CSC369 Tutorial 2

Kernel modules
Synchronization (basics)
A1 & system calls

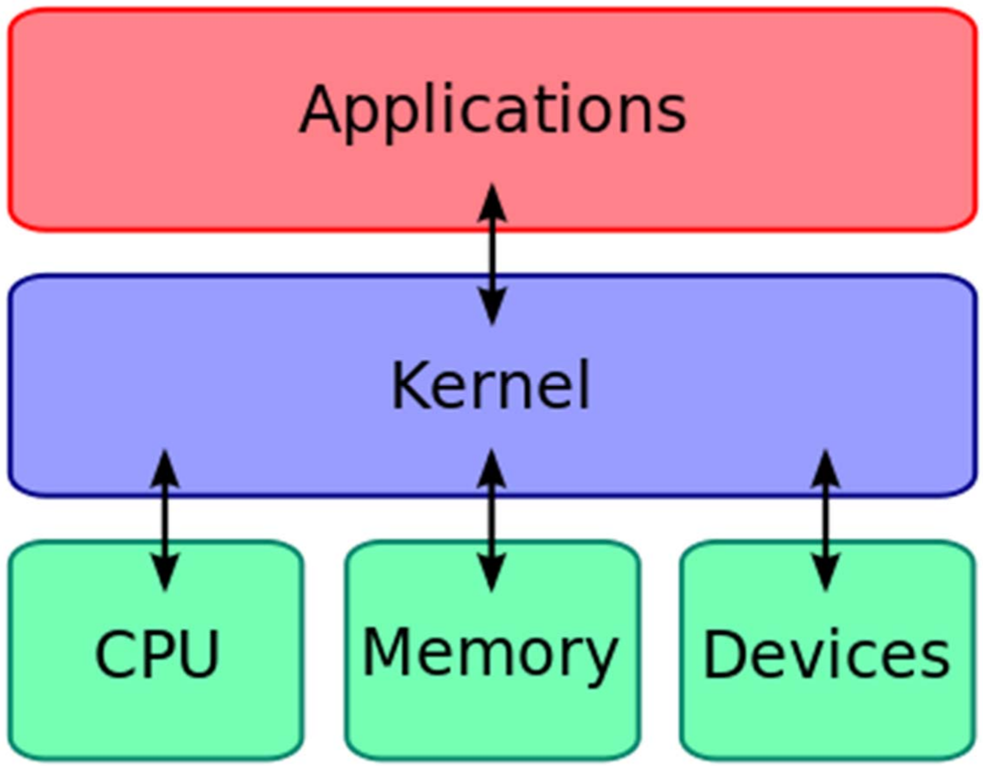
Handout is being distributed, read it and you'll find the answers along today's tutorial.

Vote on the poll for A1 office hours!

Kernel

- Kernel = computer program that connects the user applications to the system hardware
- Handles:
  - Memory management
  - CPU scheduling (Process and task management)
  - Disk management
  - User access to other I/O devices (e.g., network card)

Kernel



Source: wikipedia.org

Kernel modules

- Object file that contains code to extend the kernel's functionality
- Why do we need them? Why not include all possible functionality in the kernel directly?
  - Kernel code lies in main memory (limited resource)
  - Kernel should be minimal
  - Avoid functionality bloating
  - For each new functionality added => recompile kernel, reboot, … ugh!
  - Instead, develop modules separately, load as needed
  - Modularity => Better chance to recover from buggy new code without a complete kernel crash!

Why should I bother?

- Because it’s cool!

- Better understanding on how the OS works
- Write awesome extensions to the OS
- Write your own device drivers!

