

Operating Systems Design

10. Devices

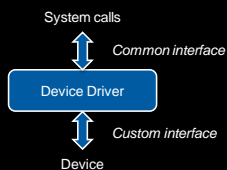
Paul Krzyzanowski
pzk@cs.rutgers.edu

Categories of devices

- **Block devices** (can hold a file system)
- **Network** (sockets)
- **Character devices** (everything else)
- Devices as files:
 - Character & block devices appear in the file system name space
 - Use open/close/read/write operations
 - Extra controls may be needed for device-specific functions (*ioctl*)

Device driver

Software in the kernel that interfaces with devices



Device System

Contains:

- Buffer cache & I/O scheduler
- Generic device driver code
- Drivers for specific devices (including bus drivers)

Device Drivers

- **Device Drivers**
 - Implement mechanism, *not* policy
 - **Mechanism**: ways to interact with the device
 - **Policy**: who can access and control the device
- Device drivers may be compiled into the kernel or loaded as modules

Kernel Modules

- Chunks of code that can be loaded & unloaded into the kernel on demand
- Dynamic loader
 - Links unresolved symbols to the symbol table of the running kernel
- Linux
 - `insmod` to add a module and `rmmod` commands to remove a module
 - `module_init`
 - Each module has a function that the kernel calls to initialize the module and register each facility that the module offers
 - `delete_module`: system call calls a `module_exit` function in the module
 - Reference counting
 - Kernel keeps a **use count** for each device in use
 - `get()`: increment – called from open when opening the device file
 - `put()`: decrement – called from close
 - You can remove only when the use count is 0

Device Driver Initialization

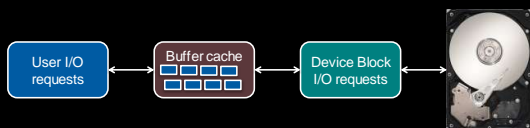
- All modules have to register themselves
 - *How else would the kernel know what they do?*
- Device drivers register themselves as devices
 - Character drivers
Initialize & register a `cdev` structure & implement `file_operations`
 - Block drivers
Initialize & register a `gendisk` structure & implement `block_device_operations`
 - Network drivers
Initialize & register a `net_device` structure & implement `net_device_ops`

Block Devices

- **Structured access** to the underlying hardware
- Something that can host a file system
- Supports only block-oriented I/O
- Convert the user abstraction of the disk being an array of bytes to the underlying structure
- Examples
 - USB memory keys, disks, CDs, DVDs

Buffer Cache

- Pool of kernel memory to hold frequently used blocks from block devices
- Minimizes the number of I/O requests that require device I/O
- Allows applications to read/write from/to the device as a stream of bytes or arbitrary-sized blocks



Blocking & Non-blocking I/O

- Buffer cache deals with the device level
- Options at the system call level
- **Blocking I/O:**
 - user process waits until I/O is complete
- **Non-blocking I/O:**
 - Schedule output but don't wait for it to complete
 - Poll if data is ready for input (e.g., `select` system call)

Asynchronous I/O

- Request returns immediately but the I/O is scheduled and the process will be signaled when it is ready
 - Differs from non-blocking because the I/O will be performed in its entirety ... just later
- If the system crashes or is shut off before modified blocks are written, that data is lost
- To minimize data loss
 - Force periodic flushes
 - On BSD: a user process, `update`, calls `sync` to flush data
 - On Linux: `kupdated`, a kernel update daemon does the work
 - Or force synchronous writes (but performance suffers!)

Buffered vs. Unbuffered I/O

Buffered I/O:

- Kernel copies the *write* data to a block of memory (buffer):
 - Allow the process to write bytes to the buffer and continue processing: buffer does not need to be written to the disk ... yet
- Read operation:
 - When the device is ready, the kernel places the data in the buffer
- Why is buffering important?
 - Deals with device burstiness (*leaky bucket*)
 - Allows user data to be modified without affecting the data that's read or written to the device
 - Caching (for block devices)
 - Alignment (for block devices)

File systems

- Determine how data is organized on a block device
- *NOT* a device driver
- Software driver
 - Maps low-level to high-level data structures
- *More on this later...*

Network Devices

- Packet, not stream, oriented device
- Not visible in the file system
- Accessible through the *socket* interface
- May be hardware or software devices
 - Software is agnostic
 - E.g., ethernet or loopback devices
- *More on this later...*

