Operating system interfaces

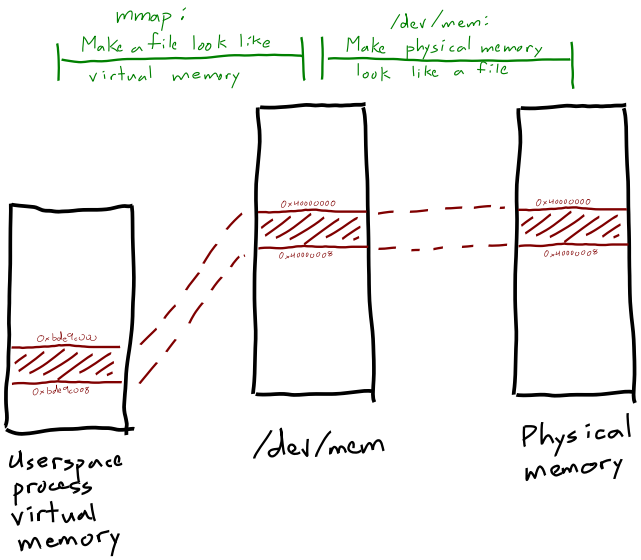


/dev/mem



See implementation in `tree/drivers/char/mem.c` of Linux kernel source

/dev/mem interaction

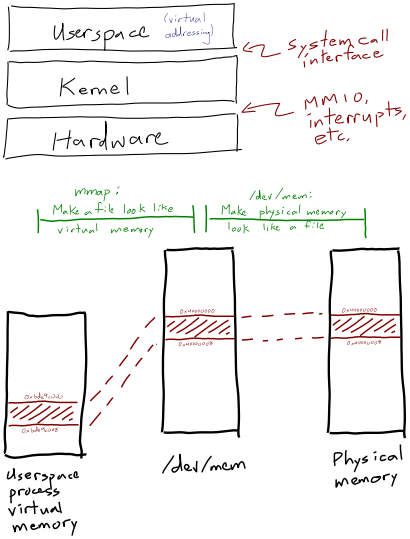


Last lecture

Interaction with hardware block from userspace:

- Memory-map `/dev/mem` to access physical memory
- Not super complicated
- But:
  - ~~Requires root~~
  - Programming mistakes could be disastrous
  - Concurrency issues
  - No good way to do DMA

Wouldn't it be great if something in the kernel could worry about hardware stuff for us?



Outline

- Basics of Linux kernel programming environment
- Building a kernel module
- “Hello, world!” kernel module
- “Hello, world!” character device driver

Outline

- Basics of Linux kernel programming environment
- Building a kernel module
- "Hello, world!" kernel module ← (intro to Linux kernel
- "Hello, world!" character device driver ← used for popcount

