Device Drivers

Device drivers are a collection of functions grouped into a module within the kernel. They can be statically linked into the kernel and available at boot time, or dynamically loaded as needed.

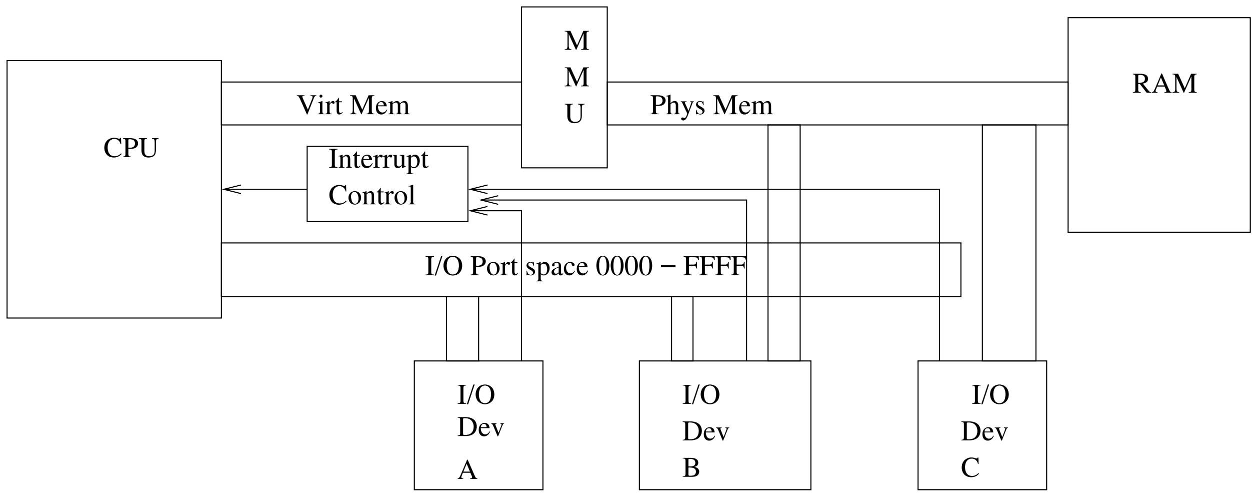
The purpose of device drivers is to provide a universal interface to hardware devices, isolating the details of manipulating the hardware within the device driver code. Some device drivers correspond to actual hardware, others are "pseudo-devices", such as the pseudo-tty driver which provides remote command-line access through programs such as telnet or ssh. Other notable pseudo-devices include `/dev/zero`, which is an infinite source of 0 bytes and a sink for data, and `/dev/null`, which is used as a sink for data or a source of immediate EOF.

Within the Linux kernel, a device driver can be thought of as having four facets:
• Routines called when the driver is loaded, which register the driver with the kernel and probe for hardware (if applicable). Similar routines exist for unloading a driver (if it is not in use).
• Routines which are called directly, or indirectly, from a synchronous context, i.e. a system call or fault handler. E.g. during a read system call, control eventually reaches the device driver's read routine, which initiates the I/O operation, and then puts the task to sleep pending completion.
• Routines which are called in an interrupt context. This is often called the "bottom" of the driver. Interrupt handler routines must be lightweight. They must not consume excessive CPU time, or perform any operation which might block indefinitely.
• Routines which are called in a deferred, or "soft" interrupt context. When a device driver must perform a complicated or potentially blocking operation in response to an interrupt, it uses this callback mechanism to avoid encumbering the interrupt response path.

I/O Addressing and Device Registers

Access to I/O devices is an extremely architecture-dependent issue. On the X86 architecture, there is a separate 16-bit address space called **I/O Ports** which is used for I/O device control and status registers. This address space is not the same as memory, and must be accessed using special IN and OUT instructions in X86 assembly. In addition, many I/O devices on the X86 PC architecture have mappings to physical addresses. The kernel uses an area of its virtual address space to create mappings to these special I/O memory areas. The physical addresses containing I/O devices are "holes" in the pool of available page frame addresses.

Because I/O ports are not memory addresses, they are not subject to the MMU. I/O ports generally can only be accessed from kernel mode. It is possible to associate an I/O port permissions mask with a process which allows that process to access specific I/O ports. This is sometimes done to accelerate graphics performance for user-mode programs. Setting I/O permissions requires root (superuser) access. Most other architectures do not have the equivalent of I/O ports and all hardware must be accessed via memory.



While space here does not permit a full discourse on I/O device architecture, in general we can say that any given I/O device has one or more `registers` which are accessed either via I/O port number addresses and/or physical memory addresses. The selection of which port numbers or physical addresses are used is part of the auto-configuration process that is built in to I/O protocols such as PCI. I/O device registers have nothing to do with CPU registers. In general, an I/O device has the following types of registers:
• Configuration: Writing to a configuration register changes the basic operation of the device, selects modes of operation, determines what other port or physical addresses the device might use, determines what interrupt vectors the device might use, etc.
• Control: Writing to a control register starts or stops a device operation. E.g. on the obsolete floppy disk I/O controller, the control register can be used to start and stop the spinning of the diskette media.
• Status: Reading from a status register gives information such as whether data are ready to be transferred, if the device is functioning properly, if media are inserted, etc.
• Data transfer: On "polled" I/O devices (see discussion below), data transfer can be accomplished by reading/writing the data transfer register(s)

Often, a physical register at a given I/O port address contains a combination of these functions. E.g. a port number might be a Configuration & Control register during an OUT operation, and a Status register during IN.

