Operating System Design

Chapter 13:
Managing I/O Devices

Serial No 24
Nitish Rawat (M110248CS)
National Institute Of Technology Calicut

Outline

- Device Drivers
- Levels of Kernel Support
- Buffering Strategies of Device Drivers
- Registering a Device Driver
- Initializing a Device Driver
- Monitoring I/O Operations
- Block Device Drivers
- Initializing a Block Device Driver
- Buffer Heads

VFS Handling of Device Files

- Device files live in the system directory tree but are intrinsically different from regular files and directories.
- It is the VFS's responsibility to hide the differences between device files and regular files from application programs.
- VFS changes the default file operations of a device file when it is opened. The device-related function acts on the hardware device to perform the requested operation.

Device Drivers

- It is a software layer that makes hardware device respond to programming interface.
- The actual implementation of all VFS functions (*open*, *read*, *lseek*, *ioctl*, and so forth) is delegated to the device driver.
- Before using a device driver, two activities must have taken place: **registering the device driver** and **initializing the device driver**. Finally, it must also **monitor the I/O operation**.

Types of Device Drivers

- Types of Drivers differ in
  - the **level of support** that they offer to the User Mode applications,
  - as well as in their **buffering strategies** for data collected from the hardware devices.

Levels of Kernel Support

Three possible kinds of support for a hardware device:

- No support at all – eg. X Window System
- Minimal support
- Extended support – eg. Internal hard disk

Buffering Strategies of Device Drivers

- Input data is read one character at a time from the device input register and stored in a proper kernel data structure; the data is then copied at leisure into the process address space. Similarly, it is done for output data.
- Device driver must be able to cope with data, even when CPU
  - Use of the DMA processor (DMAC) to transfer blocks of data.
  - Use of circular buffer of two or more elements.

Registering a Device Driver

- Registering a device driver means linking it to the corresponding device files.
- Accesses to device files whose corresponding drivers have not been previously registered return the error code ***-ENODEV***.
- Statically compiled device driver – registration performed during kernel initialization phase.
- Conversly – Compiled as a kernel module and can also unregister itself when module is inloaded

Registering a Device Driver

- Character device driver
  - Described by a `chrdevs` array of `device_struct` data structures.
  - Each structure includes two fields:
    - `name` points to name of the device
    - `fops` points to a `file_operations` structure.

Registering a Device Driver

- Block device driver
  - Described by a `blkdevs` array of 255 data structures.
  - Each structure includes two fields:
    - `name` points to name of the device
    - `bdops` points to a `block_device_operations` structure.
- The `register_blkdev()` functions insert a new entry into one of the tables eg:
  - `register_chrdev(6, “lp” , &lp_fops);`

Methods of Block device drivers

Method	Event that triggers the invocation of the method
open	Opening the block device file
release	Closing the last reference to a block device file
ioctl	Issuing a ioctl() system call on the block device file
check_media_change	Checking whether the media has been changed
revalidate	Checking whether the block device holds valid data

Initializing a Device Driver

- Initializing a driver means allocating precious resources of the system, which are therefore not available to other drivers.
- The assignment of IRQs to devices is usually made dynamically, right before using them, since several devices may share the same IRQ line.
- Other resources possible are page frames for DMA transfer buffers and the DMA channel itself.

Device Driver resource schema

- A usage counter is increamented in the open method of the device file and decremented in the release method.
- The open method checks the value of counter. If it is null, the device driver must allocate the resources and enable interrupts.
- The release method also checks the value of counter. If it is null, the method disables interrupts and releases allocated resources.

Monitoring I/O Operations

- The device driver that started an I/O operation must rely on a monitoring technique that signals either the termination of the I/O operation or a time-out. The techniques available are called
  - Polling mode
  - Interrupt mode

Monitoring I/O Operations

- In case of a **terminated operation**, the device driver reads the status register of the I/O interface to determine whether the I/O operation was carried out successfully.
- In case of **time-out**, the maximum time interval allowed to complete the operation elapsed and nothing happened.

Polling mode

- The CPU checks (polls) the device's status register repeatedly until its value signals that the I/O operation has been completed.
- An example of polling looks like the following:

```
for (;;) {
        If (read_status(device) &
           DEVICE_END_OPERATION) break;
        If (--count == 0) break;
}
```

Interrupt mode

- It is preferable to implement the driver using the interrupt mode where the device driver doesn't know in advance how much time it has to wait for an answer from the hardware device.
- The driver includes two functions:
  - The `foo_read()` that implement the read method of the file object.
  - The `foo_interrupt()` function that handles the interrupt

Interrupt mode

- The main operations of the `foo_read()` function are the following:
  - Acquires the `foo_dev->sem` semaphore, thus ensuring the no other process is accessing the device.
  - Clears the `intr` flag.
  - Issues the read command to the I/O device.
  - Executes `wait_event_interruptible` to suspend the process until the `intr` flag becomes 1.

Interrupt mode

- When the scheduler decides to reexecute the process, the second part of foo_read() is executed and does the following:
  - Copies the character ready in the foo_dev->data variable into the user address space.
  - Terminates after releasing the foo_dev->sem semaphore.

Block Device Drivers

- Kernel support for block device handlers includes the following features:
  - A uniform interface through the VFS
  - Efficient read-ahead of disk data
  - Disk caching for the data

Block Device Drivers

- When a block device file is being opened, the kernel must determine whether the device file is already open.
- We use the term “block device driver”, we mean the kernel layer that handles I/O data transfers from/to a hardware device specified by both a major number and a minor number.

