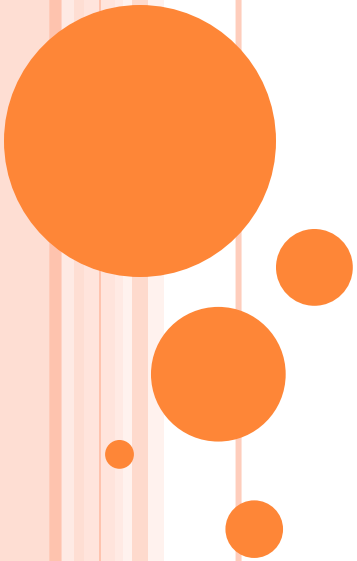


Linux Device Drivers (Character Drivers) C-DAC, Mumbai

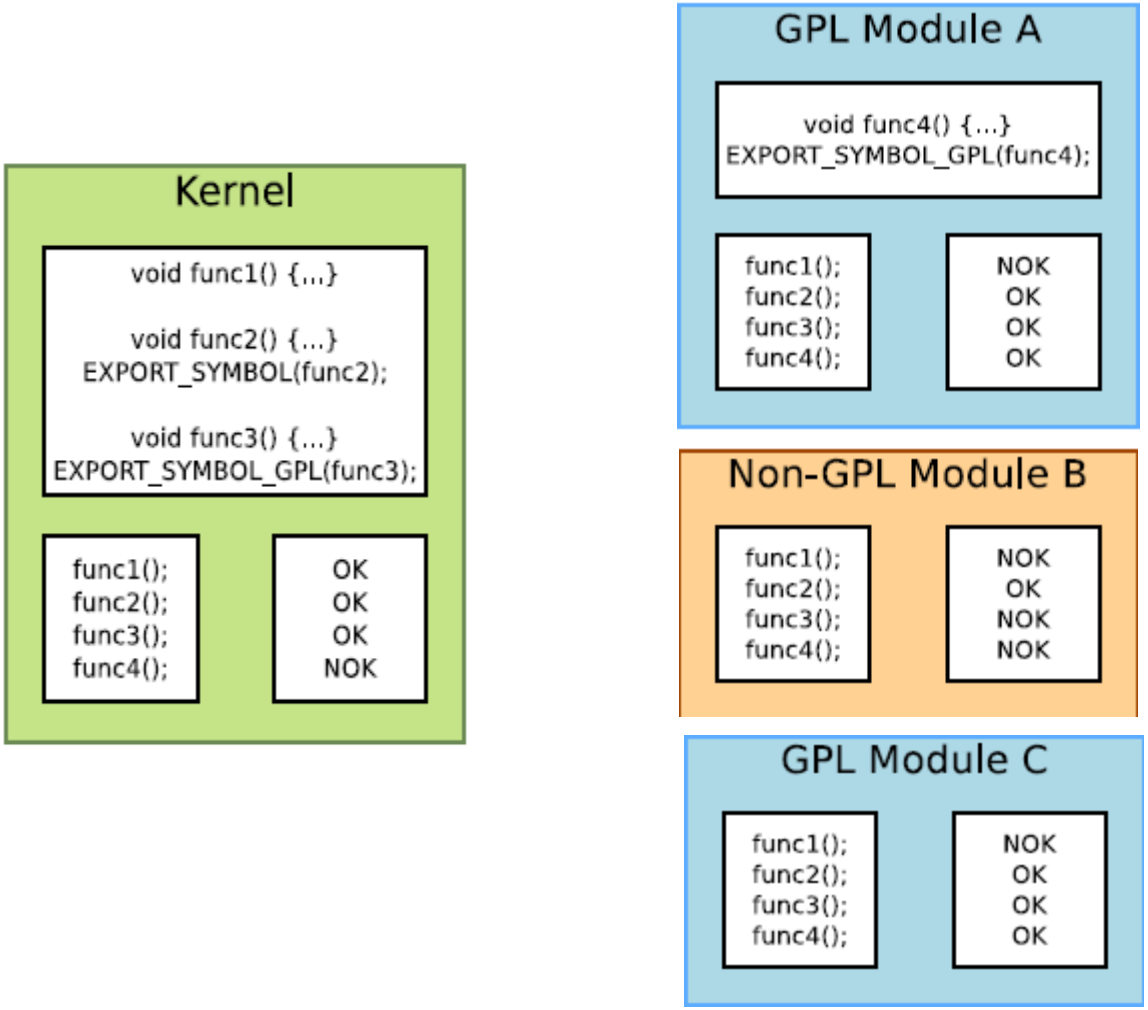
Agenda

- Exporting Symbols
- Module Parameters
- Major and Minor Numbers
- Important Data Structures
- Character Device Registration
- The *open Method*
- The *release Method*
- The *read Method*
- The w*rite Method*
- The Current Process

