Introduction (Linux Device Drivers, 3rd Edition (www.makelinux.net/ldd3))

Device Drivers -> **DD**

- They are a well defined programming interface between the applications and the actual hardware
- They hide completely the details of how the device works

Users interact with hardware through a set of *standardized calls* that are independent of the specific driver

The device driver (DD) implements these user functions, which translate *system calls* into device-specific operations that act on real hardware

Note that some DD functions are NOT callable by the user but instead act on behalf of the hardware, e.g., *interrupt service routines* (more on this later)

Also, as we already know, the programming interface for DDs in Linux allows them to be built separately as a **module**, and 'plugged in' at runtime.

This simplifies the development/debug of DDs and improves kernel customization capabilities (important for resource constrained systems)

Introduction

Note that DDs should focus on *mechanism*, i.e., the capabilities to be provided by the DD, and NOT on *policy*, on deciding how those capabilites can be used

The focus on mechanism will require that you have an intimate knowledge of the hardware component that will be controlled

A major challenge of DD developement is supporting concurrency, i.e., simultaneous use by multiple processes

With SMP, supporting concurrency has become even more important

The kernel is a large executable in charge of handling a variety of tasks:

- Process management: Creating and destroying processes, providing inter-process communication mechanisms, supporting the notion of concurrency
- Memory management: Implementing virtual memory systems, and providing for dynamic allocation and de-allocation of memory to programs
- Filesystems
- Device control (the topic of this lecture)
- Networking

