Operating Systems

Devices

By Paul Krzyzanowski
last update: March 22, 2012

> *We live in a world of things, and our only connection with them is that we know how to manipulate them or to consume them.*
> *— Erich Fromm (1890-1980), The Sane Society, chapter 5*

Introduction

Ultimately, a computer is useful only if it can access devices and communicate with people and objects. Common devices include keyboards, displays, disks, mice, printers, scanners, and keyboards. They also include devices on the system board such as timers, and graphics chips, and audio chips. Each piece of hardware that is connected to a computer requires software to control it. This software is known as a **device driver**. The device driver allows us to have a consistent interface to the kernel and to hide device-specific details within the driver. For instance, the kernel outside of the device driver should not care about the details of writing a disk block and whether the the underlying device is a magnetic disk or flash memory. Likewise, it should not care whether your computer has a Microsoft mouse or a Logitech mouse. A device driver communicates with the **device controller** to control the device itself. A device controller is the hardware (and/or firmware) that controls the operation of the device. Its controls are typically mapped onto specific physical memory locations and it may also generate interrupts to alert the processor of events and use DMA (Direct Memory Access) to transfer blocks of data between the device and system memory.

Device drivers implement mechanism, not policy. A **mechanism** specifies the ways of interacting with the device. A **policy** enforces who can access the device and in what way they can control it. The policy will be handled at higher layers in the operating system and the device driver has only to implement the operations needed to interface with the device.

Very broadly, the device subsystem of an operating system contains three components:

1. Generic device driver code: these are abstract interfaces for devices and code for managing instances of and references to devices.
2. Drivers for specific devices: this is code written for interfacing with specific devices. This group also includes the code for **bus drivers**, such as discovering components on a USB bus and sending and receiving data on the bus.
3. The buffer cache: this manages the caching and I/O to block-oriented devices such as disks.

Categories of I/O

There are four broad categories of I/O:

- block devices
- character devices
- file systems
- network (sockets)

Of these, character and block devices will appear in the file system name space (we'll see how shortly).

Block devices

A **block device** provides *structured* access to the underlying hardware. Block devices are devices that can host a file system. They support addressable block-oriented I/O (e.g., *read block number*, *write block number*) and exhibit persistence of data. That is, you can read the same block number over and over again and get the same contents. Examples of block devices are: USB memory keys, disks, CDs, and DVDs.

Because blocks are addressable and persistent, block I/O lends itself to caching frequently used blocks in memory. The **buffer cache** is a pool of kernel memory that is allocated to hold frequently used blocks from block devices. By using the buffer cache, we can minimize the number of I/O requests that actually require a device I/O operation. the buffer cache also allows applications to read/write from/to the device as a stream of bytes or arbitrary-sized blocks. For example, suppose that a disk has a minimum block size of 512 bytes (as most disks do). If you want to modify 16 bytes on a disk, you need to read the entire 512-byte block (or two blocks if the 16 bytes straddle a block boundary), store the data somewhere (in the buffer cache), modify the 16 bytes, and write the modified block(s) out again.

File systems

The file system determines how data is organized on a block device in order to present higher-level software with a hierarchy of directories and files, with access permissions for both.

A file system is *not* a device driver. It is a software driver within the operating system that maps between low-level and high-level data structures.

Network devices

The network access device is a packet-oriented, rather than stream-oriented device. It is not visible in the file system but is accessible through the *socket* interface. We will examine this device when we look at networking. The underlying network device may be either hardware (e.g., an ethernet controller) or software (e.g., a loopback driver). Higher level networking software will be agnostic to this.

Character devices

Character devices provide **unstructured** access to the underlying hardware. In some ways, they are a catch-all for anything that is not a block or network device. Examples of character devices include:

- Traditional character-stream devices, such as a terminal multiplexer, modem, printer, scanner, or mouse.
- The frame buffer. This is considered a character device even though it has its own buffer management policies and custom interfaces
- Sound devices, I^2C bus controllers, and other bus interfaces.