Kernel Symbol Table

- When a module is loaded, any symbol is exported by the module becomes part of the kernel symbol table
- New modules can use symbols exported by your modules, and you can stack new modules on top of other modules.
- Using Stacking to split modules into multiple layers can help reduce development time by simplifying each layer
- Functions and variables have to be explicitly exported by the kernel to be visible from a kernel module
- Two macros are used in the kernel to export functions and variables:
- **EXPORT_SYMBOL(symbolname)**, which exports a function or variable to all modules
- **EXPORT_SYMBOL_GPL(symbolname)**, which exports a function or variable only to GPL modules

Symbol Exported to Modules



Module Parameters

```
#include <linux/moduleparam.h>
module_param(
name, /* name of an already defined variable */
type, /* either byte, short, ushort, int, uint, long, ulong,
charp, or bool.(checked at compile time!) */
perm
/* for /sys/module/<module_name>/parameters/<param>,
0: no such module parameter value file */
);
/* Example */
int irq=5;
module_param(irq, int, S_IRUGO);
```

Modules parameter arrays are also possible with

```
module_param_array(name,type,num,perm);
```

but they are less common.

Hello Module With parameters

```
/* hello_param.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
MODULE_LICENSE("GPL");
/* A couple of parameters that can be passed in:
   how many times we say hello, and to whom */
   static char *whom = "world";
   module_param(whom, charp, 0);
   static int howmany = 1;
   module_param(howmany, int, 0);
```

Hello Module With parameters

```
static int __init hello_init(void)
{
    int i;
    for (i = 0; i < howmany; i++)
    printk("(%d) Hello, %s\n", i, whom);
    return 0;
}
static void __exit hello_exit(void)
{
    printk("Goodbye,  %s\n", whom);
}
module_init(hello_init);
module_exit(hello_exit);
```

Major and Minor Numbers

- Devices are referred to by names in the filesystem.
- These names are called device files or simply nodes in the filesystem.
- They find their existence in the */dev directory of the* filesystem.
- They provide an interface to a device from the user space and is common to the kernel space to access the device too.

Major and Minor Numbers

- On issuing an ls -al on the */dev directory, the following is* listed

```
crw-rw---- 1 root uucp 33, 3 Mar 20 2005 cux3
crw-rw---- 1 root uucp 33, 4 Mar 20 2005 cux4
crw------- 1 root root 10, 193 Mar 20 2005 d7s
brw-r----- 1 root disk 12, 0 Mar 20 2005 dos_cd0
brw-r----- 1 root disk 12, 1 Mar 20 2005 dos_cd1
brw-rw---- 1 root disk 14, 0 Mar 20 2005 dos_hda
lrwxrwxrwx 1 root root 8 Oct 4 15:38 dmdsp0 -> /dev/dsp
```

- The ‘c’ at the beginning of the file attributes specifies that the device is a character device. The ‘b’ indicates that the device is a block device. ‘l’ indicates a link to a device file present at an other location.

Major and Minor Numbers

- brw-r----- 1 root disk 12, 0 Mar 20 2005 dos_cd0
- In the listing shown above, the two numbers 12,0 (bold) are the major and minor numbers respectively.
- The major number identifies the driver associated with the device. Multiple drivers may share major numbers
- For example, a major number may identify a driver for a hard disk.

Major and Minor Numbers

- Minor Numbers are used by the kernel to determine exactly which device is being referred to. The minor numbers could serve as an index into a list of devices for the related driver.
- For example, a minor number may be used to refer to one of the 4 hard disks present in the system.
- Both the major number and minor number together are called Device Numbers

Internal representation of Device Numbers

- Within the kernel, the dev_t type is used to hold device numbers. dev_t is a 32 bit quantity with 12 bits set aside for the major number and 20 bits set aside for the minor number
- To obtain the major and minor parts of a dev_t, macros come in handy
  - MAJOR(dev_t dev)
  - MINOR(dev_t dev)
- Instead, to create a device number, use
  - MKDEV (int major, int minor)

User space access to a Device Number

- Applications in the user space can refer to a device by creating a device file or node in the /dev/ directory of the filesystem.
- This is possible by assigning a name to a device number using the shell command
  - mknod name_device device_type major minor
  - eg: mknod /dev/MyCharDevice c 253 15
- In the above example, ‘c’ specifies that the device is a character device, 253 represents the major number and 15 represents the minor number. MyCharDevice refers to the name associated with the driver at user space.