Character Devices

- **Unstructured** access to underlying hardware
- Different types (anything that's not a block or network device):
 - Real streams of characters: *Terminal multiplexor, serial port*
 - Frame buffer: *Has its own buffer management policies and custom interfaces*
 - Sound devices, I/O controllers, etc.
- Higher-level software provides line-oriented I/O
 - tty driver that interacts with the character driver
 - **Raw vs. cooked I/O**: *line buffering, eof, erase, kill character processing*
- Character access to block devices (disks, USB memory keys, ...)
 - Character interface is the unstructured (raw) interface
 - I/O does NOT go through buffer cache
 - Directly between the device and buffers in user's address space
 - I/O must be a multiple of the disk's block size

All objects get a common file interface

All devices support generic "file" operations:

```
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 *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readahead) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fil_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    ...
}
```

Device driver entry points

- Each device driver provides a fixed set of **entry points**
 - Define whether the device has a block or character interface
 - Block device interfaces appear in a **block device table**
 - Character device interfaces: **character device table**
- Identifying a device in the kernel
 - **Major number**
 - Identifies device: index into the device table (block or char)
 - **Minor number**
 - Interpreted within the device driver
 - Instance of a specific device
 - E.g., Major = SATA disk driver, Minor = specific disk
- Unique device ID = { type, major #, minor # }

How do you locate devices?

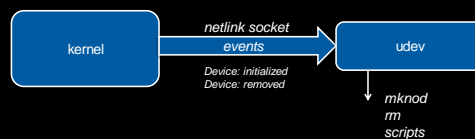
- Explicit namespace (MS-DOS approach)
 - C:, D:, LPT1:, COM1, etc.
- Big idea!
 - *Use the file system interface as an abstract interface for both file and device I/O*
 - Device: file with no contents but with metadata:
 - Device file, type of device, major & minor numbers
 - Devices are traditionally located in /dev
 - Created by the *mknod* system call (or *mknod* command)

Device names: Windows

- Windows NT architecture (XP, 2000, Vista, Win 7, ...)
 - When a device driver is loaded
 - It is registered by name with the Object Manager
 - Names have a hierarchical namespace maintained by Object Manager
 - \Device\Serial0
 - \Device\CDRom0
 - (Linux sort of did this with devfs and devtmpfs)
- Win32 API requires MS-DOS names
 - C:, D:, LPT1:, COM1:, etc.
 - These names are in the \??\ Directory in the Object Manager's namespace
 - Visible to Win32 programs
 - Symbolic links to the Windows NT device names

Linux: Creating devices in /dev

- Static devices (mknod)
- udev – kernel device manager
 - user-level process



Character device entry points

Character (and raw block) devices include these entry points:

- open:** open the device
- close:** close the device
- ioctl:** do an I/O control operation
- mmap:** map the device offset to a memory location
- read:** do an input operation
- reset:** reinitialize the device
- select:** poll the device for I/O readiness
- stop:** stop output on the device
- write:** do an output operation

Block device entry points

Block devices include these entry points:

- open:** prepare for I/O
 - Called for each open system call on a block device (e.g. on mount)
- strategy:** schedule I/O to read/write blocks
 - Called by the buffer cache. The kernel makes *bread()* and *bwrite()* requests to the buffer cache. If the block isn't there then it contacts the device.
- close:** called after the final client using the device terminates
- psize:** get partition size

Kernel execution contexts

- **Interrupt context**
 - Unable to block because there's no process to reschedule *nothing to put to sleep and nothing to wake up*
- **User context**
 - Invoked by a user thread in synchronous function
 - May block on a semaphore, I/O, or copying to user memory
 - E.g., *read* invoked by the *read* system call
 - (Linux) Driver can access global variable `context`
 - Pointer to `struct task_struct`: tells driver who invoked the call
- **Kernel context**
 - Scheduled by kernel scheduler (just like any process)
 - No relation to any user threads
 - May block on a semaphore, I/O, or copying to user memory

