Linux Misc Drivers

Kernel coding is differen

Can be hard to understar
different syntax, function
advanced C code in kern

Caution

Topics

1) How can we easily create a new driver?
2) How can we read data from a driver?
3) Are user level pointers dangerous?
4) How can we write data to a driver?

Setting up a Misc Driver

Driver Interaction

- Programs and users interact with drivers via nodes (files) in the file system
  - /dev/ - access the driver's service

    ```
    (host)$ echo 'Hello world' > /dev/ttyUSB0
    ```

  - /proc/ - access information about the driver

    ```
    (bbg)$ cat /proc/cmdline
    ```

- So, a driver creates nodes to allow access to it.

Misc-Driver

- Writing a driver can be complicated!
  - Allocating major/minor..
    for connecting into the file system
  - Creating nodes (files) in /dev and /sys for interacting with driver
  - Registering as a character (char) driver
- Kernel helps with a simplified structure for “normal” drivers:

  ..

Misc Data Structures

- #include <linux/miscdevice.h>
- struct miscdevice
  - struct holding:
    - ..
    - node number, and
    - pointer to file_operations struct (“fops”).

```
#define MY_DEVICE_FILE  "my_demo_misc"

static struct miscdevice my_miscdevice = {
        .minor   = MISC_DYNAMIC_MINOR,     // Let system assign
        .name    = MY_DEVICE_FILE,         // /dev/.... file.
        .fops    = &my_fops                // Callback functions.
};
```

Misc Data Structures

- file_operations (“fops”)
  - ..
  - Each member in struct is a function pointer; set the member to point to your function.

```
// My functions which I need called to handle file operations
static int my_open(struct inode *inode, struct file *file) { ... }
static int my_close(struct inode *inode, struct file *file) { ... }
static ssize_t my_read(struct file *file, char *buff, size_t count, loff_t *ppos) { ... }

// Set callbacks:  (structure defined in /linux/fs.h)
struct file_operations my_fops = {
    .owner      =  THIS_MODULE,
    .open       =  my_open,
    .release    =  my_close,
    .read       =  my_read,
};
```

Misc Functions

- Register and Unregister (call from your init and exit)
  - misc_register(&my_miscdevice);
  - misc_deregister(&my_miscdevice);

```
static int __init my_init(void)
{
	return misc_register(&my_miscdevice);
}

static void __exit my_exit(void)
{
	misc_deregister(&my_miscdevice);
}
```

