Char Drivers

Char Drivers

- Major and Minor Numbers
  - Major Number
    typically indicates the family of the device.
  - Minor Number
    allows drivers to distinguish the various devices they manage.
  - ***dev_t*** holds device numbers,both the major and minor parts.
    a **32-bit** quantity with **12** bits for major number and **20** for the minor number.
  - Dynamic allocation of Major Numbers
    already what used for ***kernelnewbies.ko***,
    however there was a static way.
  - Dynamic Allocation is preferred over static, as
    a randomly picked major number will lead to conflicts
    and trouble if this driver widely deployed.

```
root@karimeshapa-Inspiron-5537:/home/karimeshapa/ldd/kernelnewbies# ls -al /dev/kernelnewbies
crw------- 1 root root 511, 0 23:07 18 س /dev/kernelnewbies
root@karimeshapa-Inspiron-5537:/home/karimeshapa/ldd/kernelnewbies#
root@karimeshapa-Inspiron-5537:/home/karimeshapa/ldd/kernelnewbies#
```

/include/linux/types.h

```
13  typedef u32 __kernel_dev_t;
14
15  typedef __kernel_fd_set     fd_set;
16  typedef __kernel_dev_t      dev_t;
```

/include/linux/kdev_t.h

```
7   #define MINORBITS   20
8   #define MINORMASK   ((1U << MINORBITS) - 1)
9
10  #define MAJOR(dev)  ((unsigned int) ((dev) >> MINORBITS))
11  #define MINOR(dev)  ((unsigned int) ((dev) & MINORMASK))
12  #define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

Char Drivers

- Major and Minor Numbers, *Cont'd*
  - The kernel uses ***chrdevs*** global variable to manage device number's allocation.
  - ***alloc_chrdev_region()*** iterates ***chrdevs*** from last and find and empty entry to return as the major number.
- Char Device Registration
  - Allocation APIs
    alloc_chrdev_region() : register a range of char device numbers.
    class_create() : create a struct class to your device.
    device_create() : creates a device and registers it with sysfs.
    cdev_init() : initialize a cdev structure with fileops,making it ready to add to the system with ***cdev_add()***.
    cdev_add() : add a char device to the system.
  - Remove APIs
    cdev_del(), device_destroy(), class_destroy(), unregister_chrdev_region()

/fs/char_dev.c

```
34  static struct char_device_struct {
35      struct char_device_struct *next;
36      unsigned int major;
37      unsigned int baseminor;
38      int minorct;
39      char name[64];
40      struct cdev *cdev;      /* will die */
41  } *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

Char Drivers

- Advanced Operations

  - Ioctl
    user-space needs to control the device
    i.e. eject media, change a baud rate, report error information,…
    ioctl numbers can be any numbers however it's better
    to follow the kernel convention ***/include/uapi/asm-generic/ioctl.h***

  - Sleeping
    sleeping process is accomplished through a data structure called a **wait queue**.
    wait_queue_head_t my_queue;
    init_waitqueue_head(&my_queue);
    wait_event(queue, condition);
    wait_event_interruptible(queue, condition);
    wait_event_timeout(queue, condition, timeout);
    wait_event_interruptible_timeout(queue, condition, timeout);
    void wake_up(wait_queue_head_t *queue);
    void wake_up_interruptible(wait_queue_head_t *queue);

user-space
sys/ioctl.h

```
int ioctl(int fd, unsigned long cmd, ...);
```

Char Drivers

- Advanced Operations, *Cont'd*

  - Poll and select
    block a process until any of a given set of file descriptors becomes available for reading or writing.

    unsigned int (*poll) (struct file *filp, poll_table *wait);

    The driver adds a wait queue to the ***poll_table*** structure by calling the function ***poll_wait()***

```c
static unsigned int chardrvs_poll(struct file *filp, poll_table *wait)
{
    struct chardrvs_dev *dev = (struct chardrvs_dev *)filp->private_data;
    unsigned int mask = 0;

    mutex_lock(&dev->lock);
    poll_wait(filp, &dev->wq_f, wait);
    mutex_unlock(&dev->lock);
    if (dev->avail)
        mask |= POLLOUT;
    return mask;
}
```

Char Drivers

- Sequence File Interface

  - The ***seq_file*** interface assumes that you are creating a virtual file that steps through a sequence of items that must be returned to user space.

  - Over time, ***/proc*** methods have become notorious for buggy implementations when the amount of output grows large.

  - As a way of cleaning up the ***/proc*** code and making life easier for kernel programmers, the ***seq_file*** interface was added.

  - The kernels implements major functionality to be used by driver writer in order to ***open, read, seek, release,…*** to deal with ***/proc*** created files.

  - ***Why sequesnce files?***
    Implementations of **proc** files had an awkward limitation of not being able to “print” more than a single **page** of output.

    Sequence files solve this problem by generating the “output” of a **proc** file as a sequence of writes, each of which can be up to a page in size, with no limit on the number of writes.

```
static int mydrv_open(struct inode *inode, struct file *file)
{
    return single_open(file, mydrv_show, NULL);
}

static struct proc_ops myproc_fops = {
    .proc_open    = mydrv_open,
    .proc_read    = seq_read,
    .proc_lseek   = seq_lseek,
    .proc_release = single_release,
};
```