In addition to supporting "character" devices, the character device drivers are also used with block devices to bypass the buffer cache and provide **raw** I/O operations directly to the user's program address space. For example, this feature is used for programs that dump and check file systems. When accessing block devices via this interface, the program has to abide by the restrictions of the hardware and ensure that the I/O is a multiple of the device's block size.

For terminal devices (virtual terminals as well as old-style serial-port terminals), it may make sense to process input from those devices in one of two modes:

line mode (cooked)

Input characters are echoed by the system as they are received (e.g. typed) and collected in a buffer until a return is received. Then the entire line is made available to the process that is waiting on a *read* operation. The user may correct typing errors in this mode and use special characters to start/stop output, suspend a process, or delete the current input line.

raw mode

Characters are made available to the process as soon as they are received. This is the only mode that makes sense for a non-terminal device such as a mouse. For a terminal, this mode is needed if a screen editor wishes to process every keystroke that it gets.

These modes were created in the era of dumb terminals to provide users with rudimentary line buffering capabilities without having a *read* system call return as soon as a single character was typed. Even shells tend to support in-place command editing and must revert to raw mode. The time that you mostly come in contact with a line driver is whenever you use a terminal window and run programs that request line-oriented input (e.g., via *fgets*).

Modularity

A goal in managing devices is that I/O operations should have a consistent interface for different devices. This is known as **device independence**. Device independence implies that higher levels of the operating system, libraries, and user applications need not, to the greatest extent possible, know that they are accessing one device over another. A user should not have to modify a program if the output is to go to a USB memory key rather than to a disk. This isn't always possible since devices may have very distinct capabilities but it is desirable to maximize the number of common interfaces.

Drivers are modular and may often be dynamically loaded into or removed from the kernel at any time. Under the Linux operating system, one can use the `insmod` command to add a kernel module and the `rmmod` command to remove it. A dynamic loader links unresolved symbols within the module to the symbol table of the running kernel.

When presented with a new module, a kernel on its own does not know what the code within that module is for. Each module has a function, called *module_init*, that the kernel calls when the module is first loaded to allow the module initialize itself and register each facility that the module supports, whether it is a driver or some other software abstraction, such as a file system. Under Linux, a device driver has allocate, initialize, and register a structure for the category of device that it controls:

Character drivers

Initialize and register a `cdev` structure and implement `file_operations`

Block drivers

Initialize and register a `gendisk` structure and implement `block_device_operations`

Network drivers

Initialize and register a `net_device` structure and implement `net_device_ops`

The *delete_module* system call calls a function named *module_exit* function in the module prior to removing the module from the kernel. The kernel also keeps a **use count** for each device in use. A module can be removed only if the use count is 0.

While we will focus on the kernel structure here, it is important to note that some drivers can be implemented at the user level. For example, the X windows server is a user-level process that offers graphic resources to X clients. It sits on top of a kernel-level device driver that gives it access to the frame buffer. Client applications connect to this user-level server to perform communications with the device.

Blocking, Non-Blocking, Asynchronous I/O, and Buffering

The system call interface allows us to either wait for data or not.

Blocking I/O means that the user process will *block* (wait) until the I/O is complete. If you're reading 1,500 bytes from a device, the *read* will block until the data is ready. The *select* system call is often used with non-blocking I/O to check whether data is ready for a device.

Non-blocking I/O means that an I/O system call would not put the process to sleep waiting on I/O. Instead, the process would get a status stating that the I/O is not ready. If

you're reading 1,500 bytes from a device and only 12 bytes are ready, your *read* request will return only the 12 bytes.

Asynchronous I/O is similar to non-blocking I/O in that the process would not be put to sleep. It is a two-stage operation: a request for a *read* or *write* operation is first made, and returns immediately. Later on, the process is notified that the operation is complete.