Buses and device driver layering

A long time ago, most I/O devices were "dumb" and relied on the CPU and the device driver to operate them. A disk device interface might literally have bits for turning the spindle motor on/off, stepping the head to find the track, and moving the data bit-by-bit from the head. In modern computers, the devices are much smarter. Very often, there are additional layers which are known as "buses" to isolate the CPU and operating system from the low-level details. On these buses, specific "protocols" are followed, similar to network protocols, which define the state diagram of the bus, the types of transactions and messages which can take place, error handling, etc.

Historically, buses can be categorized as serial or parallel. In the latter case, the I/O bus resembles a memory bus, with control, address, clock and data lines. At one time, parallel I/O buses predominated, with the most prolific being the Small Computer Systems Interface (SCSI) which used buses of 50, 68 or 80 wires.

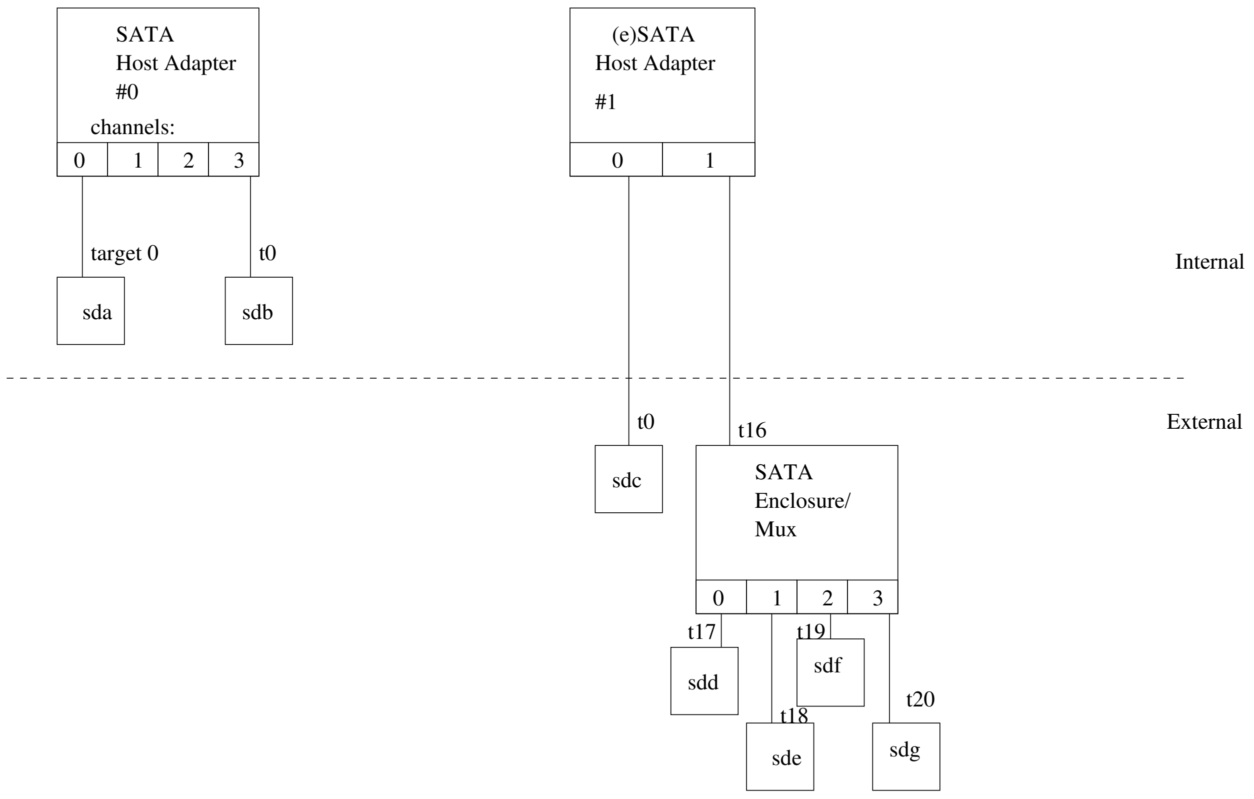
In modern computer architecture, buses are physically implemented with point-to-point serial interfaces (much like modern Ethernet ports). With rising data transfer rates, it becomes impossible to manage the "slew" between the data lines of a parallel bus, and it becomes impractical to have multiple devices tapped into the same bus wire (other than power) because of capacitive loading and transmission line type effects. Thus we see mostly SATA/SAS (essentially a serialized version of SCSI) for disk drives and Universal Serial Bus (USB) for just about everything else. Although these are "serial" bus ports, to improve performance, the port may have multiple "lanes", each with its own clock and data wires. This does not make it parallel because electrically, each lane is independent.

A given instance of a bus can logically have multiple target devices attached to it, even if physically there are some electronics in between to resolve issues at the physical level. E.g. a USB "Hub" extends a USB bus and allows additional devices to be logically part of that bus. A multiplexer (or "extender") can allow multiple SATA or SAS hard drives to be accessed via a single SATA/SAS physical port.

From the standpoint of the kernel, there is a hierarchy of layers that must be navigated to communicate with a target I/O device:
• The Host Controller is the physical chipset which is directly accessed by the kernel using IN/OUT instructions and/or memory-mapped I/O. It provides one or more physical bus ports which allow for bus-level communication with other devices.
• The Bus layer is a collection of kernel modules that provide the protocol support for a given bus. E.g. we want to transfer data to a given device at a given bus address, one of these modules prepares the necessary USB or SAS/SATA message blocks and submits it to the correct host controller module. In theory, the bus layer modules are independent of the host controller drivers. It shouldn't matter which specific chipset we have on our computer, because the protocol messages are all the same
• Bus addressing layer: A given controller may have multiple physical ports, each of which forms an independent bus. The Linux kernel calls these "channels". On each channel, physical devices connect to the bus with a specific bus address known as the target address (which must be unique to that given bus). Finally, each target may have sub-addressing.