Allocating and Freeing Device Numbers

- To obtain a char device a device number has to be registered for it. This is done by

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

- Here first refers to the beginning of the device number range. The count refers to the contiguous number of device numbers required. The name is the representation of the character device in the /proc/devices
- As with most kernel functions, the return value is zero on successful registration and is negative otherwise.

Allocating and Freeing Device Numbers

- In order to allocate a device number without prior knowledge of the same, the following function may be used

  ```
  int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
  unsigned int count, char *name);
  ```

- The arguments remain the same but for the initial argument that is the device number that is alloted to the device. The argument firstminor refers to the first minor number requested. This is normally set to 0.
- These functions are usually called in the init_module section of the driver

Allocating and Freeing Device Numbers

- Device numbers should be freed when no more in need. This is possible in calling the function

  void unregister_chrdev_region(dev_t first, unsigned int count)

- Device numbers should be freed in the module's cleanup_module.
- Before a user-space program can access one of the device numbers, the driver needs to connect them to its internal functions and implement those operations.

Important Data Structures

File Operations:

- This is a structure of function pointers, each of which implements a particular method that defines the functionalities provided by that driver
- The operations include some functionalities such as open, read, write, close, ioctl etc.,
- struct file_operations

```
{
.owner = THIS_MODULE,
.read = MycharDev_read,
.write = MyCharDev_write,
.open = MyCharDev_open,
.release = MyCharDev_close,
.ioctl  = MyCharDev_ioctl,
}
```

Important Data Structures

- **file:** This structure represents an open file. It is used to control the operations of the driver.
- It is similar to the FILE structure in the user space, but has nothing to do with the FILE structure of the user space
- It is created by the kernel in the open system call and exists until the close function is executed. After all instances of the file are closed, the kernel releases the data structure
- It contains a pointer to the file operations structure created in the driver

Important Data Structures

- **Some of the important parameters of this structure are**
- **mode_t f_mode** -Identifies the file as either readable or writable
- **loff_t f_pos** -The current reading or writing position
- **struct file_operations *f_op** -The operations associated with a file. The kernel assigns the pointer as a part of its implementation of open and then reads it when it needs to dispatch any operations.
- **void *private_data** -this pointer can be used to point to allocated data, but it must be freed before the file structure is destroyed

Important Data Structures

- **Inode Structure:** The inode structure is internally used by the kernel to represent files. It is entirely different from the file structure
- There can be many file structures associated with a file but they all point to a single inode structure for that file
- The inode structure holds two important fields with respect to drivers
  - dev_t irdev;
    - Used to hold the actual device number
  - struct cdev *i_cdev;
    - Kernels internal structure that represents char devices

Character Device Registration

- The kernel uses structures of type struct cdev to represent character devices internally
- There are two ways of allocating and initializing one of these structures