In the kernel, the buffer cache provides asynchronous I/O interface to block devices. A write to the buffer cache does not necessarily translate to an immediate write to the disk. This is good because it reduces the amount of I/O needed to the device in case that same block gets modified again in the future. There is a risk, however. If the system crashes or is shut off spontaneously before all modified blocks are written, that data is then lost.

To minimize data loss, the operating system will periodically force a flush of data that's cached in the buffer cache. On BSD systems, a user process, `update`, calls the *sync* system call every 30 seconds to flush data. On Linux, the operation is done by a process called `kupdated`, the kernel update daemon.

Buffered I/O is where the kernel makes a copy of the data on a *write* operation from the user process to a temporary buffer. Similarly, for a *read* operation, the kernel reads data into a buffer first and then copies the requested data to the user process. The operating system's buffer cache is an example of a use of input/output buffering.

Buffering is clearly an overhead since it requires copying blocks of data to an intermediate area. However, it has a number of benefits:

- It deals with device burstiness. Data may be read in chunks while you want to read it as a byte stream. A buffer provides a place to store the chunks while giving you the stream of bytes. This specific example is called a *leaky bucket*, since it provides a way to get a constant bandwidth stream with bursty input.
- Buffering allows user data to be modified without affecting the data that is actually being written to or read from the device. This is especially critical for asynchronous write operations. If you modify a buffer after you requested the *write* but before the data was copied to the device, the device gets the modified data. By buffering, you can provide a point-in-time snapshot of the data.
- Buffers serve as a cache for frequently used data. This is the main purpose of the buffer cache. It's often the case that the same blocks will be accessed from the disk over and over again. If they are cached then we don't have to perform the disk operation ... and we save a lot of time.
- Buffers help with alignment. As we mentioned in the above example, if you want to read 16 bytes but a disk forces you to read at least 512 bytes, you need to have a place to store the extra data. If you want to write 16 bytes, you'll need to read 512 bytes into a buffer, modify the 16 bytes, and then write the results back to the disk.

Devices as files

On many operating systems (POSIX-ones such as Linux, UNIX, BSD, OS X, as well as NT-derived Windows systems), devices are presented via the file I/O interface. Since we read and write to files, why not use the same mechanisms for devices? Certain operations may not make sense, such as a *seek* function on a character device, but all functions don't need to be fully supported to present the abstract interface.

On Linux, each character device driver initializes itself upon being loaded, registers a `cdev` structure that describes the device, and implements a set of functions in a `file_operations` structure:

```
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 (*readdir) (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 *, fl_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 *);
    ...
}
```

These functions are called by the kernel to transform generic input/output requests to device-specific operations. The "looks-like-a-file" interface abstraction applies only to character devices.

Each device driver implements a fixed set of **entry points**. It defines whether the device has a block or character interface (or multiple interfaces). Block device interfaces appear in a **block device table**. Character device interfaces appear in a **character device table**.

Block devices are used to implement file systems and are accessed via the operating system's buffer cache. A block device driver initializes and registers a `gendisk` (generic disk) structure and implements functions defined in `block_device_operations`.

A network device driver initializes and registers a `net_device` (network device) structure and implements functions defined in `net_device_ops`.

Locating devices

Within the kernel, a device is identified by two numbers:

- **major number**: Identifies the specific device driver and is an index into the device table (either the block or character table).
- **minor device number**: This number is interpreted within the device driver. It identifies which, of possibly several, specific devices the request is for. For example, a major number may identify a SATA disk driver. A minor number will identify a specific disk. on the SATA interface.

How do user programs locate devices? There are a couple of approaches:

1. **Use an explicit namespace for devices** that is unique from files. This is the approach taken by MS-DOS and supported by the Win32 API. Devices are given names such as `C:`, `D:`, `LPT1:`, `COM1:`, etc. The colon cannot be used as part of a file name and a device that hosts a file system will have the device name identified as part of the full pathname of the file (e.g., `D:\paul\proj1\main,cpp`).