To further complicate matters, many mass storage devices are now connected via USB. Although from a bus perspective these target devices are USB devices, they speak the same protocol known historically as SCSI (which is also spoken by SATA and SAS hard drives). The Linux kernel will then instantiate a virtual SCSI "bus" to represent these USB devices as if they were connected via SCSI, SAS or SATA, so that it can use the existing SCSI disk layer to access them.

The `/sys/bus` directory within the `/sys` pseudo-filesystem allows one to visualize the physical devices that the kernel "sees" on each bus. Other utilities, such as `lsusb`, present this in a more user-friendly output format. Reading `/proc/scsi/scsi` gives a list of the SCSI (including SATA, SAS and USB-connected) storage devices that the kernel sees. One can also write to this file (as superuser) to effect certain control operations, such as forcing the kernel to re-scan for a new disk.



Data Transfer Modes, Direct Memory Access

Transferring data to and from I/O devices can be done in what is called **polled mode**, in which the CPU transfers one small chunk of data directly using the device control registers, then waits for an interrupt to indicate that the device is ready for the next chunk. Such an approach is not efficient for devices which need to transfer bulk data with any appreciable speed.

Direct Memory Access means that I/O devices can read and write physical memory without the intervention of the CPU. A DMA transfer is initiated by the kernel. First one or more page frames are allocated for the transfer (or they may already be allocated...e.g. a mapped file). The kernel pokes the hardware device control registers to inform it of the buffer address, then initiates the operation. Data transfer happens and the CPU is not involved in this. The CPU receives an interrupt when the transfer has been completed. Some I/O devices work one request at a time, and must be given contiguous page frames. Others can establish a complex work queue of different transfers to different pages.

DMA can be said to be synchronous or asynchronous. The above describes synchronous DMA. In asynchronous mode, the device decides when to initiate a transfer. The kernel pre-allocates a pool of page frames for the device's use, and receives interrupts whenever a transfer has happened. This asynchronous mode is the way network interface cards have to work, since the kernel can not predict in advance when a network packet will arrive.

Page frames which are being used for DMA transfer are locked by the kernel by setting flags in the page descriptor data structure. Another term for this is **pinning** the page to the transfer. Pinning the page prevents it from being reclaimed or another I/O transfer being initiated to or from the same page, until the first transfer is done.

On the X86 architecture, the DMA controller speaks directly to the memory arbiter/controller with physical addresses. In some other computer architectures, I/O devices are given their own virtual address space with an IO MMU.

Major and Minor numbers

Device drivers are identified within the kernel by major and minor numbers. By default, they are each 8 bits long and together form a 16-bit **device number**, which is of type `dev_t`. (On some modern Linux kernels, the device number address space is larger) A unique major number is assigned to each device driver in the system, while the minor numbers are assigned by the device driver to refer to the devices it manages. For example, all of the `/dev/hda*` devices, which refer to (legacy) ATA hard drives, are managed by one device driver with major number 3. Unique minor numbers refer to each partition of each disk in the system.

There are two distinct namespaces for major device numbers, `character` and `block`. Block devices are those in which filesystem volumes usually live. They are generally addressed in terms of large chunks of contiguous data ("blocks") and are random-access. Character devices tend to work one character at a time, and are not random-access. Examples of character devices include "COM", mouse and keyboard ports.

On some UNIX operating systems other than Linux, e.g. Solaris, block devices also have a character device interface, which is the "raw" (unbuffered) mode. In Linux, this is not the case -- unbuffered access is done by using the `O_DIRECT` flag when opening the block special file.

Allocation of major device numbers is determined by the maintainers of kernel code. Most important devices have pre-allocated numbers. Device driver developers can use some of the unassigned numbers.

The actual major and minor device numbers are of little concern to the end-user or programmer. Devices are accessed through the regular filesystem namespace, by means of character special and block special inodes. These, which were mentioned in Unit 2, have inode type fields of `S_IFCHR` or `S_IFBLK`, respectively.

Block Devices and the Buffer Cache

There are two ways of looking at block devices. On the one hand, they are an array of sectors. At the same time, a given sector may contain a portion of a file, or may contain a filesystem data structure such as an inode, but not both. Further complicating this, block device files can be accessed via mmap in addition to read/write system calls.

Within the Linux kernel, another interface exists known as the **Buffer Cache**. It is synchronized with the page cache, but is addressed in terms of device numbers and block (sector) numbers. Filesystem modules tend to use the buffer cache interface to read inodes, superblocks, freelist bitmaps, indirect blocks and other metadata and control structures.

Buffers are controlled by `struct buffer_head` structures. When a page frame holds data which are also accessible through the buffer cache interface, the `private` field of the page descriptor is used as a pointer to the buffer head representing the first block within the page. Other blocks in the same page are chained through their buffer head structures through the `b_this_page` field of the buffer head. these areas.

During normal file read/write activity, whole pages at a time are read/written. The filesystem layer translates this 4096-byte chunk which is contiguous with respect to the file into a series of buffer cache requests representing the disk blocks which comprise that chunk. It could be (and indeed it often is the case with EXT2/3/4 filesystems using a 4K allocation unit) that these are also contiguous disk blocks, but that is not always the case. When the I/O operation completes, the list of sectors comprising that page is recorded, as described in the paragraph above. At the same time, that page has to be correlated to the address space of the block device special file:

The inode associated with a particular block device has an `address_space` with a radix-64 tree. Given a device number and offset, we can find if there is a page frame which contains the image of that part of the raw disk. However, some complexity is introduced when the same piece of data is accessed both through a regular file and through direct use of the block device file. The filesystem does not necessarily use a block size (minimum allocation size) which is the same as the page size. Since files are not necessarily allocated to contiguous disk blocks, a page of memory representing the contents from the file's perspective may contain the images of multiple, non-contiguous blocks from the block device's standpoint.

