EMBEDDED DEVICE DRIVERS

Linux Device Drivers on Beaglebone Black

LKM: Device Files

- This course focus:
  - Character devices

- So far
  - We assigned ***major:minor*** numbers
  - But we lack the device “file”
    - So user space can not interact with our device
    - Neither can our device interact with any hardware

- Let’s create a device node / file entry in ***/dev***

LKM: Device node creation

- Device files / nodes enable interaction
  - Between user space and hardware
  - Not normal files, and usually created in ***/dev***

  - Once created, user space programs can
    - open, read, write, close, mmap(), etc.

  - When user space program requests an operation
    - The kernel traps this operation
    - And passes it to our device driver
    - Which then performs the operation
    - On behalf of the user space program

- Device nodes can be created **manually** / **programmatically**

LKM: Node creation (manual)

- Shell command
  *$ **mknod** –m <perms> <dev_name> <dev_type> <major> <minor>*

  - *perms*: permissions on the device
  - *dev_name*: Full path name (/dev/…)
  - *dev_type*: ***c*** for char, ***b*** for block
  - *major, minor*: Numbers assigned to this device

- Example:
  $ sudo mknod –m 0755 /dev/my_device c 202 0

- Note that device node can be created
  - Only by the root user
  - Even before loading the device driver!

LKM: Node creation (auto)

- Programmatic creation of a dev node
  - Follows a hierarchy:
    - Class ➔ Device
  - Actions:
    - Create a class
      - Created under ***/sys/class***
    - Create a device within this class
      - Created in ***sysfs***
- Header files:

  #include <linux/device.h>
  #include <linux/kdev_t.h>

LKM: Node creation API

- Class creation

  *struct class *****class_create****(struct module *owner, const char *name);*

  - *owner*: Who owns this class *(THIS_MODULE)*
  - *name*: String name for this class

- Device creation

  *struct device *****device_create****(struct *class, struct device *parent, dev_t dev, void *drvdata, const char *fmt, …);*

  - *class*: Class under which device is registered
  - *parent*: Parent device in hierarchy
  - *dev*: The *dev_t* for the char device
  - *drvdata*: Data to be added to the device for callbacks
  - *fmt*: String of the device name

LKM: Node cleanup API

- Class destroy

  *void **class_destroy**(struct class *cls);*

- Device destroy

  *void **device_destroy**(struct class *cls, dev_t dev);*

LKM: Device node exercise

- Refer ***mod3*** directory
  - File ***mod33.c*** contains the src code
    - We seek allocation for a *major:minor* from kernel
    - We create a device class ***(cdac_cls)***
    - We create a char device under this class ***(cdac_dev)***
    - Compile, load the module
    - Observe the dmesg output to get ***major:minor*** allocation
    - Also check the entries created:

      $ ls –l /sys/class | grep cdac

      $ ls –l /sys/dev/char/ | grep cdac

      $ ls –l /dev | grep cdac
    - Unload the module

LKM: File Operations

- So far, for our char device, we have managed to
  - Assign ***major:minor*** (static / dynamic)
  - Create class and device nodes
  - But how do we use this device?
    - Through file operations
    - Since the device node is basically a “file” under Linux

- To perform file operations
  - *(**fops** in kernel-speak)*
  - We need to register structures
    - That tell the kernel what to do
    - When a particular file operation needs to be performed
  - Most driver operations use:
    - ***struct cdev, struct file_operations, struct inode***

LKM: struct cdev

- Internal structure
  - Used by kernel
  - For char devices
- Relevant elements
  - *owner*
    - *THIS_MODULE*
  - *ops*
    - For file operations

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

struct cdev {
    …
    struct module *owner;
    const struct
file_operations *ops;
    …
};
```

LKM: struct cdev API

- Allocation
  - Embedded

    *void **cdev_init**(struct cdev *cdev, struct file_operations *fops);*
  - Runtime

    *struct cdev *my_cdev = **cdev_alloc**();*
    my_cdev->ops = &my_fops;

- Registering with kernel

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

  - *cdev*: cdev structure
  - *dev*: major:minor holder
  - *count*: No of devices to associate with the device

- De-registering from the kernel

  *void **cdev_del**(struct cdev *cdev);*

LKM: struct file_operations