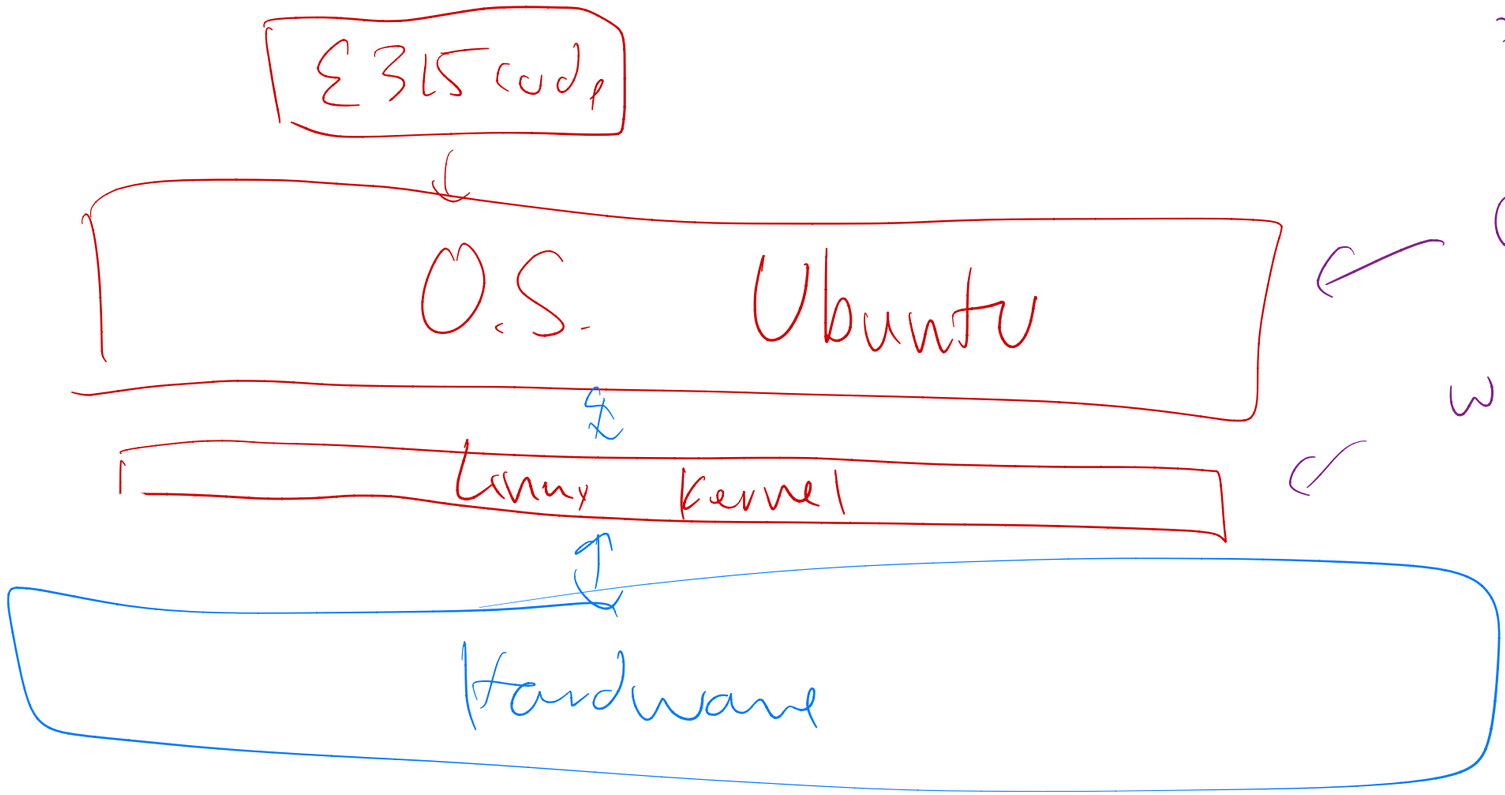
What do you know about Linux?

OSX / Darwin / BSD

*BSD / BSD

CentOS / Linux

windows / windows NT



Kernel versus userspace programming

- Kernel has an event-driven model
- No lower-level software to clean up after kernel
- C standard library does not apply
- Multiple processes could interact concurrently
- Highest level of system privilege*
  - Implications for system security and stability
- Symbols shared across the kernel
  - Use `static` to limit scope

#include <stdio.h>
<stdlib.h>

Kernel versus userspace programming

- Kernel has an event-driven model
- No lower-level software to clean up after kernel
- C standard library does not apply
- Multiple processes could interact concurrently
- Highest level of system privilege*
  - Implications for system security and stability
- Symbols shared across the kernel
  - Use `static` to limit scope

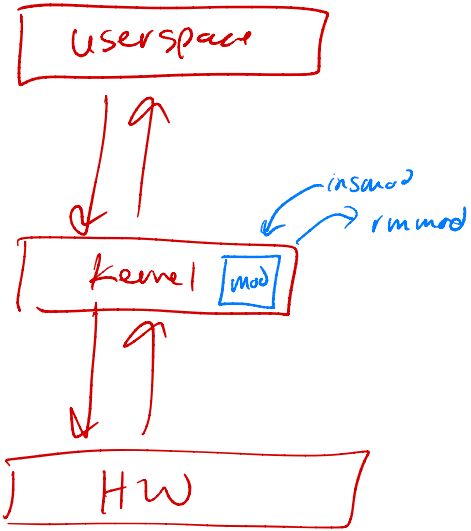
Linux kernel modules

- Pieces of code that can be *loaded* and *unloaded* on demand
  - `insmod`, `modprobe`, `rmmod`
- Can provide additional kernel functionality without rebooting the system
- Frequently used for device drivers