So, given a device number and block number, the kernel can translate that block number into the nearest page-aligned offset within the block device, and then consult the radix-64 tree associated with the block device's inode. If there is a page descriptor attached to the corresponding node of that tree, then that page *might* contain the cached copy of the block in question. The kernel needs to walk through the buffer head list associated with that page and see if there are any matches.

A block device file being like any other file, it can be mmap'd. There are some very subtle consistency problems which can arise when a block device file is mmap'd and so is a file which lives on that block device! Since blocks are smaller than pages and not allocated contiguously, a page frame can either hold the valid image of a page-aligned contiguous portion of the file, or the valid image of a page-aligned, contiguous portion of the block device, but not both at the same time. Fortunately this is not a common situation. It is generally considered bad form to be working on the block device directly at the same time it is mounted.

Direct and Raw I/O

A major advantage of memory-mapped files is the saving of an additional memory copy operation. When using e.g. the write system call, the data must be copied once from the user address space to a buffer page in kernel space, then copied again during DMA. Linux also supports **direct I/O** in which the buffer address supplied to read or write system calls is used directly by the kernel as the DMA buffer. Finally there is a "raw" character special device which can be bound to a block device and used to issue I/O requests directly to the disk. These latter two modes bypass the page cache and are useful to database applications which effectively implement their own proprietary filesystems to store the database tables, and therefore are not concerned with caching or the UNIX filesystem model.

Block I/O & Disk Strategy

All I/O operations with block devices are handled by the device driver using `buffer_head` structures, because the block device works at the granularity of a block, not a page. To be more correct, device I/O happens in terms of 512 byte **sectors**, and blocks always contain contiguous sectors. An I/O request is encapsulated in another data structure called `struct bio`. Each `bio` refers to one or more `buffer_head` structures and contains a direction flag (either READ or WRITE).

Linux assumes that disks are addressed in terms of sequential sector numbers. Of course, actual physical disks are addressed in terms of three variables: cylinder, head and sector. Older disk technology, such as floppy disks, required the disk I/O commands to be in these CHS terms, and if that is the case, the device driver will translate. All modern hard disks use Logical Block Addressing and perform the translation to the internal geometry within the hard disk's on-board controller. This translation is done such that "S" is least significant and "C" is most signficant. Therefore, the effect of presenting a series of I/O requests to the disk with the logical block number increasing by one each time is to step the head assembly from the innermost to the outermost track. Note that since recording density in terms of Mbits/square inch tends to be a limiting factor, there will be more sectors per track on the outer parts of the platter than the inner.

Moving the head assembly is the slowest operation for a hard drive. For example, a specification for a particular mid-range hard drive, 250GB capacity, 7200 RPM, gives a data transfer rate of 70MB/sec. Average random-access seek time is 9msec, while average single-track seek time is 1msec. The average rotational latency (because of the need to wait for the disk head to line up with the sector in question) is approximately one half of the rotational period, which for the drive in question yields 4.16 msec.

Therefore it will take approximately 7 microseconds to read the data from one 512-byte sector, but up to 600 times longer to read two non-adjacent sectors on the same track, and over a thousand times longer to read two sectors which are on different tracks. It is apparently advantageous to transfer contiguous sectors and to minimize head seeking. The classic operating systems approach to this problem is known as the **"Elevator Algorithm"** and resembles the manner in which a traditional (i.e. non-"smart") elevator services requests. All disk I/O requests for a given device are placed into a sorted list. The device driver has a notion of the current request and the direction, either towards higher logical sector numbers or towards lower. The driver services all of the requests in one direction, then works in the other direction until all requests are done, then changes direction again, etc.

Unfortunately, this algorithm does not exhibit any queing "fairness", in that much older requests may get passed up in favor of much newer requests. Linux allows the elevator algorithm to be tuned by the administrator to balance fairness against performance, and calls this "Completely Fair Queueing".

To some extent, the on-board disk controller itself performs the same function. With modern hard drive interfaces such as SATA-II and SCSI, a mode of operation called **Tagged Queuing** is used. Requests are given to the hard disk along with a small integer tag to identify them. The hard disk's on-board controller buffers these requests and satisfies them in the order which it thinks is optimal. As each request completes, the disk sends a message back to the controller board in the computer. This message contains the tag number of the completed request.

The advent of Solid State Disks has complicated matters. There is no "seek time" because there is nothing mechanically moving within. Read performance is basically independent of block address ordering, but because writes are performed in larger chunks, and because the SSD internal controller tends to translate the block numbers presented by the host into internal block numbers to accomplish "wear levelling", it can be difficult for the kernel to make good scheduling decisions that optimize performance. This continues to be an active area of OS research.

Partitions, Concatentations and Logical Devices

More often than not, the raw disk drive is larger than the largest volume that we'd like to store on it, or, for security, performance or administration reasons, we'd like to have our overall filesystem consist of several independent volumes, but we have fewer physical disks than the number of volumes we'd like to create. This is where partitioning comes into play.

When a hard disk is used with partitions, the first sector contains a `partition table` which contains a list of partitions. Each partition is described by its starting sector offset (not including the partition table), its length, an identifier of the type of filesystem which is intended to be created on it, and flags such as whether the partition should be writable or mountable.

Historically, each operating system had its own partition table format (as well as its own volume format) and that made reading disks from a "foreign" system challenging. Linux supports several partition table formats. The most common are MBR/FDISK (MBR==Master Boot Record and FDISK was the name of the DOS command to partition the disk), which is the same format used by MSDOS and Windows, and EFI/GPT (Extensible Firmware Interface / Globally Unique Identifier Partition Table) which is supported by many operating systems and hardware platforms. Because the older MBR/FDISK format is limited to disks of 2TB and under, it is generally necessary to use GPT for larger disks.