Interrupt Handler

- Device drivers register themselves with the interrupt handler
 - *Hooks* registered at initialization: call code when an event happens
- Operations of the interrupt handler
 - Save all registers
 - Update interrupt statistics: counts & timers
 - Call interrupt service routine in driver with the appropriate unit number (ID of device that generated the interrupt)
 - Restore registers
 - Return from interrupt
- The driver itself does not have to deal with saving/restoring registers

Handling interrupts quickly

- Processing results of an interrupt may take time
- We want interrupt handlers to finish quickly
 - Don't keep interrupts blocked

Delegation: top half → bottom half

- Split interrupt handling into two parts:
 - **Top half (interrupt handler)**
 - Part that's registered with `request_irq` and is called whenever an interrupt is detected.
 - Saves data in a `buffer/queue`, schedules bottom half, exits
 - **Bottom half (work queue – kernel thread)**
 - Scheduled by top half for later execution
 - Interrupts enabled
 - This is where there real work is done
 - Linux 2.6 provides tasklets & work queues for dispatching bottom halves
- Bottom halves are handled in a *kernel context*
 - Work queues are handled by kernel threads
 - One thread per processor (`events/0`, `events/1`)

I/O Queues

- When I/O request is received
 - Request is placed on a per-device queue for processing
- Device Status Table
 - List of devices and the current status of the device
 - Each device has an I/O queue attached to it

Device status table

I/O queue

I/O Queues

- Primary means of communication between top & bottom halves
- I/O queues are shared among asynchronous functions
 - Access to them must be synchronized (critical sections)

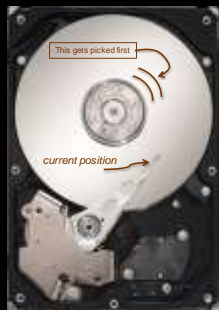
I/O Scheduling for Block Devices (disks)

Elevator Algorithms

- Elevator algorithm (**SCAN**)
 - Know: head position & direction
 - Schedule pending I/O in the sequence of the current direction
 - When the head reaches the end, switch the direction
- **LOOK**
 - When there are no more blocks to read/write in the current direction, switch direction
- **Circular SCAN (C-SCAN)**
 - Like SCAN, but: when you reach the end of the disk, seek to the beginning without servicing I/O
 - Provides more uniform wait time
- **C-LOOK**
 - Like C-SCAN but seek to the lowest track with scheduled I/O

Shortest Seek Time First (SSTF)

- Know: head position
- Schedule the next I/O that is closest to the current head position
- Analogous to shortest job first scheduling
- Distant cylinders may get starved (or experience long latency)



Scheduling I/O: Linux options

- **Completely Fair Queuing (CFQ)**
 - default scheduler
 - distribute I/O equally among all I/O requests
 - Synchronous requests
 - Go to per-process queues
 - Time slices allocated per queue
 - Asynchronous requests
 - Batched into queues by priority levels
- **Deadline**
 - Each request has a deadline
 - Service them using C-SCAN
 - If a deadline is threatened, skip to that request
 - Helps with real-time performance
 - Gives priority to real-time processes. Otherwise, it's fair

Scheduling I/O: Linux options

- **NOOP**
 - Simple FIFO queue - minimal CPU overhead
 - Assumes that the block device is intelligent
- **Anticipatory**
 - introduce a delay before dispatching I/O to try to aggregate and/or reorder requests to improve locality and reduce disk seek.
 - After issuing a request, wait (even if there's work to be done)
 - If a request for nearby blocks occurs, issue it.
 - If no request, then C-SCAN
 - Fair
 - No support for real time
 - May result in higher I/O latency
 - Works surprisingly well in benchmarks!!

Smarter Disks

- Disks are smarter than in the past
 - E.g.: WD Caviar Black drives: dual processors, 64 MB cache
- Logical Block Addressing (LBA)
 - Versus Cylinder, Head, Sector
- Automatic bad block mapping (can mess up algorithms!)
 - Leave spare sectors on a track for remapping
- Native Command Queuing (SATA & SCSI)
 - Allow drive to queue and re-prioritize disk requests
 - Queue up to 256 commands with SCSI
- Cached data
 - Volatile memory; improves read time
- Read-ahead caching for sequential I/O
- Hybrid Hard Drives (HDD)
 - Non-volatile RAM (NVRAM)