Introduction

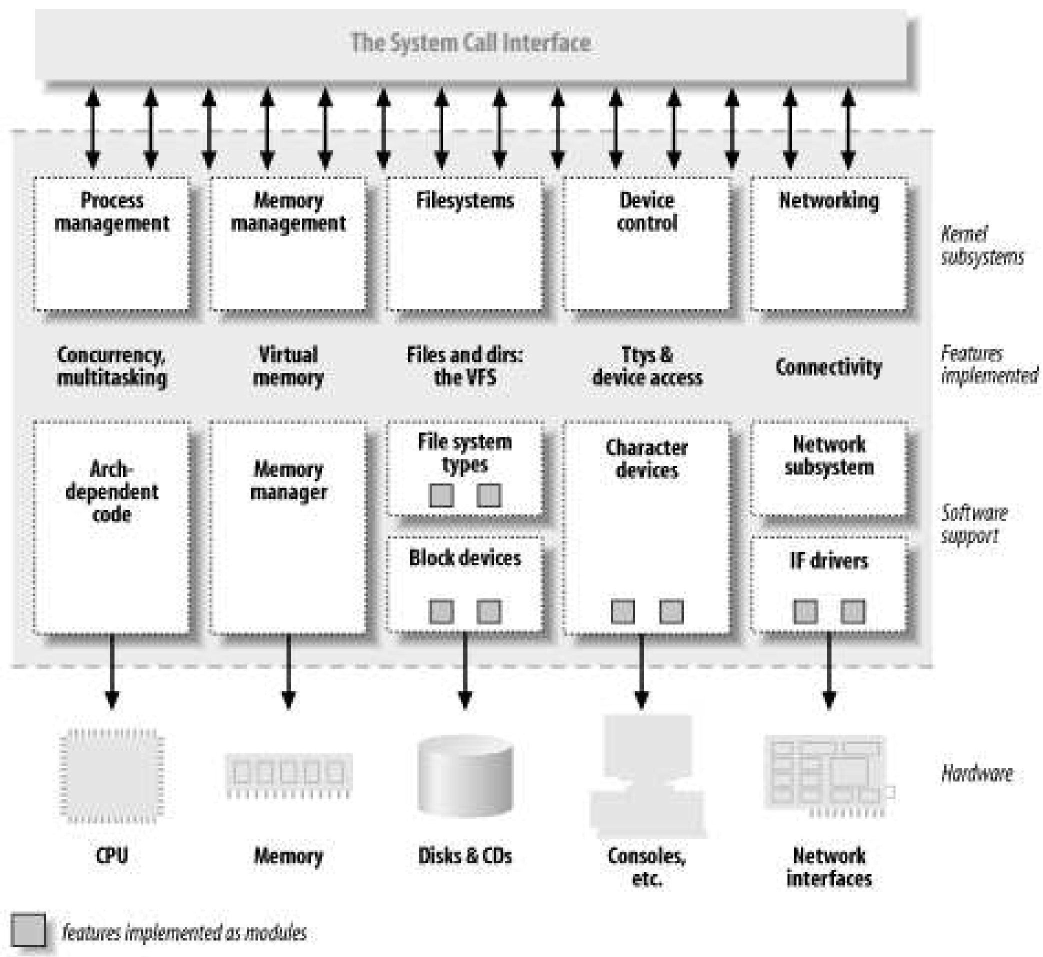
Overview of the kernel

Loadable modules enable functionality to be added at runtime

The figure shows the different classes of modules that can be loaded

You use functions such as *insmod* and *rmmod* to add/remove modules at runtime

Figure 1-1. A split view of the kernel



Introduction

Three basic classes of modules:

- char(acter) device:

  A *char device* is one that can be accessed as a stream of bytes, similar to a file

  Examples include the text console and serial ports

  These devices are accessed using filesystem nodes in /dev

  Unlike files, char devices usually do not allow random movement within the stream

- block device:

  Also represented as filesystem nodes in /dev, but can host a filesystem

  Best example is a hard drive

  Block devices support the transfer of entire blocks of data, e.g., 512 bytes, at a time

  All-in-all, block devices are similar to char devices from the user perspective and differ only in how data is managed internally by the kernel

- network device:

  Are stream oriented devices such as *eth0* but have no entry in /dev

Introduction

Note that some devices, such as USB, can serve multiple roles, e.g., as a char device (a USB serial port), a block device (a USB memory card reader) and as a network device (a USB ethernet interface)

Linux supports other types of modules, including *filesystems*, that layer on top of device based modules

Building and Running Modules

Building modules for 2.6.x requires that you have a configured and built kernel tree on your system

- This is a change from previous versions of the kernel, where a current set of header files was sufficient
- 2.6 modules are linked against object files found in the kernel source tree

Fortunately for us, the Zedboard provides a ideal platform to experiment with modu-lues, without the danger of destroying a Linux installation, e.g., on your laptop

As is traditional, we begin with a 'Hello world' module.

Building and Running Modules

Hello world module:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL"); // Free license

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);
```

Building and Running Modules

This module is about as simple as it gets

It has only two functions:

- *hello_init*: Invoked when the module is loaded into the kernel
- *hello_exit*: Invoked when the module is removed

module_init and *module_exit* functions are special kernel macros that tell the kernel the names of the functions to be used for these two roles

printk is similar to the standard C library function *printf*

This special version is used with DD code b/c DD code does NOT have access to the C library

printk provides for a special indicator string, here *KERN_ALERT*, to indicate the priority of the message

There are a variety of priorities, each with their own unique symbol

Bear in mind that where the *printk* writes its message is dependent on the priority level

Building and Running Modules

To compile and run:

```
% 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#
```

Note that you must have a properly configured and built kernel tree (here it is located at '/usr/src/linux-2.6.10') in order for this to work

Building and Running Modules

Compiling Modules: There are special considerations in the command 'make' for building kernel modules

For the 'hello_world' program, a makefile with the following line is sufficient:

```
obj-m := hello.o
```

This line leverages the extended syntax provided by GNU *make* and states that there is one module to be built, *hello.ko*, from the object file *hello.o*

If you have more than one source file, this expands to:

```
obj-m := module.o
module-objs := file1.o file2.o
```

Type this command from the source directory of the module (change ~/kernel-2.6)