When a physical disk drive is "discovered" by a Linux kernel, either during the initial scan of the buses during boot, or when a new device appears on a bus (e.g. a USB storage device is plugged in), the kernel first fetches sector 0 of the disk. By looking at the first few bytes, it can take an educated guess if the partition table is FDISK, GPT or some other supported type. It then loads the partition table (which may span several sectors) and in addition to creating a device inode such as `/dev/sdh` to represent the entire physical disk, also creates nodes `/dev/sdh1` for the first partition, `/dev/sdh2` for the second, etc. Any accesses to the partitions are adjusted by adding the appropriate sector offset to arrive at the raw disk block number.

It can also happen that we need to make a single filesystem (volume) which is larger than any single disk that we have. Linux has a tool known as LVM (Logical Volume Manager) to create virtual disk drives which are formed by first concatenating multiple physical disks (or partitions). That concatentation can then be re-partitioned as needed to create logical volumes. In some cases, a logical volume might also have a partition table placed on it.

Mirroring

Because hard disks are prone to data loss or complete failure (this includes Solid State Disks too), many system administrators configure mirrored drives. There are several ways to approach this. Systems which are intended for server use often have a hardware card which performs the mirroring, or it can be done by the kernel using the metadisk (`md`) driver and related tools.

We haven't the time here to explore all of the mirroring configurations. The reader is invited to explore "RAID" (Redundant Array of Inexpensive Disks) "levels" such as RAID-1 and RAID-5.

When a physical disk is mirrored, we use the virtual disk interface, e.g. `/dev/md1`. A write to that virtual disk results in write requests being issued to two (or more) underlying physical disks (or partitions) so there are always two very recent copies of the data. A read can be satisfied from either disk and thus read performance is approximately doubled.

In the event that one disk fails, the other disk continues to handle all read and write requests, so there is no interruption to service. An alert is raised and the system administrator replaces the defective disk. The kernel is informed of this replacement and begins a background kernel-mode task to "re-sync" the mirror. Each block of the good disk is read, in sequence, and written to the replacement disk. This could take hours or even days depending on the size of the disk and the amount of ordinary read/write traffic load.

A modern trend has been to move away from mirroring and concatenation at the disk level and instead have filesystems which natively support being composed of multiple disks. One of the most popular is ZFS (Zettabyte Filesystem) which originated with the Solaris kernel but is now available for other UNIX variants including Linux. One advantage of having the filesystem manage the disks is that it knows what is on them. If a disk needs to be replaced on a large volume which is not that full, ZFS is more efficient because it only has to re-sync the data that are actually there, vs. the entire disk.

Creation of the /dev inodes

Traditionally, the inodes under /dev were created statically, or manually by the system administrator. On Linux systems, the device drivers which are part of the kernel at boot time each execute a self-probe to see if any devices are present which they handle.

System utilities which vary from implementation to implementation determine the results of these probes and create the appropriate /dev entries without user intervention. A similar process is performed when device drivers are dynamically loaded into the kernel after boot. These `/dev` names do not necessarily correspond with physical/bus addressing, but follow some naming convention which is specific to each driver. E.g. the `sd` driver simply names the hard drives a,b,c etc. as it discovers them.

One can use the mknod command or system call to create an inode which is a block device or character device, specifying the major and minor device number in doing so. Let us say we execute the command `mknod /tmp/disk b 8 1`. Since major device 8 is the `sd` disk driver, this is equivalent to the system-created `/dev/sda1` node. While it may be useful to create device nodes by hand, it can also be dangerous. E.g. let us say we have a setuid-root utility which allows non-root users to mount USB memory sticks at a certain place in the filesystem such as `/mnt/sticks`. A malicious user could create the above block device inode on this USB stick and give 666 permissions to it. Then, when the removable volume is mounted, this would allow the non-root user to open the raw disk partition for reading and writing! To guard against this, the `nodev` mount option is used which tells the kernel to ignore all block and character device inodes on that mounted filesystem. (A similar `nosuid` option ignores the setuid and setgid bits on executables for security reasons).

E.g. Opening and Reading a block device special file

In traditional UNIX systems, these device special inodes are created statically in a directory called `/dev/`. The `mknod` command, which requires root access, creates device nodes. Let us say we execute the command:

```
        more /dev/sda1
```

This will page through the raw contents of the first partition of the first hard drive (not a particularly useful operation though.) Within the kernel, calling `open` with the pathname `/dev/sda1` performs the following steps:

• Allocate a free file descriptor number from `current->files->fdtable[]`
• Allocate a `struct file` and point fdtable[fd] to it.
• Lookup the supplied path name, resulting in a `dentry`. Point `file->f_dentry` to it.
• Get the `inode` corresponding to `dentry->inode`. (If it is not already in the inode cache, read it in from disk). Check the inode permissions to see if the open is allowed (generally only root is allowed to open raw disk devices). At this point, the inode methods table is still the one associated with the parent filesystem (e.g. ext2fs).
• Seeing that the inode type is a device special file, the inode is modified so that it appears to be in one of two dummy filesystems, `bdev` or `cdev`, for block or character special nodes respectively. The generic open routine for either of these dummy filesystem types looks at the major device number to find the device driver descriptor. The device driver's