insmod → insert module
rmmod → rm module
lsmod → list modules

Linux kernel modules

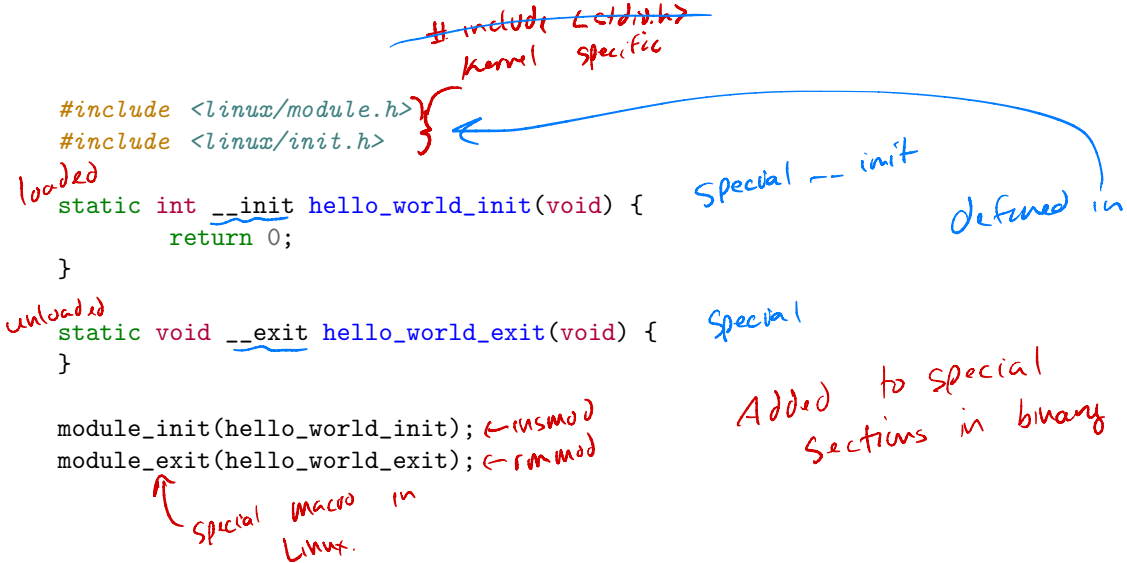
- Pieces of code that can be *loaded* and *unloaded* on demand
  - `insmod`, `modprobe`, `rmmod`
- Can provide additional kernel functionality without rebooting the system
- Frequently used for device drivers



Module init and exit

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

static int __init hello_world_init(void) {
        return 0;
}

static void __exit hello_world_exit(void) {
}

module_init(hello_world_init);
module_exit(hello_world_exit);
```

Module init and exit



Printing to kernel message ring buffer

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

static int __init hello_world_init(void) {
        pr_info("Hello, world!\n");
        return 0;
}

static void __exit hello_world_exit(void) {
        pr_info("Goodbye, world!\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);
```

How much do I want you to see this?

10/10 → pr_err

8/10 → pr_warn

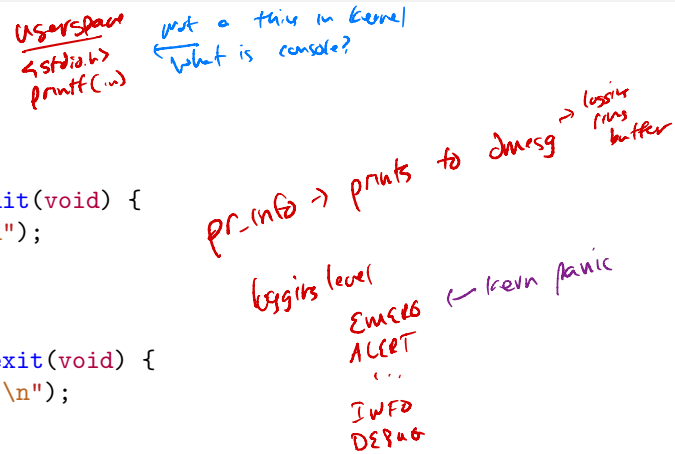
⋮

3/10 → pr_info

1/10 → pr_debug

Printing to kernel message ring buffer

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

static int __init hello_world_init(void) {
        pr_info("Hello, world!\n");
        return 0;
}