2. **Incorporate device names into the file system namespace**. A device is a file with no contents (data) but whose metadata (descriptive information about a file) identifies it as a device, gives the device type (block or character), and provides the major and minor numbers. On POSIX systems, devices may be located anywhere in the file system but are traditionally placed under the `/dev` directory. These special files can be created with the *mknod* system call (or `mknod` command). When the file is opened, it is recognized as a device special file and high-level file operations are passed to the functions defined in the `file_operations` structure of the character device driver (located in the character device table and indexed by the major number).

   The Windows NT architecture (XP, 2000, Vista, Windows 7, etc.) uses a similar concept, although the device names do not live on the disk file system. When a device driver is loaded, it is registered by name with the Object Manager, which is responsible for presenting a name space to applications. Device names have a hierarchical namespace that is maintained by the Object Manager and integrated with the

namespace of files and other objects. For example, `\Device\Serial0`, `\Device\CDRom0`. One snag is that the Win32 API requires the old-style MS-DOS device names. These names reside in the `\??` directory in the Object Manager's namespace. The names are visible to Win32 programs and are symbolic links to the Windows NT device names

In summary, the Big Idea was to use the file system interface as an abstract interface for both file and device I/O.

Device driver structure

Character (and raw block) devices include these entry points and must implement the `file_operations` functions mentioned earlier:

- **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 devices must implement these functions:

- **open**: prepare for I/O - called for each *open* system call on a block device (e.g. when a file system is mounted)
- **strategy**: schedule I/O to read or write blocks. This function is called by the buffer cache. The kernel makes *bread* and *bwrite* requests to the buffer cache. If the block is not there then it requests it from the device. The strategy routine finds a list of I/O requests (`CURRENT`, defined to be `blk_devpMAJOR_NR].current_request`). It implements the code to move data from the device to the block (read) and from the block to the device (write).
- **close**: called after the final client using the device terminates.
- **psize**: get partition size — the capacity of the device.

Execution contexts and Interrupt handling

When kernel code, such a device driver (or anything else, for that matter), is executed, it is done so in one of three contexts:

Interrupt context

Interrupt context is created by the spontaneous change in the flow of execution when a hardware interrupt takes place. Any actions performed in this context cannot block because there is no process that requested the action that could be put to sleep and scheduled to wake up when an operation is complete.

User context

The kernel running in user context is invoked by a user thread when it makes a system call (or generates a trap). For example, when the user calls the *read* system call, the wrapper function for the system call executes a trap instruction that jumps to a well-known entry point in the kernel and switches the processor to run in kernel (supervisor) mode. However, the memory map is still unchanged and the kernel is aware of the user thread from which the request came (on Linux, the global variable `context` contains a pointer to `struct task_struct` that identifies the calling thread). Because of this, the thread (or

process) may block on a semaphore, I/O, or copy operation. It's the same user thread that just happens to be executing kernel code.

Kernel context

The kernel itself may schedule worker threads just like it schedules user processes. For all practical purposes, these no different than any threads that are associated with user processes; they just happen to run in kernel mode. Because they are scheduled entities, even though they have no relation to any user thread, they may block on I/O or any other operation that would cause a thread to block. When the condition that made the thread block does not exist anymore, the thread is put on the ready queue and rescheduled by the schedulers. Note: system call processing takes place in *kernel mode* in the *user context*, not in the kernel context.