open method is called. In our case, this is a block device. Generally, block devices are already "open" because they were probed during system startup, so there is nothing much to do here, other than to note within the private section of the inode that our target is SCSI disk "a", partition #1.
• The system call returns with the file descriptor. When we later perform a read system call on this file descriptor:
• the `read` method of the inode is invoked. This points to the generic read method for block devices.
• The "buffer cache" is consulted to see if we already have an image of this particular range of disk sectors in memory.
• Assuming that we do not, an I/O request is formed and submitted to the block I/O subsystem, which dispatches the request to the correct driver. A page frame is allocated and "pinned" for DMA transfer. The `address_space` data structure for the block device inode is updated to reflect that this page frame contains (or will contain) the image of those blocks within the disk.
• The driver is `sd`, the generic SCSI disk driver. `sd` knows that SCSI disk "a" is, let us say, target #1 on channel #0 of SCSI host controller #0. `sd` creates a SCSI READ REQUEST message and invokes the host controller driver for controller #0 to deliver that message on channel 0 to target 1. The `sd` driver also translates the offset, which is relative to partition #1, into the Logical Block Address for the disk drive. The read system call goes to sleep.
• The drive receives the request. When it has read the requested sector or group of sectors, it responds with a DATA TRANSFER message. The host controller receives this message and does a DMA operation to transfer the data into the DMA buffer page frame. Once the data transfer is complete, the controller raises an interrupt. Note that the CPU is not involved with the actual data transfer.
• The host controller driver handles the interrupt. It determines which disk I/O request just completed, and wakes up the read tasks that was sleeping on that request.
• The read system call wakes up and copies the data into the user-supplied buffer.

Interrupts

Interrupt handlers are effectively stealing CPU cycles from whatever task happens to be running at the time. Recall that interrupts can be handled either when the processor is executing user code, or when it is executing kernel code. Interrupts may interrupt the kernel in either a synchronous (fault/system call) context, or in an asynchronous one (i.e. executing another interrupt handler).

Because interrupt handlers are barging in uninvited, it is only polite that they consume as few CPU cycles as possible before executing the IRET instruction. Interrupt handlers should not perform extensive computation or data structures manipulations, and they must absolutely, positively never block.

If an interrupt handler wants to acquire a spin lock, it must be a spin lock which only interrupt handlers grab. Conversely, a spin lock which is needed by both synchronous and asynchronous kernel code must also be protected by disabling interrupts. Otherwise, a synchronous kernel routine could acquire a spin lock and then be interrupted by an interrupt handler. The handler would stall trying to obtain the spin lock, which of course would never be released because the kernel control path holding the lock has been interrupted.

Because interrupt handlers "steal" CPU time, they bypass the scheduler's policy attempts. If device drivers are coded poorly, and have bulky interrupt handlers, then heavy I/O will make the system feel unresponsive, and scheduler tuning efforts, such as using the `nice` command or system call, will not be effective, because interrupts are not scheduled, they just happen. We'll see how Linux approaches this problem with deferrable functions.

Interrupt Registration

When a device driver first initializes its hardware, the driver code registers with the kernel the interrupt vector(s) that the hardware is expected to use. When an interrupt arrives with a particular vector code, the generic kernel interrupt handler checks each registered device driver interrupt service callback routine. Each driver must check to see if its hardware caused the interrupt (e.g. by looking at a bitwise flag in the device's control register). Handling of a particular vector is serialized, e.g. while handling IRQ3, another instance of IRQ3 can not intervene. In almost all cases, the IRQ line that a particular device uses can be configured, and the kernel attempts to place each device or device group (handled by the same driver) on a unique IRQ. The pseudofile `/proc/interrupts` will list the active IRQs on your Linux system.

Interrupt Handler Synchronization

In general there are three phases to handling an interrupt (e.g. disk I/O complete).
• **Critical actions:** These must be performed with interrupts on the local processor disabled (the `CLI` instruction on X86), otherwise race conditions with severe consequences could occur. Critical actions include updating the kernel data structures which track pending interrupts, saving registers on the stack, and acknowledging the interrupt by writing to the appropriate motherboard interrupt control register. On the X86 architecture, further interrupts are already blocked when an interrupt handler is invoked.
• **Non-critical**: Some actions may be performed very quickly by the interrupt service routine with no blocking problems. After all of the critical actions take place, local interrupts can be enabled again (the `STI` instruction on X86) and these non-critical tasks could be interrupted by another handler Such non-critical tasks include: checking if the particular hardware device actually caused the interrupt, acknowledging the interrupt for

that particular device, and in some cases reading data.
• **Deferrable**: Actions which might block (e.g. shared data structure which may be locked from a system call) or which might take a long time (e.g. copying a large data buffer) are not executed during the reign of the interrupt handler. Instead, the interrupt handler notes that there is still work to be done, and allows the handler to return. At some later time, this deferred work is performed.

Deferrable Functions, tasklets and work queues

The Linux kernel has several mechanisms by which work can be deferred The basis for all of them is a **Soft IRQ**. There are a maximum of 32 soft IRQs in the (X86-32 Linux) kernel, controlled by a bitwise flag word. This is checked at periodic points within the kernel, including when returning from an interrupt. When a particular softIRQ has been activated by having its bit turned on in a device driver interrupt handler, the associated function gets called at the time that softIRQs are being checked. The softIRQ flag is cleared when the soft handler is called, but that handler may determine that the activation condition still exists and may thus turn itself back on.

One of the main ideas of deferrable functions is that asynchronous kernel code should not be able to steal too much CPU time from user processes or synchronous code. To address this, if softIRQs are called for more than 10 times in a row while returning from a hard IRQ, the kernel stops looking at them, allows the hard IRQ to return, and wakes up a kernel task called `ksoftirqd`. This task also runs any active softIRQs, ad nauseam, but because it is a task it is in a synchronous context and is subject to the whims of the scheduler. It thus competes fairly for CPU time with other tasks.