Block Device Drivers

- A real complication, however, is that block device files that have the same major and minor numbers but different pathnames are regarded by the VFS as different files, but they really refer to the same block device driver.
- To keep track, the kernel uses a hash table indexed by the major and minor numbers.
- If a block device driver associated with the given numbers is not found, the kernel inserts a new element into hash table.

`block_device` data structure

- Hash table includes 64 lists of descriptors. The fields are:

Type	Field	Description
Struct list_head	bd_hash	Pointers for the hash table list
atomic_t	bd_count	Usage counter for the block device descriptor
struct inode *	bd_inode	Pointer to the main inode object of the block device driver
dev_t	bd_dev	Major and minor numbers of the block device
int	bd_openers	Counter of how many times the block device driver has been opened
struct block_device_operations *	bd_op	Pointer to the block device driver operation table
struct semaphore	bd_sem	Semaphore protecting the block device driver
struct list_head	bd_inodes	List of inodes of opened block device files for this driver

Initializing a Block Device Driver

- Kernel customizes the methods of the file object when a block device file is opened. The default file operation methods for block device files are:

Method	Function for block device file
open	blkdev_open()
release	blkdev_close()
llseek	block_llseek()
read	generic_file_read()
write	generic_file_write()
mmap	generic_file_mmap
fsync	block_fsync()
ioctl	blkdev_ioctl()

Initializing a Block Device Driver

- The function checks whether the block device driver is already in use:
  - `bd_acquire(inode);`
  - `do_open(inode->i_bdev, filp);`
- The device driver initialization function must determine the size of the physical block device. The length, represented in 1024-byte units, is stored in the `blk_size` global array.

Initializing a Block Device Driver

- `The bd_acquire() function essentially executes the following operations:`
  - Checks whether block device file is already open.
  - Looks up block device driver in the hash table.
  - Stores the address of the block device driver descriptor in `inode->i_bdev`.
  - Adds inode to the list of inodes of descriptor.

Initializing a Block Device Driver

- Next, blkdev_open() invokes do_open(), which executes the following main steps:
  - Checks and initializes bd_op field of the block device driver descriptor corresponding to the major number of the block device file.
  - Invokes the `open` method of the block device driver descriptor
  - Increments the `bd_openers` counter of the block device driver drescriptor
  - Sets the `i_size` and `i_blkbits` fields of the block device inode object (`bd_inode`).

Sectors, Blocks, and Buffers

- Data transfer operation for a block device acts on a group of adjacent bytes called a ***sector***.
- In most disks, the size of a sector is 512 bytes.
- The kernel stores the sector size of each hardware block device in a table named `hardsect_size`.
- Block device drivers transfer a large number of adjacent bytes called a ***block*** in a single operation

Sectors, Blocks, and Buffers (contd...)

- In Linux, the block size must be a power of 2 and cannot be larger than a page frame.
- The same block device driver may operate with several block-sizes, since it has to handle a set of device files sharing the same major number. For instance, a block device driver could handle a hard disk with two partitions containing an Ext2 filesystem and a swap area.

Sectors, Blocks, and Buffers (contd...)

- Each block requires its own ***buffer***, which is a RAM memory area used by the kernel to store the block's content.
- When a driver reads/write a block from/to disk. It fills/updates the corresponding buffer with values.
- The size of a buffer always matches the size of the corresponding block.

Buffer Heads

- The ***buffer head*** is a descriptor of type `buffer_head` associated with each buffer.
- It contains information needed by the kernel to know how to handle the buffer.

The Fields of a Buffer Head

Type	Field	Description
unsigned long	b_blocknr	Logical block number
unsigned long	b_size	Block size
kdev_t	b_dev	Virtual device identifier
kdev_t	b_rdev	Real device identifier
unsigned long	b_rsector	Number of initial sector in real device
unsigned long	b_state	Buffer status flags
unsigned int	b_count	Block usage counter
char *	b_data	Pointer to buffer
unsigned long	b_flushtime	Flushing time for buffer
struct wait_queue *	b_wait	Buffer wait queue
struct buffer_head *	b_next	Next item in collision hash list
struct buffer_head **	b_pprev	Previous item in collision hash list
struct buffer_head *	b_this_page	Per-page buffer list
struct buffer_head *	b_next_free	Next item in list
struct buffer_head *	b_prev_free	Previous item in list
unsigned int	b_list	LRU list including the buffer
struct buffer_head *	b_reqnext	Request's buffer list
void (*)( )	b_end_io	I/O completion method
void (*)	b_dev_id	Specialized device driver data

Flags of b_state field (1 of 2)

- `BH_Uptodate` – Set if the buffer contains valid data.
- `BH_Dirty` – Set if the buffer is dirty.
- `BH_Lock` – Set if the buffer is locked.
- `BH_Req` – Set if the corresponding block is requested and has valid data.
- `BH_Mapped` – Set if the buffer is mapped to disk.

Flags of b_state field (2 of 2)

- `BH_New` – Set if the corresponding file block has never been accessed.
- `BH_Async` – Set if the buffer is being processed by end_buffer_io_async()
- `BH_Wait_IO` – Used to delay flushing the buffer to disk when reclaiming memory.
- `BH_launder` – Set when the buffer is being flushed to disk when reclaiming memory.
- `BH_JBD` – Set if the buffer is used by a journaling filesystem.

Buffer heads

- The `b_dev` field indentifies the virtual device containing the block stored in the buffer while the `b_rdev` field identifies the real space.
- This distinction has been introduced to model Redundant Array of Independent Disks (RAID) storage.
- The `b_blocknr` field represents the logical block number.

References:

- Understanding the Linux Kernel, Second Edition
by Daniel P.Bovet and Marco Cesati

Questions?

Thank You