Device drivers do not get invoked directly when a system interrupt occurs. Instead, they register themselves with the kernel's **interrupt handler** when they initialize themselves. This is known as a **hook**. Hooking is a general term for allowing software to intercept messages or events. It also allows multiple device drivers to share the same interrupt if necessary. The interrupt handler performs the following functions:

1. Save all registers
2. Update interrupt statistics
3. Call interrupt service routine in driver with the appropriate unit number (ID of device that generated the interrupt)
4. Restore registers
5. Return from interrupt

This process of registering to intercept an interrupt (or, in the general case, any message or event flow) is known as **hooking**. By separating interrupt handling from the driver, the driver does not have to deal with saving or restoring registers or dealing with any specifics of the interrupt handling mechanism. Moreover, interrupts may be shared and the operating system can keep global statistics on interrupts.

Because an interrupt is, by its very nature, an interruption of the current workflow on the system, we'd like interrupt service routines to finish quickly. In many systems, interrupts are disabled while an interrupt is being serviced, so that adds more urgency to finish quickly. Also, since the interrupt context cannot block, we have to make sure that the interrupt service routine does not perform any potentially blocking actions, such as waiting on I/O or a semaphore.

To ensure speedy interrupt servicing, interrupt handling is generally split into two parts:

- The **top half** (**interrupt service routine**) is the function that is registered with the kernel's interrupt handler (in Linux, via *request_irq*) and is called whenever an interrupt is detected. Its goal is to finish as quickly as possible. It saves any data in a buffer or queue (e.g., grabs an incoming packet from the ethernet card), schedules a *bottom half*, and exits.
- The **bottom half** of the interrupt handler (also known as the **work queue** or **kernel thread**) is the part that is scheduled by the top half for later execution. This is where the real work of the interrupt servicing is done. The bottom half is run in the *kernel context* and is scheduled by the kernel's process scheduler. Because of this, interrupts are enabled and the thread can perform blocking operations. On Linux 2.6, the kernel provides *tasklets* and *work queues* for dispatching bottom half threads. Linux creates one work queue thread per processor (with names such as `events/0` and `events/1`).

I/O Queues

I/O queues are the

primary means of communication between the top and bottom halves of an interrupt handler. A **device status table** keeps track of a list of devices and the current status of each device (e.g., idle or busy). Each device has an I/O queue attached to it. Whenever an I/O request is received, it is placed on a device's queue for processing.

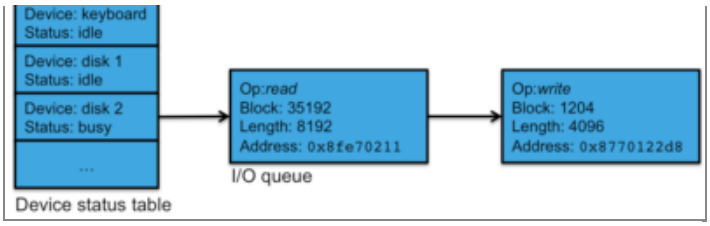

Figure 1. I/O queues

When an input or output operation completes, the device controller generates an interrupt, which causes the interrupt handler to call the top half of the device driver's interrupt service routine. The interrupt service routine removes the appropriate request from the device's queue, notifies the requestor that the request completed, and starts the next request from the queue.

Frameworks and a unified device/driver model

In earlier times, the block/character division of device drivers worked well. Drivers were primarily disk interfaces (block devices) or serial terminals (character devices). As more devices were added to machines, the character driver became the catch-all driver for any kind of device. Non-I/O related controls were implemented as custom commands that are called via the *ioctl* system call.

Frameworks were created to formalize a set of interfaces for drivers of specific types. For instance. The ALSA framework is a kernel-level API for audio. Instead of having each audio chipset create its own custom interfaces, they can all be written to conform to ALSA APIs. Frameworks define common parts and common interfaces for the same types of devices (e.g., ALSA core, TTY serial, SCSI core, framebuffer devices). All these devises are still seen as normal devices to users and treated as such by the kernel. A framework just defines a common interface and a set of operations that the device must implement (e.g., e.g., framebuffer operations, ALSA audio operations).