The softIRQ mechanism is used extensively within the kernel to deal with timeout-related events, in conjunction with the kernel's timebase system which allows callback functions to be established which will happen a certain number of ticks later. An example would be retranmission of network data in the TCP protocol. When the TCP segment is transmitted, a timer callback is registered, then cancelled when the ACK arrives. If there is no ACK, the timer is called back, and this activates a softIRQ which ultimately leads back to the TCP protocol routines and effects the retransmission. Note that the TCP routine is not called directly from the timer interrupt context.

The softIRQ mechanism is also used to implement **tasklets**. These allow device drivers, network protocol drivers or other kernel code to dynamically register a deferred function and activate it on demand. If there are any tasklets activated, this causes the softIRQ allocated to the tasklet subsystem to be active. Tasklets are serialized. Let's say a device driver has registered a tasklet to copy data out of a device buffer. It does not have to worry that this tasklet might be called simultaneously on two different processors.

Neither softIRQs nor tasklets may execute blocking operations (e.g. sem_dec), because they potentially execute in an interrupt context, on somebody else's time. Work queues are used when the kernel needs to perform a job which may potentially block or take a really long time. A kernel task is allocated to each work queue.

Character special devices

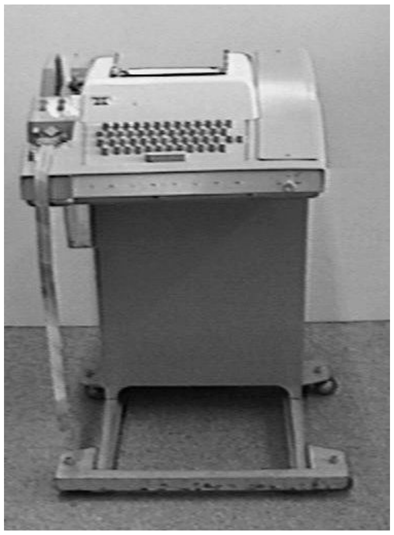
Unlike block devices, character devices are generally not random-access and not meant to store files. Examples of character devices include the keyboard, mouse, sound card, serial ports. As with block devices, when a process opens a character device special file, it is intercepted and the inode associated with the open file is one from a dummy filesystem, in this case `cdev`. Thereafter, read and write operations are handled by the device driver, which is free to impose its own semantics which may be different from ordinary file I/O (e.g. when reading from a terminal, by default read blocks until a newline is received).

Unlike block device files, read and write of character devices does not involve the page cache. It is not possible to `mmap` a character file. The methods which a character device driver exposes are: open, read, write, release (last close), ioctl.

The last method, ioctl, is a catch-all interface for any and all control operations on the device, such as setting the speed. An example of the use of `ioctl` will be given below.

The TTY

The term "tty" has its roots with the original serial printer-terminal of yesteryear, the TeleTYpe:



While the thirst for graphically-oriented user interfaces has made old news out of text-based systems, a lot of important stuff still gets done over ttys (although generally not the hard-copy style depicted above). It is certainly the most efficient way of connecting to a computer.

The role of the UNIX tty driver

Under most operating systems, the tty device driver provides command-line-oriented functionality: As characters are received, they are accumulated in a line buffer. Characters are echoed back to the "screen" (this is known as full-duplex mode). Local editing, in the form of backspace,word backspace and line redo is provided. When the end-of-line character (CR or LF) is received, then and only then is the accumulated line released to the caller of read(). This functionality is known as **line discipline**, and this specific mode of operation is known as **canonical mode**.

Below the line discipline layer is the actual hardware side of the driver. For real tty ports, this is an interface to a UART chip which implements one or more RS232 ports.

ioctl

The ioctl system call is used to get and set parameters of I/O devices, and is one of the most un-orthogonal and overloaded aspects of the UNIX filesystem interface!

```
int ioctl(fd,request,arg)
int fd,request;
char *arg;
```

fd refers to an open device special file. The *request* code is an integer which is unique at least among all the ioctl requests supported by a device driver. *arg* typically points to a structure containing arguments for the ioctl operation, although it can also just point to an int, and in some cases (antiquated usage), it isn't a pointer at all, but rather an int!

An include file, often found in `<sys/>`, defines the request codes and structures for a particular device driver or class of drivers.

To effect ioctl over ttys in UNIX, there are many ways of doing the same thing, because many different tty handling schemes have merged and diverged over the years. We'll look at the System V/Solaris termio approach, which is supported under most versions of UNIX. For more information, read the man pages `termios` and `tty_ioctl`

```
#include <termios.h>

/* struct termios, as defined in termio.h, shown below */

struct termios {
        unsigned short  c_iflag;        /* input modes */
        unsigned short  c_oflag;        /* output modes */
        unsigned short  c_cflag;        /* control modes */
        unsigned short  c_lflag;        /* line discipline modes */
        char    c_line;                 /* line discipline */
        unsigned char   c_cc[NCCS];     /* control chars */
};
```

The current terminal settings are retrieved with TCGETS and set with TCSETS:

```
struct termios t;
        if (ioctl(fd,TCGETS,&t)<0)
        {
                perror("TCGETS failed");
                return -1;
        }
        t.c_cc[VERASE]='\177';          /* Set erase char to DEL */
        if (ioctl(fd,TCSETS,&t)<0)
        {
                perror("TCSETS failed");
                return -1;
        }
```

There are many flag bits and options which will not be covered here. Some are archaic, such as the specification of how much delay should be added between the output of a Carriage Return character (ASCII code 0x0D) and more printable characters (the purpose being to allow simple mechanical printing devices time to move the print head back to the start of the line). Others are specific to RS232 hardware, such as baud rate, parity and stop bits.