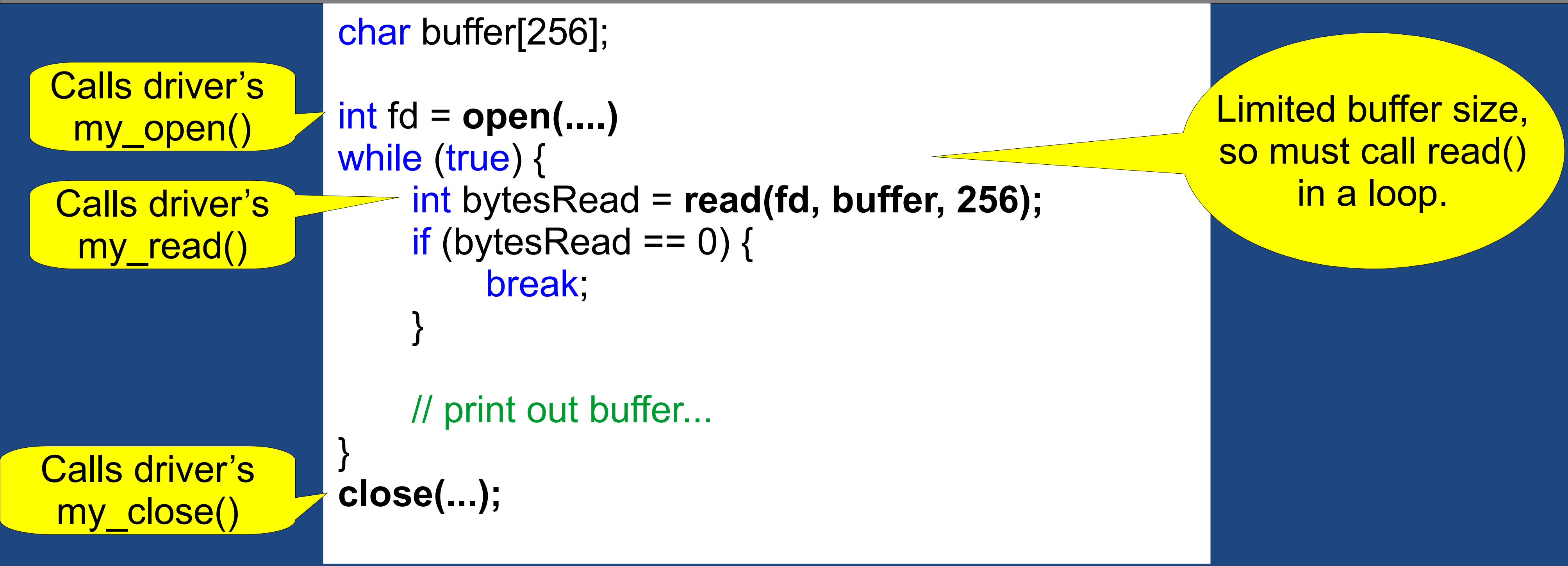
- **Demo:** See demo_misc_template.c

Reading from a Misc Driver’s Virtual File

Recap: Virtual File

- Misc driver creates a node in the file system which is a virtual file.
  - All read and write calls to this node are relayed, by the kernel, to the driver.
  - The driver's file_operations struct links read/write operations on the node to our functions.

User Level: Reading from virtual file



```
char buffer[256];

int fd = open(....)
while (true) {
        int bytesRead = read(fd, buffer, 256);
        if (bytesRead == 0) {
                break;
        }

        // print out buffer...
}
close(...);
```

Calls driver's my_open()

Calls driver's my_read()

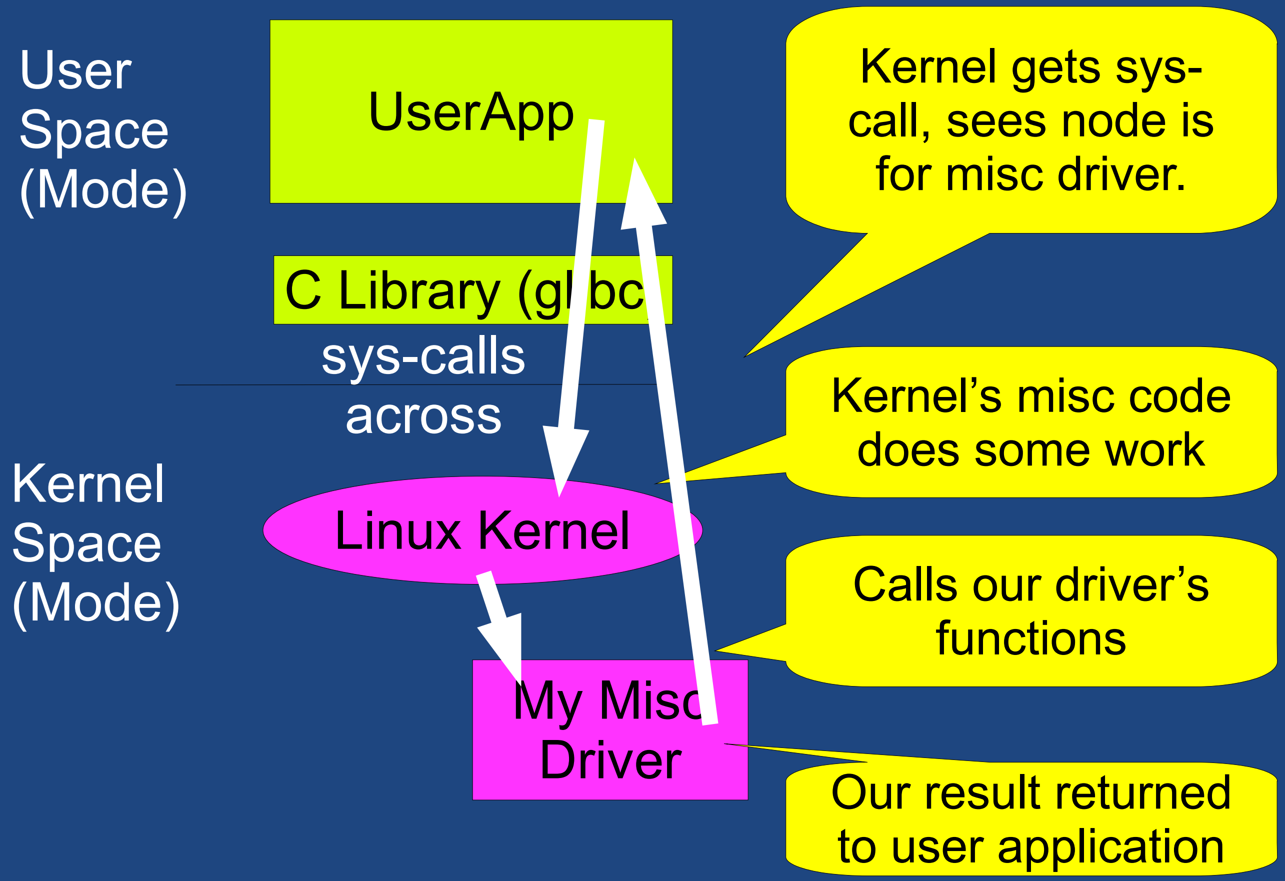
Calls driver's my_close()

Limited buffer size, so must call read() in a loop.