Linux kernel modules

- Basic utilities:
  - `insmod`: to load a module
  - `rmmod`: to unload a module
- The `modprobe` utility
  - More complex, deals with module dependencies
  - We won't be using it
- Module objects: `.ko` files

Loading/unloading a kernel module

- As root, or using sudo
- Loading:
  - `insmod mymodule.ko`
- Unloading:
  - `rmmod mymodule.ko` (or: `rmmod mymodule`)
- Entry point:
  - `module_init(mymodule_init);`
  - `module_exit(mymodule_exit);`

Kernel module example

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

MODULE_DESCRIPTION("My kernel module");
MODULE_AUTHOR("John Doe");
MODULE_LICENSE("GPL");

static int mymodule_init(void) {
        printk( KERN_DEBUG "Hello world!\n" );
        return 0;
}

static void mymodule_exit(void) {
        printk( KERN_DEBUG "I'm outta here\n" );
}

module_init(mymodule_init);
module_exit(mymodule_exit);
```

Compiling a module

- Different than a regular C program
- Must use different headers
- Must not link with libraries. Why?
- Must be compiled with the same options as the kernel in which we want to load it
- Standard method: kbuild
  - Two files: a Makefile, and a Kbuild file

Example

- Makefile:

```
KDIR=/lib/modules/`uname -r`/build
kbuild:
    make -C $(KDIR) M=`pwd`
clean:
    make -C $(KDIR) M=`pwd` clean
```

- Kbuild file:

```
EXTRA_CFLAGS=-g
obj-m  = mymodule.o
```

Printing messages

- Use printk
  - e.g., `printk( KERN_DEBUG “Hello world\n");`
- Dude, where’s my output?
  - Not displayed at stdout
  - Can be retrieved from the system logs
  - Use `dmesg` command
    - can use grep to search and filter:
      dmesg | grep Hello -C 3

Debugging a kernel module

- More complicated than a regular program
- A bug in a module can lead to the whole OS malfunctioning
- Buggy module: can lead to a “kernel oops”
- Avoid reboot cycles => use VM for CSC369!
  - VM snapshots can be useful
- Do not develop modules directly on your Linux box without a VM! – painfully slow!
- For A1, use rudimentary (yet efficient) method: printk statements

Debugging a kernel module

- You can use a debugger, but not very useful
  - Simple bugs can be tracked easily with printks
  - Use ksymoops utility
- Complex bugs – not even a debugger will help as much
  - Need to know in depth the OS structure
  - Multiple contexts, interrupts, VM, etc.
- Kernel oops message can be translated using ksymoops (memory locations, backtrace, etc.)
  - http://opensourceforu.com/2011/01/understanding-a-kernel-oops/

Linux kernel API – some differences

- Different headers – make sure to include them!
- Success/failure conventions:
  - 0 == success
  - Non-zero (*negative*) == failure (-ENOMEM, -EINVAL, etc.)
  - See <include/asm-generic/errno-base.h> and <include/asm-generic/ernno.h>
- Memory allocation: kmalloc/kfree

```
#include <linux/malloc.h>
if(!(string = kmalloc(len+1, GFP_KERNEL))) {
     return -ENOMEM;
}
...
kfree(string);
```

- GFP_KERNEL can lead to suspending the current process; so it cannot be used in an interrupt context.
- GFP_ATOMIC when the kmalloc won't suspend the current process; can be used anywhere

Strings and printing

- Standard string functions:
  - strcmp, str(n)cpy, str(n)cat, memcpy, etc.
- Same header: <string.h>

- Printing: `printk`, defined in `<linux/kernel.h>`
- Similar syntax, plus category of message:

```
printk(KERN_WARNING “Uh-oh, you better check this: %s\n", buff);
printk(KERN_DEBUG “This buffer looks spooky: %s\n", buff);
#define KERN_EMERG  "<0>"   /* system is unusable    */
#define KERN_ALERT  "<1>"   /* action must be taken immediately */
#define KERN_CRIT   "<2>"   /* critical conditions      */
#define KERN_ERR    "<3>"   /* error conditions      */
#define KERN_WARNING   "<4>"   /* warning conditions      */
#define KERN_NOTICE "<5>"   /* normal but significant condition */
#define KERN_INFO   "<6>"   /* informational      */
#define KERN_DEBUG  "<7>"   /* debug-level messages      */
```

Synchronization

- Concurrent multiple threads accessing the same resource (e.g. shared data)
- Things can go wrong if access is not protected
  - example: 2 threads incrementing a counter:
    - increment is *not atomic*, reading the counter and writing the value back are separate operations