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

OR

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
my_cdev->ops = &myfops;
```

- Once the structure has been set up, the kernel is notified with the call to

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

- To remove a character device

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

The *open Method*

- The open method is provided for a driver to perform any initialization in preparation for later operations.
- Check for device specific errors, Initialize the device if it is being opened for the first time
- Update the f_op pointer, if necessary Allocate and fill any data structure to be put in

  filp->private_data

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

The *release* Method

- As the name signifies, this call is used to release access to the requested device. The common functions that are performed in this method are
  - Deallocate anything that the open allocated
  - Shut down the device on last close

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

The *read Method*

- Interacts with the user space application program and implements the read functionality for the user space

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

- If the return value equals the count, the requested number of bytes have been transferred.
- If it is positive but less than the count, only part of the data has been transferred. 0 indicates EOF. A negative value indicates error.

The *write Method*

- **Similar to the read method.**
  - ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *f_pos);

Read and Write Methods

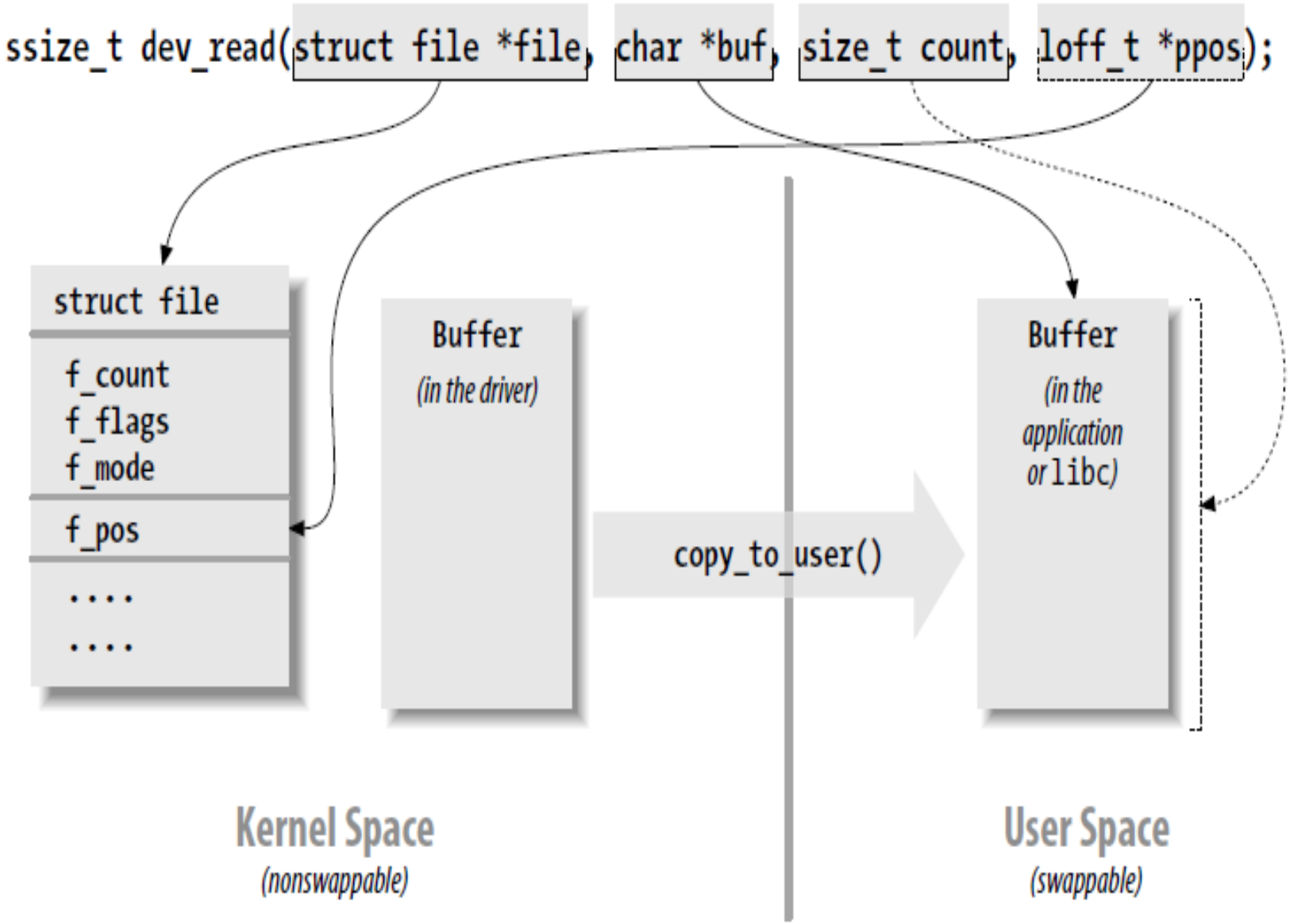
- buff argument to the *read and write methods is a user-space* pointer. Therefore, it cannot be directly dereferenced by kernel code.
- There are a few reasons for this restriction:
  - the user-space pointer may not be valid while running in kernel mode at all.
  - Even if the pointer does mean the same thing in kernel space, user-space memory is paged, and the memory in question might not be resident in RAM when the system call is made.
  - The pointer in question has been supplied by a user program, which could be buggy or malicious. If your driver ever blindly dereferences a user-supplied pointer, it provides an open doorway allowing a user-space program to access or overwrite memory anywhere in the system.

COPY_TO_USER AND COPY_FROM_USER

- Obviously, your driver must be able to access the user-space buffer in order to get its job done. This access must always be performed by special, kernel-supplied functions, however, in order to be safe.

  (which are defined in *<asm/uaccess.h>*)
- 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);

- The role of the two functions is not limited to copying data to and from user-space: they also check whether the user space pointer is valid. If the pointer is invalid, no copy is performed; if an invalid address is encountered during the copy, on the other hand, only part of the data is copied. In both cases, the return value is the amount of memory still to be copied. The *scull code looks for this error return, and returns -EFAULT to* the user if it's not 0.


ssize_t dev_read(struct file *file, char *buf, size_t count, loff_t *ppos);
struct file
f_count
f_flags
f_mode
f_pos
....
....
Buffer
(in the driver)
copy_to_user()
Buffer
(in the application or libc)
Kernel Space
(nonswappable)
User Space
(swappable)

IOCTL (I/P & O/P CONTROL)