For example, a framebuffer driver has to implement functions defined in `struct fb_ops`, which is a set of framebuffer-specific functions and include functions such as:

```
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(),xx_get_caps(), etc.
```

It also has to allocate a `fb_info` structure with a call to *framebuffer_alloc()*, set the `->fbops` field to the operation structure, and register the framebuffer device with register_framebuffer().

The Linux 2.6 kernel also tried to unify the relationship between devices, drivers, and buses. with the unified device/driver model. A **bus driver** is a driver for interacting with each communication bus that supports multiple devices. Example bus drivers are USB, PCI, SPI, MMC, and I2C. It is responsible for:

- registering the bus type
- registering adapter/interface drivers (USB controllers, SPI controllers, …). These are devices that are capable of detecting and providing access to devices connected to the bus
- allowing the registration of device drivers (USB, I2C, SPI devices)
- matching device drivers against detected devices.

I/O Scheduling and Disks

From the 1960s through around 2010, the magnetic disk has been a crucial device in many computers. Much of the structure of the buffer cache and block drivers has been designed with the disk in mind. For high storage density, random access, and low cost, it's still hard to beat this device, although flash memory is making strong inroads.

A disk (for data storage), is a disk-shaped piece of metal, typically 2.5 inches on laptops and 3.5 inches on larger systems (in the distant past, 5-18 inches were common sizes). It is coated with magnetic material. Several disks are sometimes mounted on one spindle. An individual disk is known as a platter.

Data is recorded on the surface of a disk on a series of concentric **tracks**. Each track is divided into a set of fixed-length blocks called **sectors** (which are separated from each other by inter-record gaps). A sector typically stores from 512 bytes to several kilobytes of data. A head reads or writes data on a track as the disk spins (typically at 7200 rpm, but other common speeds are 4200, 5400, 10,000, and 15,000 rpm). There is one read/write head per recording surface. A cylinder is the set of tracks stacked vertically (each track is under its own head). Since there is only one set of heads per cylinder, the disk arm must move (or seek) to access cylinders. A disk is a random access device (meaning that you can request any sector to be read at any time), but the access time for the data depends on several factors:

1. How much does the disk arm have to move to get to the required cylinder (**seek time**)?
2. How much does the disk have to spin to get the needed sector positioned under the read/write head (**rotational latency**)?
3. How fast can data be transferred to the computer's memory (**transfer rate**)? For SATA disks, a bit over 100 MB/sec is a typical sustained transfer rate.

The time for a disk to rotate to the right position may seem insignificant since it's spinning at 7,200 rpm, but let's consider what this time means to the CPU. If a disk spins at 7,200 rpm, then it spins at 120 revolutions per second, taking 8.3 milliseconds to rotate once. The average latency is half of that, or 4.2 milliseconds. A 3.2 Ghz Intel Core i7 processor can perform approximately 76,383 million instructions per second. Therefore, the CPU can perform approximately 318 million instructions in the time that it takes the disk to spin so that the proper sector will be positioned under the disk read/write head. Seek times are even longer. As far as processors are concerned, disks are — and always have been — slow.

Seek times are even worse. Average seek time on a disk is typically twice as slow as rotational latency (it used to be much slower in the past when disks used stepper motors to drive the read/write head).

Disk Scheduling

As with other slow devices, requests for disk service are stored in a queue. By the time a disk has completed one data transfer operation there may be several more requests in the queue. The question that arises is that of which order should these requests be processed. Most often, the goal is to get all disk block read/write requests to get serviced as quickly as possible. Given that seeks are the slowest part of a disk, the challenge is to minimize overall head movement on a disk. There are several algorithms that can be used to sort disk requests.

First come, first served

The simplest thing to do is to service each disk request on the queue in the order that it arrived. The drawback is that this scheme does not try to optimize seek time. Consider having one request for data on cylinder 20, the next for data on cylinder 1,800, and the following for data on cylinder 21 and the one after that for data on cylinder 1,900. Surely it makes more sense to service cylinders 20 and 21 together and to service cylinders 1800 and 1900 together! The first-come-first served algorithm does not take this into consideration and will seek to cylinder 20, then to cylinder 1800, then to 21, and then back to 1900. We find

ourselves spending most of our time seeking back and forth rather than transferring data.