static void __exit hello_world_exit(void) {
        pr_info("Goodbye, world!\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);
```



Building a Linux kernel module

- Need Linux headers for development
  - On Debian-based systems, package like `linux-headers-`*arch* will pull in needed dependencies
  - Version *must* match the kernel that you are compiling for
- Standard Makefile location: `/lib/modules/$(shell uname -r)/build/Makefile`
  - We ask the kernel's build process to build our module on our behalf

On the Pynq, you must run '`make scripts prepare`' in `/lib/modules/$(shell uname -r)/build` once before `make` will work

And there might be timestamp issues. Set the system date if there are.

Building a Linux kernel module

- Need Linux headers for development
  - On Debian-based systems, package like `linux-headers-`*arch* will pull in needed dependencies
  - Version *must* match the kernel that you are compiling for
- Standard Makefile location: `/lib/modules/$(shell uname -r)/build/Makefile`
  - We ask the kernel's build process to build our module on our behalf

On the Pynq, you must run 'make scripts prepare' in `/lib/modules/$(shell uname -r)/build` once before `make` will work

~~And there might be timestamp issues. Set the system date if there are.~~

Makefile example

```
obj-m += hello_world.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Makefile example

try uname
try $(uname -r)

Makefile
kernel.h

Do Demo

```
obj-m += hello_world.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

$

print working directory

Hello world demo

Hello world demo

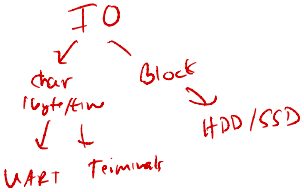
readelf -a hello-world.ko

.text
.init
.exit

sudo insmod hello_world.ko

lsmod
dmesg
rmmod hello_world.ko

Character devices



- Generic class of device drivers and device files
- Access performed sequentially, byte by byte
  - Contrast with block devices
- Anything you see in `ls -l /dev` with a leading `c` is a character device
  - Pretty much everything except disks

Character devices

Access to data byte by byte device
→ others are block devices

Block Device → SSD

- Generic class of device drivers and device files
- Access performed sequentially, byte by byte
  - Contrast with block devices
- Anything you see in `ls -l /dev` with a leading c is a character device
  - Pretty much everything except disks

crw - rw- ---

Hello world character device

Let's make a completely useless device:

- Read from the device: Always returns "Hello, world!"
- Write to the device: Nothing happens

Hello world character device

Let's make a completely useless device:

- Read from the device: Always returns "Hello, world!"
- Write to the device: Nothing happens

Character device init

- `register_chrdev()`
  - Register a device major number
  - Associates a device name and number with a `struct file_operations`
- `class_create()`
  - Create a `struct class` pointer, associating module and class name
- `device_create()`
  - Create a device, register with sysfs, and create /dev entry

This is just boiler plate stuff, similar for all character devices

Character device exit

Undo everything:

- `device_destroy()`
- `class_unregister()`
- `class_destroy()`
- `unregister_chrdev()`

Implementation for our device

Implementation for our device

Device Name → /dev/___

hello_world_init

- → register chrdev
- → class create
- → device create
- → goto: cleanup
- →

Character device operations

Defined in `linux/fs.h`:

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

Character device operations

Defined in `linux/fs.h`:

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

function pointer

linux-kernel / linux/fs.h

struct file_operations

open

close

open() implementation

```
static int hello_world_open(struct inode *inodep, struct file *filep)
{
        return 0;
}
```

open() implementation

