LDD3, Ch 2

ECEN 427

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

```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);
```

While most small and medium-sized applications perform a single task from beginning to end, every kernel module just registers itself in order to serve future requests, and its initialization function terminates immediately. In other words, the task of the module's initialization function is to prepare for later invocation of the module's functions; it's as though the module were saying, "Here I am, and this is what I can do." The module's exit function (hello_exit in the example) gets invoked just before the module is unloaded. It should tell the kernel, "I'm not there anymore; don't ask me to do anything else."

This kind of approach to programming is similar to **event-driven programming**, but while not all applications are event-driven, each and every kernel module is. Another major difference between event-driven applications and kernel code is in the exit function: whereas an application that terminates can be lazy in releasing resources or avoids clean up altogether, the **exit function of a module must carefully undo everything the init function built up**, or the pieces remain around until the system is rebooted.

Incidentally, the ability to unload a module is one of the features of modularization that you'll most appreciate, because it helps cut down development time; you can test successive versions of your new driver without going through the lengthy shutdown/reboot cycle each time.

Kernel Source Code

- https://github.com/torvalds/linux
- https://elixir.bootlin.com/linux/latest/source
- uname –r
- https://elixir.bootlin.com/linux/v5.4/source
-

Libraries

As a programmer, you know that an application can call functions it doesn't define: the linking stage resolves external references using the appropriate library of functions. printf is one of those callable functions and is **defined in libc**. A module, on the other hand, is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; **there are no libraries to link to**. The printk function used in hello.c earlier, for example, is the version of printf defined within the kernel and exported to modules. It behaves similarly to the original function, with a few minor differences, the main one being lack of floating-point support.

Because no library is linked to modules, source files should never include the usual header files, <stdarg.h> and very special situations being the only exceptions.

```
TARGET_MODULE=audio_codec

ccflags-y := -std=gnu99 -Wno-declaration-after-statement

# If we are running by kernel building system
ifneq ($(KERNELRELEASE),)
        obj-m := $(TARGET_MODULE).o
# If we running without kernel build system
else
        BUILDSYSTEM_DIR:=/lib/modules/$(shell uname -r)/build
        PWD:=$(shell pwd)

all :
        # run kernel build system to make module
        $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) modules

clean:
        # run kernel build system to cleanup in current directory
        $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) clean

install:
        $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) modules_install
        depmod -A

endif
```

- cd kernel/helloworld
- sudo insmod helloworld.ko
- dmesg
- lsmod
- sudo rmmod helloworld
- dmesg

User Space and Kernel Space

The role of the operating system, in practice, is to provide programs with a consistent view of the computer's hardware. In addition, the operating system must account for independent operation of programs and protection against unauthorized access to resources. This nontrivial task is possible only if the CPU enforces protection of system software from the applications.

Every modern processor is able to enforce this behavior. The chosen approach is to implement different operating modalities (or levels)in the CPU itself. The levels have different roles, and some operations are disallowed at the lower levels; program code can switch from one level to another only through a limited number of gates. Unix systems are designed to take advantage of this hardware feature, using two such levels.

All current processors have at least two protection levels, and some, like the x86 family, have more levels; when several levels exist, the highest and lowest levels are used. Under Unix, the kernel executes in the highest level (also called supervisor mode), where everything is allowed, whereas applications execute in the lowest level (the so-called user mode), where the processor regulates direct access to hardware and unauthorized access to memory.

User Space and Kernel Space

We usually refer to the execution modes as **kernel space** and **user space**. These terms encompass not only the different privilege levels inherent in the two modes, but also the fact that each mode can have its own memory mapping—its own address space—as well.

Unix transfers execution from user space to kernel space whenever an application issues a system call or is suspended by a hardware interrupt. Kernel code executing a system call is working in the context of a process—it operates on behalf of the calling process and is able to access data in the process's address space. Code that handles interrupts, on the other hand, is asynchronous with respect to processes and is not related to any particular process.