```
// Thread 1:
updated = counter + 1;

counter = updated;
```

```
// Thread 2:

updated = counter + 1;

counter = updated;
```

Synchronization: locks

- Synchronization mechanisms with 2 operations: acquire (lock), and release (unlock)
- In simplest terms: an object associated with a particular critical section that you need to "own" if you wish to execute in that region
- Simple semantics to provide mutual exclusion:

```
acquire(lock);
    // CRITICAL SECTION
release(lock);
```

- Downsides:
  - Can cause deadlock if not careful
  - Cannot allow multiple concurrent accesses to a resource

Synchronization: spinlocks

- Busy-waiting synchronization
- Based on hardware atomic instructions, e.g. TAS – test-and-set
  - pseudocode:

```
bool test_and_set(bool *lock) {
// Atomically set to true and
// return the old value
bool old = *lock;
*lock = true;
return old;
}
```

Suppose many threads are calling this function with the same pointer,
Which threads will get False as a return value?

Synchronization: spinlocks

- TAS spinlock implementation:

```
bool lock = false;

void acquire(bool *lock) {
// Wait until the lock becomes available
while (test_and_set(lock));
}

void release(bool *lock) {
// Make the lock available
*lock = false;
}
```

Synchronization: spinlocks

- Linux kernel spinlocks: `spinlock_t` type
  - `spinlock_t myspinlock = SPIN_LOCK_UNLOCKED;`
- Operations:
  - `spin_lock_init(&myspinlock)`
  - `spin_lock/unlock(&myspinlock)`
- Can also use read/write spinlocks: `rwlock_t`
  - `rwlock_init(), read_lock(), write_lock()`
- Check out: <include/linux/spinlock.h>
- Will learn more about synchronization later in the course!

System calls

- Application:

```
write(fd, buffer, size);
```

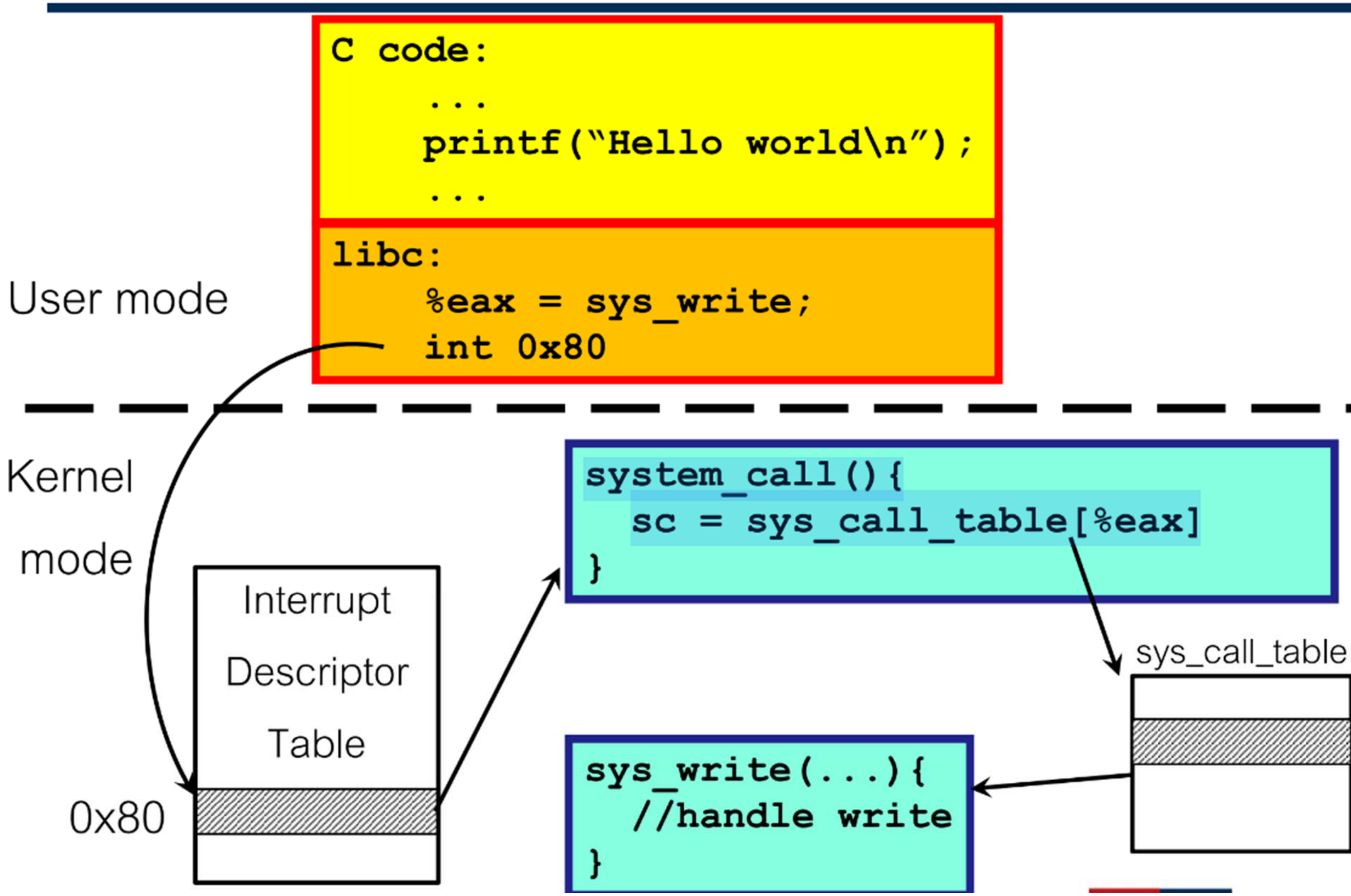
- libc wrapper for write():

```
eax = __NR_write;
ebx = buffer;
ecx = size;
//...(up to 6 parameters in total)
sysenter;
return;
```

- Kernel syscall trap handler:

```
syscall_table[eax](ebx, ecx, ...);
sysexit;
```

A normal system call



System calls: fun fact

- Which syscall is the most frequent?

System calls: fun fact

- Which syscall is the most frequent?

Answer: *gettimeofday()*

(at least *one of* the most frequent)

Linux even has special optimizations for this call

(a memory page mapped into both the kernel space and the user space for every process)

Assignment 1

- Monitoring *system calls* made by processes
  - “highjack” syscalls of interest: substitute entries in the syscall table with our own function that records call info and invokes the original
  - Some system calls: open, read, write, close, wait, exec, fork, exit, and kill
- Kernel programming
- Test and debug in a virtual machine
- Read the handout and comments in the starter code very thoroughly

VM setup for A1

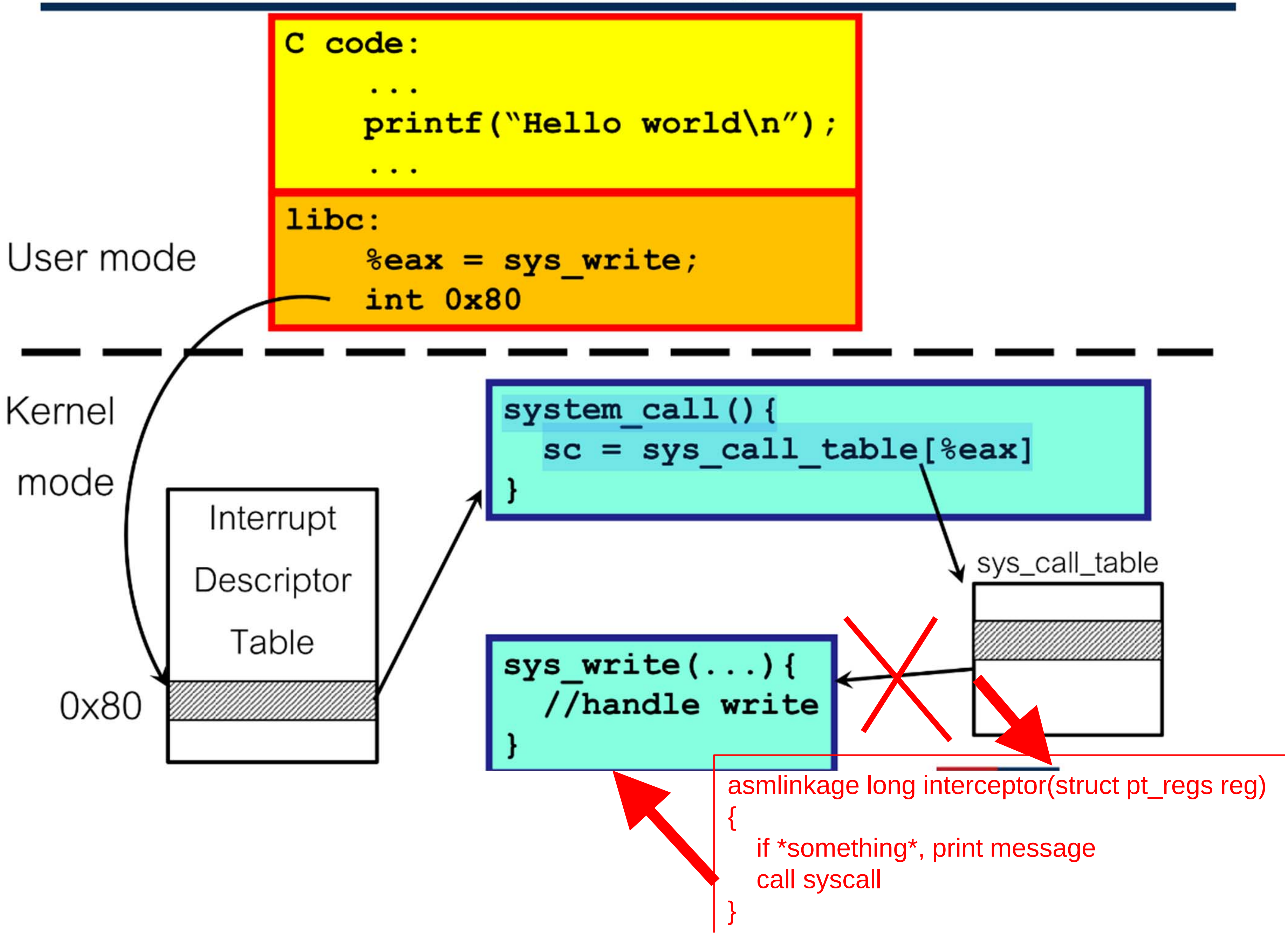
- Don’t test your kernel module on a physical machine, use a virtual machine (VirtualBox/VMWare/etc.)
- A .vmdk virtual disk file is provided
- Use CDF lab machines or your own computers; follow instructions in the assignment handout
- Develop on physical machine, test in VM

VM setup for A1

- Ways to transfer files to the VM:
  - Shared folders; need Guest Additions to be installed in the VM
  - Enable network in the VM, setup ssh server, then scp/rsync/svn checkout/etc.
    - also ssh into the VM (to have a nicer terminal)

Check Piazza, assignment handout, and google for instructions and help with VM setup

An intercepted system call



Some A1 tips

- Be careful with corner cases;
  document any assumptions that you make
  - as comments in code, or in your info.txt file
- Cleanup everything properly on unload; need to restore ***all*** kernel state that you have modified
- Passing all the provided tests doesn't guarantee that everything works correctly
- Be very careful with synchronization
  - e.g. do not forget to release locks before return when handling error conditions