```
#include <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 *);
    [...]
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    [...]
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    [...]
```

- Has function pointers
  - For registering various operations
  - On char devices
- Notice APIs differ from system calls
  - Kernel mediates
    - Between user and device driver
    - Simplifies user space calls

LKM: File perspectives

- We have 2 structs for files
  - ***struct inode***
    - Represents a file from the file-system viewpoint
    - Attributes: size, rights, access/modification times
    - Uniquely identifies a file in a file-system
    - *Used to get the major:minor of the device*
  - ***struct file***
    - Represents a file from the user's viewpoint
    - Attributes: inode, file-name, opening options, position
    - All open files have an associated *struct file*
    - *Used to determine flags and for operations on the file*

LKM: On file_operations (1/2)

- Owner
  - ***THIS_MODULE***
  - Tells the kernel not to unload the module when an operation is in progress

- Open

  *int (****open***)(struct inode *, struct file*);*
  - Usually, the first operation performed on the device
    - But not mandatory – in which case device open always succeeds

- Read

  *ssize_t (****read***)(struct file *, char __user *, size_t, loff_t *);*
  - Used to retrieve data from the device
    - Non-negative return value indicates bytes read
    - Negative return value indicates error

LKM: On file_operations (2/2)

- Write

  *ssize_t (****write****)(struct file *, const char __user *, size_t, loff_t *);*

  - Used to send data to the device
    - Non-negative return value indicates bytes written
    - Negative value indicates error

- Release (close)

  *int (****release****)(struct inode *, struct file *);*

  - Usually called when device is closed

- IOCTL

  *long (****ioctl****)(struct file *, unsigned int, unsigned long);*

  - Used to issue device specific commands that are not data specific

LKM: fops exercise (1/2)

- Refer ***mod3*** directory
  - File ***mode34.c*** contains the src code
    - Here, we create a char device ***(/dev/cdac_dev)*** like in ***mod33***
    - We create a ***fops*** using our own file operations
    - We create a ***cdev*** and register it with the kernel using the ***fops***

    - Compile and load the module on BBB
    - Once the module has loaded
      - Test the read part

        # cat /dev/cdac_dev
        - Observe *dmesg*
      - Test the write part

        # echo 123 > /dev/cdac_dev
        - Observe *dmesg*

LKM: fops exercise (2/2)

- Refer ***mod3*** directory
  - File ***mode34_app.c*** contains the src code
    - User space code to test the char device driver
      - We open the device (***/dev/cdac_dev***)
      - Do a write
      - Do a read

      All using system calls
    - Compile the .c file on BBB
    - Notice the output of *dmesg* when you run **a.out**
      - When module is unload
      - After module is loaded

LKM: Kernel memory allocation

- Kernel has its own memory space
  - Shared with scheduler, device drivers, etc.

- Kernel manages its own memory space
  - Using kernel-level memory allocators
  - Cannot use C library <stdlib.h> *(malloc, calloc, free)*

- Kernel memory management APIs
  - *kmalloc()*
  - *kfree()*

LKM: Memory mgmt. APIs

- Allocation

  *void *__kmalloc__(size_t size, gfp_t flags);*

  - *size*: Bytes of memory requested
  - *flags*: Type of memory needed
    - *GFP_USER: Alloc on behalf of user, may sleep*
    - ***GFP_KERNEL**: Alloc kernel RAM, may sleep*
    - *GFP_ATOMIC: No sleep, use emergency pools*
    - *GFP_NOIO: No IO while allocating memory*
    - *__GFP_THISNODE: Alloc on this node only*
    - *GFP_DMA: Alloc memory to used for DMA*

- Freeing

  *void **kfree**(const void *objp);*

  - *objp*: pointer returned by *kmalloc()*

- Header file

  #include <linux.slab.h>

LKM: Interactions – user/kernel

- In many applications
  - User space programs
  - Need to interact with
  - Kernel space code (like modules / device drivers)

  - Data needs to be exchanged between the spaces
    - From user to kernel
    - From kernel to user

- Linux provides specific APIs for these interactions
  - Header file

    #include <linux/uaccess.h>

LKM: Userspace → kernelspace

- Copy memory from user space to kernel space

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

  - *to*: Destination in kernel space
  - *from*: Source in user space
  - *n*: Bytes to copy

  - Returns:
    - No of bytes that **could not be copied**
    - So success is 0!

LKM: Kernelspace → Userspace

- Copy memory from kernel space to user space

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

  - *to*: Destination in user space
  - *from*: Source in kernel space
  - *n*: Bytes to copy

  - Returns:
    - No of bytes that **could not be copied**
    - So success is 0!

LKM: Interaction b/w spaces

- We now look at passing data
  - From user space to kernel space
  - And vice versa
- This needs 2 pieces of code
  - Device driver
    - To emulate a dummy device
    - Accepts data from user space and passes it back
  - User space program
    - Regular code using systems calls
    - Interacts with device driver

LKM: Space interaction exercise

- Refer ***mod3*** directory
  - The file ***mod35.c*** is the driver src code
    - Creates a character device as before
    - Read is modified to accept user space buffer
    - Write is modified to write to user space buffer
    - Compile and load the module on BBB
  - The file ***mod35_app.c*** is the user space program
    - We open the device ***(/dev/cdac_dev)***
    - We first write to it; then we read from it
    - Compile the .c file on BBB to get ***a.out***
    - Observe the ***dmesg*** output when you run ***a.out***
      - With module loaded
      - Without module loaded

THANK YOU!