```
make -C ~/kernel-2.6 M=`pwd` modules
```

You can write the 'hello.ko' module to your SD disk (mount it on your host system and copy it there), so it is available when you boot the Zedboard

Building and Running Modules

Important distinction between DDs and user applications:

- Unlike applications, you can only link to functions that are part of the kernel, i.e., never to lib C functions (see header files in 'include/linux' and 'include/asm')
- You must be very digilent in the *exit* function to 'clean up', e.g., de-allocate memory, etc. since it is NOT automatic as it is in applications
- Unlike 'seg faults' in an application, a DD fault can kill the whole kernel
- A module runs in *kernel space* (highest priviledge level), while applications run in *user space* (lowest priviledge level)

Other distinctions:

- Memory in an application is *virtual* and can be swapped out to disk (kernel memory is never 'swappable')
- Most DD functions serve as 'system calls' for applications, which can copy to and from the memory in a *user space* process
- Interrupt service routines (**ISRs**) (also included in DD modules) are 'asynchronous' to processes and are NOT system calls

Building and Running Modules

The most important distinction between user applications and DD (kernel) code -> **CONCURRENCY**

- Most user applications (except multithreading) run from start to finish, and do not need to worry about their environment
- Even the simpliest kernel modules must assume many things can be happening at once
  - Multiple processes may be accessing a DD simultaneously
  - ISRs can be invoked through interrupts, and can execute while other DD functions in the same module are executing
  - The DD functions can be invoked simultaneously on multiple processors (SMP environments)
  - Kernel code (as of release 2.6) is **preemptible**, which makes uniprocessor systems subject to the same issues as multiprocessor systems

This means that DD code must be **reentrant**, i.e., it must be capable of running on behalf of more than one process simultaneously

- For example, data structures must be carefully designed to keep multiple threads of execution separate, and shared data must be protected, e.g., semiphores

Building and Running Modules

Race conditions, i.e., where the results of a computation depend on the relative order of execution, must be avoided when writing DD code

You can no longer assume that your DD functions are not preemptive, i.e., you can not assume that a segment of code in a DD function will execute from start to finish without the possibility of being 'put to sleep'

Concurrency problems are very difficult to debug, so you must learn how to program for concurrency

- There are a set of kernel support functions to assist with concurrency, as we shall see

Char Drivers

The *scull* device (simple character utility for loading localities) is used as an example in the text

`http://www.makelinux.net/ldd3/?u=chp-10-sect-5`

scull controls a memory area as though it was a device, and is hardware independent

Char Drivers

We will look at several variants of *scull*

The first variants are referred to as *scull0* to *scull3*

Each has a memory area that is global (can be opened multiple times with data shared) and persistent (can be opened and closed without loosing contents).

Major and Minor Numbers

Char devices are accessed through 'special' files that are traditionally located in /dev, and have a 'c' as the first character in a long list (*ls -ltra /dev*)

```
crw-rw-rw-    1 root     root       1,   3 Apr 11  2002 null
crw-------    1 root     root      10,   1 Apr 11  2002 psaux
crw-------    1 root     root       4,   1 Oct 28 03:04 tty1
```

The major device numbers in the above listing are 1, 10 and 4 while the minor numbers are 3 and 1.

The major number identifies the DD associated with the device (in rare cases, several drivers may be associated with a major number)

Char Drivers

The minor number is used by the kernel to determine the actual device

You can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices

Within the kernel, the *dev_t* type defines device numbers

Given a *dev_t* or the numbers, you can convert between them in your DD:

```
maj_num = MAJOR(dev_t dev);
min_num = MINOR(dev_t dev);
my_dev_t = MKDEV(int major, int minor);
```

You must setup a char device within your DD using:

```
int register_chrdev_region(dev_t first, unsigned int count,
                           char *name);
```

where *first* is the first number you want to allocate and *name* is the name that shows up in /proc/devices and /sys

There is also an 'alloc_chrdev_region' version that allows the kernel to choose the number

As is almost always the case, the kernel returns '0' if the call is successful

Char Drivers

The directory */proc/devices* is populated first with the device number

If the /dev nodes do not exist, then there is a script (run as root) that can be used to create them. See section 3.2.3 in http://www.makelinux.net/ldd3/?u=chp-10-sect-5

Important Data Structures: *file_operations*, *file*, and *inode*

- *file_operations*:
  - Used to connect device numbers to the DD system call functions
  - It is a structure with fields for function pointers (a jump table) which implement systems calls such as ’open’, ’read’, etc.
  - Each time a device is opened, the kernel creates a *file* structure with a field *f_op*, that points to this *file_operations* structure

The types of system calls supported are provided as a ’function pointers’ in the *file_operations* structure -- here are a few:

```
int (*open) (struct inode *, struct file *);
ssize_t (*read) (struct file *, char _ _user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char _ _user *, size_t, loff_t
*);
```

Char Drivers

scull implements the following functions:

```
struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};
```

THIS_MODULE is a pointer to the module that "owns" the structure

- *struct file*: Usually referred to as *filp* for 'file pointer' in code

  Has no relation to user space *FILE* (which is a C library structure)

  The kernel creates it when the device is first 'opened'

Some of its fields

```
mode_t f_mode;
struct file_operations *f_op;
void *private_data;
```

Char Drivers

- *inode*: Used by the kernel internally to represent *files*

The major and minor numbers can be obtained from *inode* using:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

Character device registration:

The kernel uses structures of type *struct cdev* to represent char devices internally

Before the kernel invokes your device's operations, you must allocate and register one of these structures

First let's cover how *scull* manages its data, using a *struct scull_dev*

```
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */
    int qset;                 /* the current array size */
    unsigned long size;       /* amount of data stored here */
    unsigned int access_key;  /* used by sculluid and scullpriv */
    struct semaphore sem;     /* mutual exclusion semaphore     */
    struct cdev cdev;         /* Char device structure      */
};
```

Char Drivers

The initialization function which interfaces the *scull* device to the kernel:

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
   {
   int err, devno = MKDEV(scull_major, scull_minor + index);

   cdev_init(&dev->cdev, &scull_fops);
   dev->cdev.owner = THIS_MODULE;
   dev->cdev.ops = &scull_fops;
   err = cdev_add (&dev->cdev, devno, 1);
/* Fail gracefully if need be */
   if (err)
      printk(KERN_NOTICE "Error %d adding scull%d", err, index);
   }
```

A kernel *struct cdev* is initialized using:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Once the cdev structure is set up, the final step is to tell the kernel about it:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Here, *dev* is the *cdev* structure, num is the first device number, and count is usually 1

Char Drivers

Notes:

- *cdev_add* can fail with a negative return value (usually doesn't but you need to check)
- When *cdev_add* returns, your device is "live" and its operations can be called by the kernel immediately!

Let's start looking at the system calls registered in *struct file_operations*

- The *open* method

  Preps the driver for later operations

```
int (*open)(struct inode *inode, struct file *filp);
```

The *inode* argument has a pointer to *cdev* structure just setup above, but we really want the *scull_dev* structure that 'contains' a pointer to the *cdev* structure

```
struct scull_dev *dev; /* device information */
dev = container_of(inode->i_cdev, struct scull_dev, cdev);
filp->private_data = dev; /* for other methods */
```

The macro *container_of* returns a pointer to its 'parent' *struct scull_dev*

Char Drivers

The entire *scull_open* routine:

```
int scull_open(struct inode *inode, struct file *filp)
   {
   struct scull_dev *dev; /* device information */

   dev = container_of(inode->i_cdev, struct scull_dev, cdev);
   filp->private_data = dev; /* for other methods */

/* Trim to 0 the length of the device if open was write-only */
   if ( (filp->f_flags & O_ACCMODE) =  = O_WRONLY)
        scull_trim(dev); /* ignore errors */
   return 0;          /* success */
}
```