Elevator algorithm (SCAN)

In the **elevator algorithm** (also called **SCAN**), we keep the disk head moving to inner cylinders, processing queued requests in order of increasing cylinder numbers until there are no more requests in that direction. Then we start seeking to outer cylinders, processing any queued requests in order of decreasing cylinder numbers until the disk reaches the highest track number. After that we reverse again. This behavior is analogous to that of an elevator.

Figure 1. Elevator algorithm

With this algorithm, we need to know the current head position and direction of the disk head (figure 2). I/O is scheduled in the sequence of the current direction. For example, if we're at track 600 and are moving inward (toward higher track numbers), we will schedule I/O requests for higher track numbers from lowest to highest in sequence — but only for tracks 600 and higher. When the head reaches the end, we switch the direction and schedule any pending I/O sequenced from high track numbers down to 0.

LOOK

LOOK is a common-sense improvement over SCAN. Instead of seeking to the very end and the very beginning each time, let's just seek inward until there are no more requests in that direction. Then we start seeking to outer cylinders, processing any queued requests in order of decreasing cylinder numbers until there are no more requests in that direction. After that we reverse again.

Circular SCAN (C-SCAN)

1G With SCAN and LOOK, tracks in the middle of the disk get a statistical advantage in decreased average latency. In an attempt to provide more uniform wait times, the **Circular SCAN** algorithm only schedules requests when the head is moving in one direction. Once the end of the disk is reached, the head seeks to the beginning without servicing any I/O.

Circular-LOOK (C-LOOK)

This is like C-SCAN but instead of seeking to the start of the disk, we seek to the lowest track with scheduled I/O.

Shortest Seek Time First (SSTF)

To minimize average response time for processes, we were able to use a shortest job first scheduling algorithm. We can attempt a similar strategy with disk scheduling: the queue of current requests can be sorted by cylinder numbers closest to the current head position and requests made in that order. On average, the head is in the middle of the disk. The drawback with SSF scheduling is that requests for data on the ends of the disk may suffer from extreme postponement as "better" requests may keep coming in and be scheduled ahead of the less desirable requests. SSF also may create a lot of disk seeking activity as the head may seek back and forth with ever wider swings as it services the more distant requests.

Examples: Linux disk scheduling options

Linux 2.6 gives you a choice of using one of several disk scheduling algorithms. Let's take brief look at what they offer.

Completely Fair Queuing (CFQ)

This is the default scheduler on most distributions. Its goal is to distribute I/O equally

among all I/O requests. It supports two types of requests. Synchronous requests go to per-process queues with time slices allocated per queue. Asynchronous requests are batched into queues by priority levels.

Deadline

Each request has a deadline. Requests are serviced using a C-SCAN algorithm. However, if a deadline for any I/O operation is threatened, skip to that request immediately. This approach helps with with real-time performance and gives priority to real-time processes. Otherwise, it's just a C-SCAN algorithm.

NOOP

This is a super-simple scheduler. It's just a FIFO queue. This approach has minimal CPU overhead but assumes that the block device is intelligent and capable of rescheduling operations internally.

Anticipatory

Anticipatory scheduling introduces a delay before dispatching I/O to try to aggregate and/or reorder requests to improve locality and reduce disk seek. The hope is that, with the delay, there's a high likelihood of additional requests for nearby areas on the disk. After issuing a request, wait (even if there is outstanding work to be done). If a request for nearby blocks occurs, then issue it. If no request occurs, then just apply a C-SCAN algorithm. The approach is fair. While it has no support for real time and may result in higher I/O latency, it works surprisingly well in benchmarks!