330 Review

What are the different segments of memory where your program and it's data are stored?

1.

2.

3.

4.

Kernel Stack

Applications are laid out in virtual memory with a very large stack area. The stack, of course, is used to hold the function call history and all automatic variables created by currently active functions. The kernel, instead, has a very small stack; it can be as small as a single, 4096-byte page. Your functions must share that stack with the entire kernel-space call chain. Thus, it is never a good idea to declare large automatic variables; if you need larger structures, you should allocate them dynamically at call time.

__Names

Often, as you look at the kernel API, you will encounter function names starting with a double underscore (__). Functions so marked are generally a low-level component of the interface and should be used with caution. Essentially, the double underscore says to the programmer: “If you call this function, be sure you know what you are doing.”

Floating Point

Kernel code cannot do floating point arithmetic. Enabling floating point would require that the kernel save and restore the floating point processor's state on each entry to, and exit from, kernel space—at least, on some architectures. Given that there really is no need for floating point in kernel code, the extra overhead is not worthwhile.

Error Handling

"Error recovery is sometimes best handled with the goto statement. We normally hate to use goto, but in our opinion, this is one situation where it is useful."

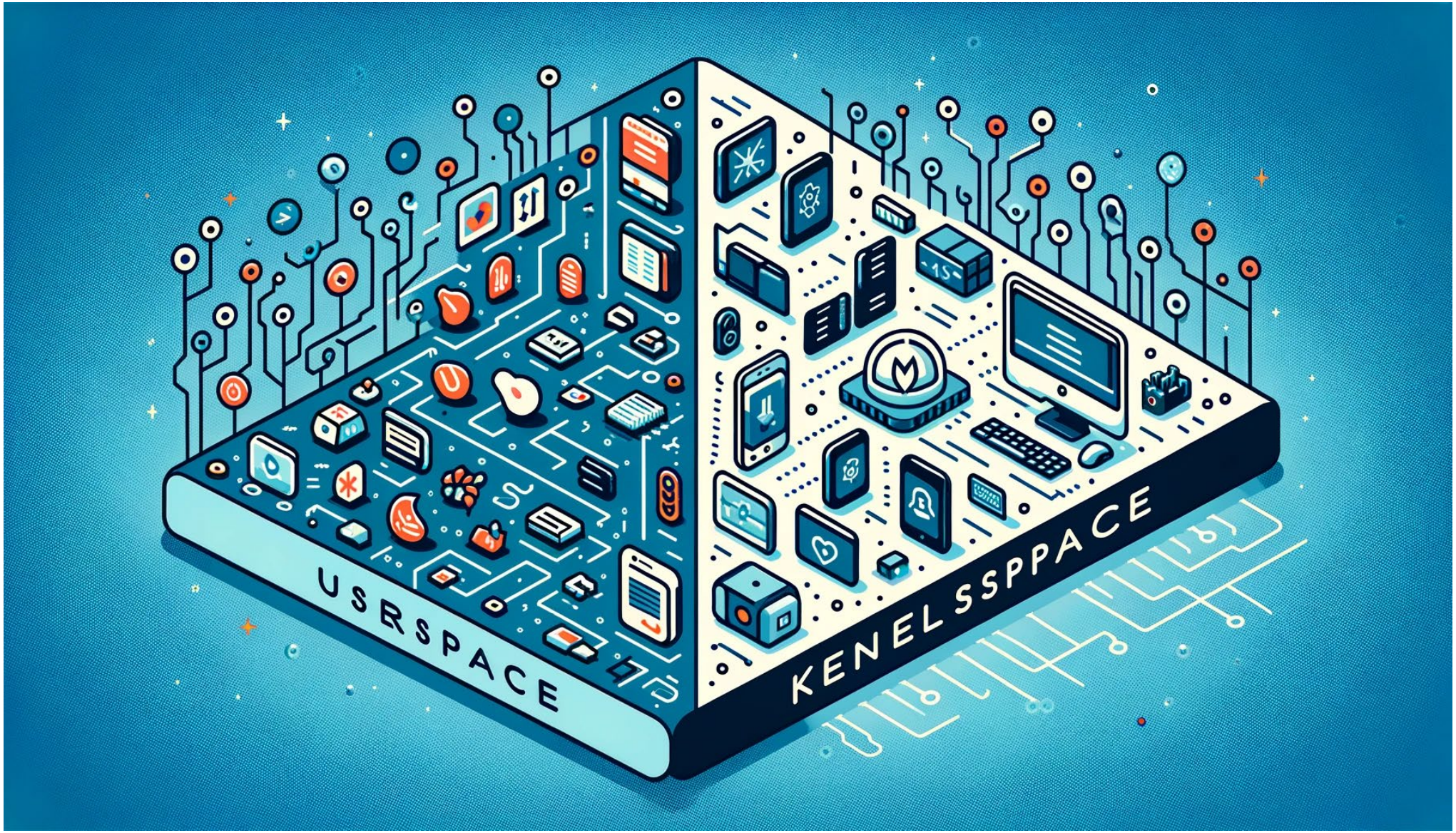
```
int __init my_init_function(void)
{
    int err;

    /* registration takes a pointer and a name */
    err = register_this(ptr1, "skull");
    if (err) goto fail_this;
    err = register_that(ptr2, "skull");
    if (err) goto fail_that;
    err = register_those(ptr3, "skull");
    if (err) goto fail_those;

    return 0; /* success */

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