```
static int hello_world_open(struct inode *inodep, struct file *filep)
{
        return 0;
}
```

success

release() implementation

```
static int hello_world_release(struct inode *inodep, struct file *filep)
{
        return 0;
}
```

write() implementation

```
static ssize_t hello_world_write(struct file *filep, const char __user *buf,
                size_t len, loff_t *offset)
{
        return len;
}
```

write() implementation

```
static ssize_t hello_world_write(struct file *filep, const char __user *buf,
                size_t len, loff_t *offset)
{
        return len;
}
```

Moving data to/from userspace

Defined in `linux/uaccess.h`:

Copy a block of data *to* userspace

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

Copy a block of data *from* userspace

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

Both return number of bytes that *could not* be copied

read() implementation

```
static ssize_t hello_world_read(struct file *filep, char __user *buf,
                size_t len, loff_t *offset)
{
        char * msg = "Hello, world!\n";
        size_t msg_len, bytes_to_copy;

        msg_len = strlen(msg);
        bytes_to_copy = msg_len*(len/msg_len); // No half messages!

        while (bytes_to_copy > 0) {
                if (copy_to_user(buf, msg, msg_len)) {
                        return -EFAULT;
                }
                buf += msg_len;
                bytes_to_copy -= msg_len;
        }

        return msg_len*(len/msg_len);
}
```

read() implementation

```
static ssize_t hello_world_read(struct file *filep, char __user *buf,
                size_t len, loff_t *offset)
{
        char * msg = "Hello, world!\n";
        size_t msg_len, bytes_to_copy;

        msg_len = strlen(msg);
        bytes_to_copy = msg_len*(len/msg_len); // No half messages!

        while (bytes_to_copy > 0) {
                if (copy_to_user(buf, msg, msg_len)) {
                        return -EFAULT;
                }
                buf += msg_len;
                bytes_to_copy -= msg_len;
        }

        return msg_len*(len/msg_len);
}
```

repeat hello world for longer messages

error / problem

# of bytes "read"

struct file_operations assignment

```
static struct file_operations fops =
{
        .open = hello_world_open,
        .read = hello_world_read,
        .write = hello_world_write,
        .release = hello_world_release
};
```

C99 form

others are NULL

udev rule

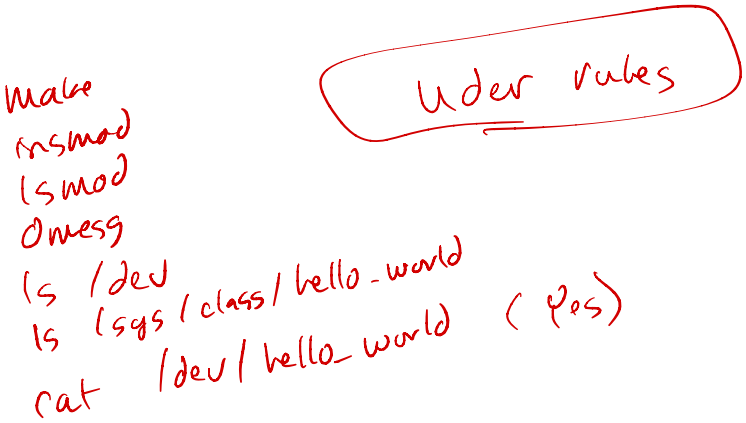
udev rule

udev rule for our preferred device permissions mode:

```
KERNEL=="hello_world", SUBSYSTEM=="hello_world", MODE="0666"
```

Code summary and demo

Code summary and demo



Summary

What we accomplished:

- We have our own code running in the kernel!!!!
- Simple character device driver
- Can send and receive data to/from userspace

Next time:

- ~~bitcount~~ oopscount-specific stuff
  - Resource locking
  - MMIO
  - DMA

Resources

- J. Corbet, A. Rubini, and G. Kroah-Hartman, “Linux Device Drivers, 3rd ed.”, O’Reilly Media, 2005.
  - A little dated, but great
  - Available on-line for free: `https://lwn.net/Kernel/LDD3/`
- In-tree documentation
- Device drivers that do something similar to what you want to do