Smarter disks

Most disk scheduling algorithms were designed for disks where the operating system would address data blocks by cylinder, head, sector and also had a good idea of where the disk head was at any time. That is no longer the case. Disks are a lot smarter and have a cache memory that is used to hold frequently used blocks, to prefetch blocks, and to resequence write requests. For example, consumer-grade Western Digital Caviar Black drives have dual processors and a 64 MB cache.

Disks support **logical block addressing** (LBA), presenting their storage as a contiguous set of block numbers so the actual location of a block is just an educated guess (we guess that contiguous blocks are quicker to access than distant blocks). In addition, disks have:

- Automatic bad block mapping, which can mess up algorithms that assume blocks are laid out sequentially. When a disk is formatted, the formatting process leaves spare sectors on a track for remapping bad blocks.
- Native Command Queuing. Both SATA and SCSI protocols allow the drive to queue and re-prioritize disk requests. Up to 256 commands can be queued up.
- The presence of a sizable cache on the disk allows frequently accessed blocks to be stored in the disk's RAM. **Read-ahead** can be enabled to optimize for sequential I/O. This causes the disk, if it has nothing else to do, to read successive blocks following an initial request into its cache.
- Some disks are a hybrid design, with a mix of non-volatile flash memory and disk storage. This enables bigger caches as well as persistent caches; the disk can acknowledge a write without actually writing it to magnetic storage.

Moreover, solid state disks are increasingly popular alternatives to magnetic storage. NAND flash memory still provides us with a block-based I/O interface but there is no seek latency whatsoever, so there is no advantage is sorting I/O requests. A fundamental problem with flash storage is that NAND flash supports a limited amount of write operations (typically under 100K through a million, depending on the vendor). Algorithms will need to be cognizant of the amount of writes they do to any given data block and need to consider

wear-leveling, where block writes can be distributed among all the blocks of storage.

References

- The Linux Kernel Module Programming Guide (http://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf), Peter Jay Salzman, Michael Burian, and Ori Pomerantz. Linux Documentation Project, May 18, 2007.
- Anatomy & Physiology of an Android (http://sites.google.com/site/io/anatomy--physiology-of-an-android), Patrick Brady, 2008 Google I/O Session Videos and Slides
- Google I/O sessions: May 19 - 20, 2010 (http://code.google.com/events/io/2010/)
- Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel (http://www.redhat.com/magazine/008jun05/features/schedulers/), D. John Shakshober, Red Hat Magazine
- Linux 2.6 (http://aplawrence.com/Linux/linux26_features.html), Dominique Heger & Philip Carinhas, Fortuitous Technologies
- INFO: Understanding Device Names and Symbolic Links (http://support.microsoft.com/kb/235128), Article ID: 235128, Microsoft Support
- An introduction to block device drivers (http://www.linuxjournal.com/article/2890), Michael K. Johnson, Linux Journal, Jan 1, 1995.
- Kernel Korner - The New Work Queue Interface in the 2.6 Kernel (http://www.linuxjournal.com/article/6916), By Robert Love, Linux Journal, November 2003
- Writing Device Drivers (http://dlc.sun.com/pdf/816-4854/816-4854.pdf), Copyright 2004, 2010, Oracle and/or its affiliates (formerly Sun).
- Introduction to Linux Device Driv (http://www.mulix.org/lectures/intro_to_linux_device_drivers/intro_linux_device_drivers.pdf), Muli Ben-Yehuda, IBM Haifa Research Labs and Haifux - Haifa Linux Club
- Linux Kernel architecture for device drivers (http://2010.rmll.info/IMG/pdf/kernel-device-drivers-rmll2010.pdf), Thomas Petazzoni, Free Electrons
- udev – A Userspace Implementation of devfs (http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-OLS2003.pdf), Greg Kroah-Hartman∗ IBM Corp. Linux Technology Center. Proceedings of the Linux Symposium. July 23-26, 2003.