```
void __exit my_cleanup_function(void)
{
    unregister_those(ptr3, "skull");
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    return;
}
```

```
int __init my_init(void)
{
    int err = -ENOMEM;

    item1 = allocate_thing(arguments);
    item2 = allocate_thing2(arguments2);
    if (!item2 || !item2)
        goto fail;
    err = register_stuff(item1, item2);
    if (!err)

        stuff_ok = 1;
    else
        goto fail;
    return 0; /* success */

  fail:
    my_cleanup( );
    return err;
}
```

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;

void my_cleanup(void)
{
    if (item1)
        release_thing(item1);
    if (item2)
        release_thing2(item2);
    if (stuff_ok)
        unregister_stuff( );
    return;
 }
```

User Space vs Kernel Space



User space vs Kernel Drivers

The advantages of user-space drivers are:

- The full C library can be linked in. The driver can perform many exotic tasks without resorting to external programs (the utility programs implementing usage policies that are usually distributed along with the driver itself).
- The programmer can run a conventional debugger on the driver code without having to go through contortions to debug a running kernel.
- If a user-space driver hangs, you can simply kill it. Problems with the driver are unlikely to hang the entire system, unless the hardware being controlled is really misbehaving.
- User memory is swappable, unlike kernel memory. An infrequently used device with a huge driver won't occupy RAM that other programs could be using, except when it is actually in use.
- A well-designed driver program can still, like kernel-space drivers, allow concurrent access to a device.
- If you must write a closed-source driver, the user-space option makes it easier for you to avoid ambiguous licensing situations and problems with changing kernel interfaces.

User space vs Kernel Drivers

But the user-space approach to device driving has a number of drawbacks. The most important are:

- Interrupts are not available in user space.
- Direct access to memory is possible only by mmapping /dev/mem, and only a privileged user can do that.
- Access to I/O ports is available only after calling ioperm or iopl. Moreover, not all platforms support these system calls, and access to /dev/port can be too slow to be effective. Both the system calls and the device file are reserved to a privileged user.
- Response time is slower, because a context switch is required to transfer information or actions between the client and the hardware.
- Worse yet, if the driver has been swapped to disk, response time is unacceptably long. Using the mlock system call might help, but usually you'll need to lock many memory pages, because a user-space program depends on a lot of library code. mlock, too, is limited to privileged users.
- The most important devices can't be handled in user space, including, but not limited to, network interfaces and block devices.