- Notes
  - read() might partially fill buffer.
  - read() returns 0 when done reading all data.
- **Demo:** See 12-ReadFile/readfile.c

  (bbg)$ ./readfile 5 /proc/version

Kernel Level: Reading from virtual file



User Space (Mode)

UserApp

C Library (glibc)

sys-calls across

Kernel Space (Mode)

Linux Kernel

My Misc Driver

Kernel gets sys-call, sees node is for misc driver.

Kernel's misc code does some work

Calls our driver's functions

Our result returned to user application

Kernel Level: Reading from virtual file

Returns # bytes
..

```
int my_read(
        struct file *file,

        char *buffer,

        int count,

        long long *ppos

);
```

Info on the currently open “file”.
Can put own data into struct if desired.

User’s buffer to write into.
!! Pointers from user space not trusted !!

Size of user’s buffer (bytes).

..
Initially set to starting offset in file, we must increment it by as many bytes as we return.

Example reading

- User App: buffer size 5, reads until driver returns 0.
- Driver: has data “AB...Z” to return (string of 26 letters).

Call 1:

ppos @ start	0
fill buffer:	ABCDE
ppos @ end	5
return	5

Call 2:

ppos @ start	5
fill buffer:	FGHIJ
ppos @ end	10
return	5

Call 3:

ppos @ start	10
fill buffer:	KLMNO
ppos @ end	15
return	5

Call 4:

ppos @ start	15
fill buffer:	PQRST
ppos @ end	20
return	5

Call 5:

ppos @ start	20
fill buffer:	UVWXY
ppos @ end	25
return	5

Call 6:

ppos @ start	25
fill buffer:	Z
ppos @ end	26
return	1

Call 7:

ppos @ start	26
fill buffer:	
ppos @ end	26
return	0

Done!

Misc Driver Read Demo

- Edit demo_miscdrv.c
  - When user does:

    (bbg)$ cat /dev/my_misc_demo

    make driver return values in data[ ] array (“ABC...Z”)
  - Solution in demo_miscdrv_sol.c

See demo notes...

HERE BE DRAGONS

Using User Space Buffers

User Level pointers in Kernel Space

- Kernel can access any memory,
  so it can follow any pointer from user space.
- ..
- User’s buffer pointer passed to kernel could be:
  - ..
  - ..
  - ..
  - ..
- Must validate user-level pointers before using them.

Reading From User Buffer

- To read data from user’s buffer:

  ```
  int bytes_not_copied =
      copy_from_user(my_buff, user_ptr, size)
  ```

  - Safely checks user program has permission to access *size* bytes at *user_ptr*.

- Only needed for pointers

  Other values (int’s, char’s, ...) passed by value, so we are not accessing user’s memory space.

- Example

```
if (copy_from_user(my_buff, user_data, 10)) {
    printk(KERN_ERR "Unable to read from buffer.");
    return -EFAULT;
}
```

Writing To User Buffer

- Writing data to user buffer:

  ```
  int bytes_not_copied = copy_to_user(user_ptr, my_buff, size)
  ```

  - returns # bytes not copied

- Define in:

  ```
  #include <linux/uaccess.h>
  ```

- Example

  ```
  char ch;
  if (copy_to_user(&user_buff[idx], &ch, sizeof(ch))) {
        return -EFAULT;
  }
  ```

Demo

- Change demo_miscdrv.c
  - change code in my_read() to write data into user's buffer safely.

Note: Attempting to access user level pointer without these macros gives kernel seg-fault.

Writing to a Misc Driver’s Virtual File

Kernel Level: Reading from virtual file

Returns # bytes
driver read from buffer.

```
int my_write(
        struct file *file,

        const char *buffer,

        int count,

        long long *ppos

);
```

Info on the currently open “file”.
Can put own data into struct if desired.

User’s buffer to read from.
!! Pointers from user space not trusted !!

Size of user’s buffer (bytes).

Offset into the file - **in-out parameter**.
Initially set to starting offset in file, we must increment it by as many bytes as we return.

Open/Close & Write Demo

- Can create own open() & close() functions for your misc driver if you need to.

```
static int my_open(struct inode *inode, struct file *file)
{
        return 0;  // Success
}

static int my_close(struct inode *inode, struct file *file)
{
        return 0;  // Success
}
```

- **Write Demo:** Change demo_miscdrv.c
  - Safely print user’s buffer.
  - Safely find and print minimum ASCII character in user’s buffer.

Summary

- Misc Driver
  - Simplify writing a character driver.
  - file_operations struct connects driver’s functions with misc driver’s code in kernel.
- Virtual file (node)
  - User app reads data from driver via my_read()
  - User app writes data to driver via my_write()
- User Level Pointers
  - Verify all pointers with:
    copy_from_user()
    copy_to_user()