Not a whole lot is done here b/c *scull* uses memory as the device

- The *release* method

  Deallocate anything that *open* allocated in *filp->private_data and* shut down the device on last close

```
int scull_release(struct inode *inode, struct file *filp)
   { return 0;}
```

Char Drivers

- The *read* and *write* methods

These functions copy data to and from *user space* into memory allocated by *scull*

```
ssize_t read(struct file *filp, char __user *buff,
    size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char __user *buff,
    size_t count, loff_t *offp);
```

For both methods, *filp* is the file pointer and *count* is the size of the requested data transfer

(See Section 3.6 of http://www.makelinux.net/ldd3/?u=chp-10-sect-5 for details on *scull* memory allocation model)

The *buff* argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed

offp is a pointer to a "long offset type" object that indicates the file position the user is accessing

Char Drivers

It is important to realize that *buff* is a *user-space* pointer, and therefore, cannot be directly dereferenced by kernel code

The *buff* pointer may be:
- Invalid or
- The *user-space* page may be in swap (generating a page fault results in an "oops" and the death of the process making the sys call)
- The pointer may be malicious, allowing memory to overwritten anywhere in the system, opening a security hole

Access to *user-space* must be done using kernel-supplied functions:

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

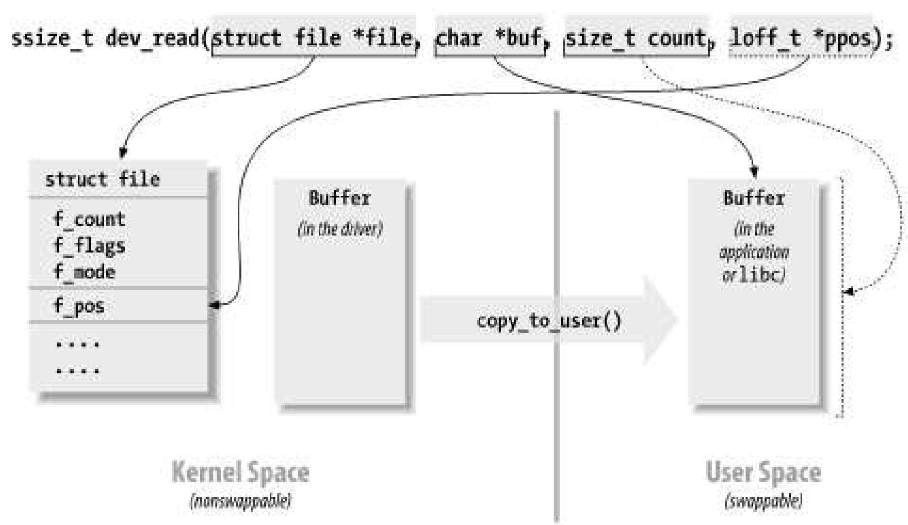
These functions behave like the *memcpy* C library function

Char Drivers

Couple of important notes:

- The *user-space* memory may be in swap, and therefore, the process will be put to sleep, requiring the DD code to be **reentrant** and in a position where it can legally sleep
- These kernel functions also check whether the *user-space* pointer is valid
  - If invalid, no copy is performed
  - If it becomes invalid during the copy, only part of the data is copied and the return value is the *amount of memory still to be copied*

Figure 3-2. The arguments to read



Char Drivers

The *read* and *write* DD methods return -1 if an error occurs (user processes use *errno* to determine the reason), while a value >= 0 tells the *user-space* process how many bytes were successfully transferred

Interestingly, if the value returned by the DD method is >= 0 but not equal to *count*, C library routines, such as *fread* reissue the system call until it succeeds

But what about the case "this is no data, but it may arrive later"

Here, the *read* system call should **block** (covered later)

Interesting things can happen for example if process A is reading the device while process B opens it for writing, which truncates the file to 0

Process A suddenly finds itself past end-of-file and next *read* call returns 0

The *read* and *write* methods are given in the on-line text

With those included, you have a complete driver that can be compiled and run as a module