- Most drivers need, in addition to the ability to read and write the device, the ability to perform various types of hardware control via the device driver.
- These operations are normally supported via the ioctl method
- In the user space, the ioctl command has the following format
  - int ioctl(int fd, unsigned long cmd, ...);
- The ioctl driver method has the prototype
  - int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);

Magic Numbers

- Magic numbers are mechanisms of identifying the commands for a particular device. They must be unique over the system. These are maintained by the kernel in 4 bit-fields
- **Type**
  - This is the magic number present in the file ioctl-number.txt. It is 8 bits wide
- **Number**
  - The ordinal (sequential) number. It is also 8 bits wide
- **Direction**
  - The direction of data transfer. Two bits
- **Size**
  - The size of the user data involved. The size is architecture dependent, and is generally limited to 13 or 14 bits

Some Important Macros

- _IO(type,nr) (command that has no Argument)
- _IOR(type,nr,datatype) (reading data for driver)
- _IOW(type,nr,datatype) (for Writing Data)
- _IOWR(type,nr,datatype) (for Bi-directional Transfer)

- _IOC_DIR(nr)
- _IOC_TYPE(nr)
- _IOC_NR(nr)
- _IOC_SIZE(nr)

Some Important Functions

- int access_ok(int type,const void* addr, unsigned long size)
- First argument should be either VERIFY_READ or WRITE
  - Returns 1 on Success or 0 on Failure

- put_user(datum,ptr)
- __put_user(datum,ptr)
- get_user(local,ptr)
- __get_user(local,ptr)
- Return 0 on Success or –EFAULT on failure

Wait Queues

- Wait Queues are mechanisms of putting a user space process into a sleep whenever the kernel driver is not able to suffice the user process's requirements.
- When a process is put to sleep, it is marked as being in a special state and removed from the scheduler's run queue. The process will not be scheduled unless an event causes the scheduling.
- The linux scheduler maintains two special states that represent a wait state. They are defined as
  - TASK_INTERRUPTIBLE and
  - TASK_UNINTERRUPTIBLE

When to use wait Queues

- There are two behaviours that warrant the use of wait queues
  - If a process calls *read but no data is available, the* process must block. The process is awakened as soon as some data arrives, and that data is returned to the caller, even if there is less data than the amount requested in the count argument to the method.
  - If a process calls *write and there is no space in the* buffer, the process must block, and it must be on a different wait queue from the one used for reading. When some data has been written to the hardware device, and space becomes free in the output buffer, the process is awakened and the write call succeeds, although the data may be only partially written if there isn't room in the buffer for the count bytes that were requested

Wait Queues

- Wait queue is managed by means of a structure wait_queue_head_t which is defined in linux/wait.h
- Static way initialisation

  DECLARE_WAIT_QUEUE_HEAD(name);
- Dynamic way initialisation

  wait_queue_head_t my_queue;

  init_waitqueue_head(&my_queue);

Wait Queues

- Sleeping in linux kernel

  wait_event(queue,condition);

  wait_event_interruptible(queue,condition);

  wait_event_timeout(queue,condition,timeout);

  wait_event_interruptible_timeout(queue,condition,timeout);

- Waking up in linux kernel

  void wake_up(wait_queue_head_t *queue);

  void wake_up_interruptible(wait_queue_head_t *queue);