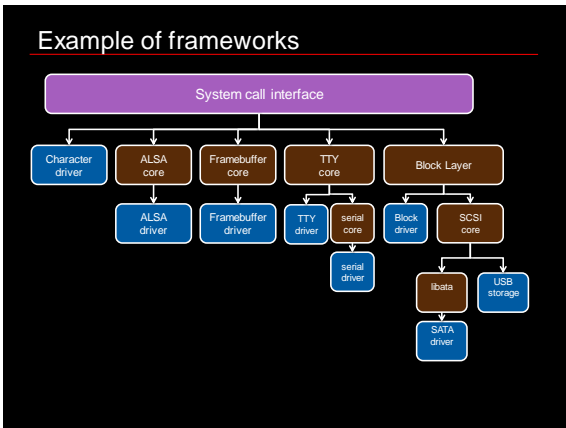
Solid State Disks

- NAND Flash
 - NOR Flash: random access bytes; suitable for execution; lower density
 - NAND Flash: block access
- No seek latency
- Asynchronous random I/O is efficient
 - Sequential I/O less so
- Writes are less efficient: erase-on-write needed
- Limited re-writes
 - Wear leveling becomes important (~ 100K-1M program/erase cycles)

Back to drivers

Frameworks

- Most drivers are not individual character or block drivers
 - Implemented under a framework for a device type
 - Goal: create a set of standard interfaces
 - e.g., ALSA core, TTY serial, SCSI core, framebuffer devices
- Define **common parts** for the same kinds of devices
 - Still seen as normal devices to users
 - Each framework defines a set of operations that the device must implement
 - e.g., framebuffer operations, ALSA audio operations
- Framework **provides a common interface**
 - ioctl numbering for custom functions, semantics, etc.

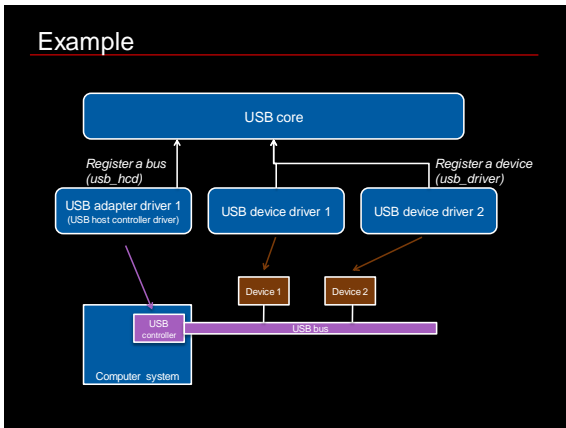


Example: Framebuffer

- Must implement functions defined in struct fb_ops
 - These are framebuffer-specific operations
 - xxx_open(), xxx_read(), xxx_write(), xxx_release(), xxx_checkvar(), xxx_setpar(), xxx_setcolreg(), xxx_blank(), xxx_pan_display(), xxx_fillrect(), xxx_copyarea(), xxx_imageblit(), xxx_cursor(), xxx_rotate(), xxx_sync(), xxx_get_caps(), etc.
- Also must:
 - allocate a fb_info structure with framebuffer_alloc()
 - set the ->fbops field to the operation structure
 - register the framebuffer device with register_framebuffer()


Linux 2.6 Unified device/driver model

- Goal: unify the relationship between: *devices, drivers, and buses*
- Bus driver
 - Interacts with each communication bus that supports devices (USB, PCI, SPI, MMC, I²C, etc.)
 - Responsible for:
 - Registering bus type
 - Registering adapter/interface drivers (USB controllers, SPI controllers, etc.): devices capable of detecting & providing access to devices connected to the bus
 - Allow registration of device drivers (USB, I²C, SPI devices)
 - Match device drivers against devices



Unified driver example

- USB driver is loaded & registered as a USB device driver
- At boot time
 - Driver registers itself to the USB bus infrastructure: *I'm a USB device driver*
- When the bus detects a device
 - Bus driver notifies the generic USB bus infrastructure
 - The bus infrastructure knows which driver is capable of handling the device
- Generic USB bus infrastructure calls *probe()* in that device driver, which:
 - Initializes device, maps memory, registers interrupt handlers
 - Registers the device to the proper kernel framework (e.g., network infrastructure)
- Model is recursive:
 - PCI controller detects a USB controller, which detects an I²C adapter, which detects an I²C thermometer



The End