Let's just look at canonical input processing. If the ICANON flag is set in *c_lflag*, then **canonical mode** input processing is set. In this mode, the device driver performs local line editing (allowing backspace, kill, etc.). When the end-of-line character is received, the line is made available. This means that read will block until the entire line is ready, then return that line. Therefore, read on a tty in canonical mode will always (unless the typist hits a lot of keys before hitting return) return fewer than the requested number of characters. Note that the newline character **is** returned by read. When an end-of-file character is received (^D by default) as the first character of a line, read returns 0.

The ECHO flag bit in *c_lflag* determines whether received characters are echoed back (full-duplex operation). This bit is turned off, for example, by the getpasswd() library routine. While in canonical mode, the *c_cc* array contains the character codes associated with various functions, such as VERASE, VWERASE, VKILL, VEOF.

When the ISIG bit is set in *c_lflag*, receipt of c_cc[VINTR] will generate a SIGINT, and receipt of c_cc[VQUIT] will generate a SIGQUIT, to be delivered to the process group which is currently attached to the terminal (see below under Job Control). A SIGINT is also generated when BRKINT is set in **c_iflags** and an RS232 line break is received.

When the ICANON bit in *c_lflag* is clear, the terminal is in **raw mode**. The characters in the *c_cc* array no longer represent VERASE, VKILL, etc. Instead, just two slots are used: VMIN and VTIME. Characters are not assembled into lines, but rather are made available to read based on the settings of c_cc[VTIME] and c_cc[VMIN] as follows:

TIME==0, MIN==0: read returns immediately with whatever characters are available (possibly 0)

TIME==0, MIN>0: read blocks indefinitely, until at least MIN characters have been read.

TIME>0, MIN==0: read returns as soon as a character is available, or when the timer expires, whichever comes first. The timer starts when read is called, and is of duration TIME*100 msec.

TIME>0, MIN>0: read returns when MIN characters are available, or it has been more than TIME*100 msec since a character has been received.

In all cases, the count argument to read represents the maximum number of characters which the caller is prepared to take. The return value is the actual number of characters read.

The same tty settings can be queried and set from the command line using the `stty` command.

Job Control and /dev/tty

Most UNIX variants provide tty job control, which facilitates running multiple jobs from the same tty line.

Job control is performed by having the tty driver maintain a **terminal process group id**, i.e.: the process group id (pgid) of the process group (job) which is in the foreground and attached to the terminal. The pgid in turn is simply the process id of the process which happens to be the group leader. A process group, or "job", is a group of interconnected processes, typically a pipeline. Input from the tty is delivered to the foreground jobs, and output from the foreground job is delivered to the tty. Furthermore, each tty is associated with a **session id**. When a tty is disconnected, by default, `SIGHUP` is delivered to all processes in that session.

The shell manipulates the terminal process group id and session id by performing `ioctls` on the tty device, and manipulates the pgid and sid values of each of its processes with the `setpgid` and `setsid` system calls.

The special pseudo-device "`/dev/tty`" is always associated with the controlling tty of the process. It is useful if it is necessary to receive input from the tty, for example, to read a password, and stdin has been redirected.

When the user hits the suspend key (c_cc[VSUSP], normally ^Z), a SIGSTOP is sent to all the processes in the foreground process group. At this point the shell, which was blocked in the wait system call, receives notification of STOPPED processes. The shell performs an ioctl to change the terminal process group id of the controlling tty back to the shell, allowing the user to type more commands. The previously running job is still stopped. If the user types "bg", the job is sent a SIGCONT and it resumes, in the background, allowing the shell to accept the next command. If the user types "fg", the job is brought back to the foreground by resetting the tty pgid to the pgid of the job, and the shell goes back to sleep pending the completion or stopping of the fg job.

When a backgrounded job (pgid != tty pgid) attempts to read or write from the tty, it is hit with SIGTTOU or SIGTTIN, since that job is not the one attached to the tty. The default result of these signals is to STOP the process(es). The job must be brought to the foreground by the shell before it can continue.

Daemons

Programs which are not intended to interact directly with a user via the terminal but instead are servers are known as `daemons`. Conventionally they are given names ending in lowercase 'd', e.g. `httpd` is the server daemon that handles HTTP requests. Most often daemons interact via network connections but there are others that are purely on the local system.

To become a daemon, the process typically forks once and allows the parent process to exit(0). The shell that invoked them thus gets an immediate return. Then the daemon must disassociate itself from the user environment. It closes all file descriptors, typically redirecting 0, 1 and 2 to a logging file, and establishes itself with a new session id and process group id, which prevents it from receiving signals relating to job control actions or tty events. It is also a good idea to `cwd` to a different directory, unique to the daemon.

Daemon processes typically exist forever. By convention, they respond to SIGTERM with a graceful shutdown. SIGHUP (there can be no "hangup" since the daemon is not connected to a tty) is conventionally used to signal a daemon to re-read its configuration files. Some daemons also respond to more complex control directives via a named pipe or network socket.

Pseudo-ttys

Ironically, pseudo-ttys are now the most common form of ttys, and real ttys are relegated to console access lines, modems, etc.

A pseudo tty is actually a pair of pseudo-devices: the slave and the master (or controller) device. The slave appears to be a tty device. ioctls which work on real ttys work on the slave device (except of course hardware-specific parameters such as baud rate). The master takes the place of the hardware-specific portion of a real tty driver.

Pseudo-ttys are commonly used in for remote network login and terminal windows under a graphical user interface such as X-Windows.

The master/controller device is opened by a process providing the "back-end" of the emulated tty, e.g. the sshd daemon. Any characters written to the master become available on the slave side as if they were received over a serial line. Similarly, any characters written to the slave side are read on the master side.

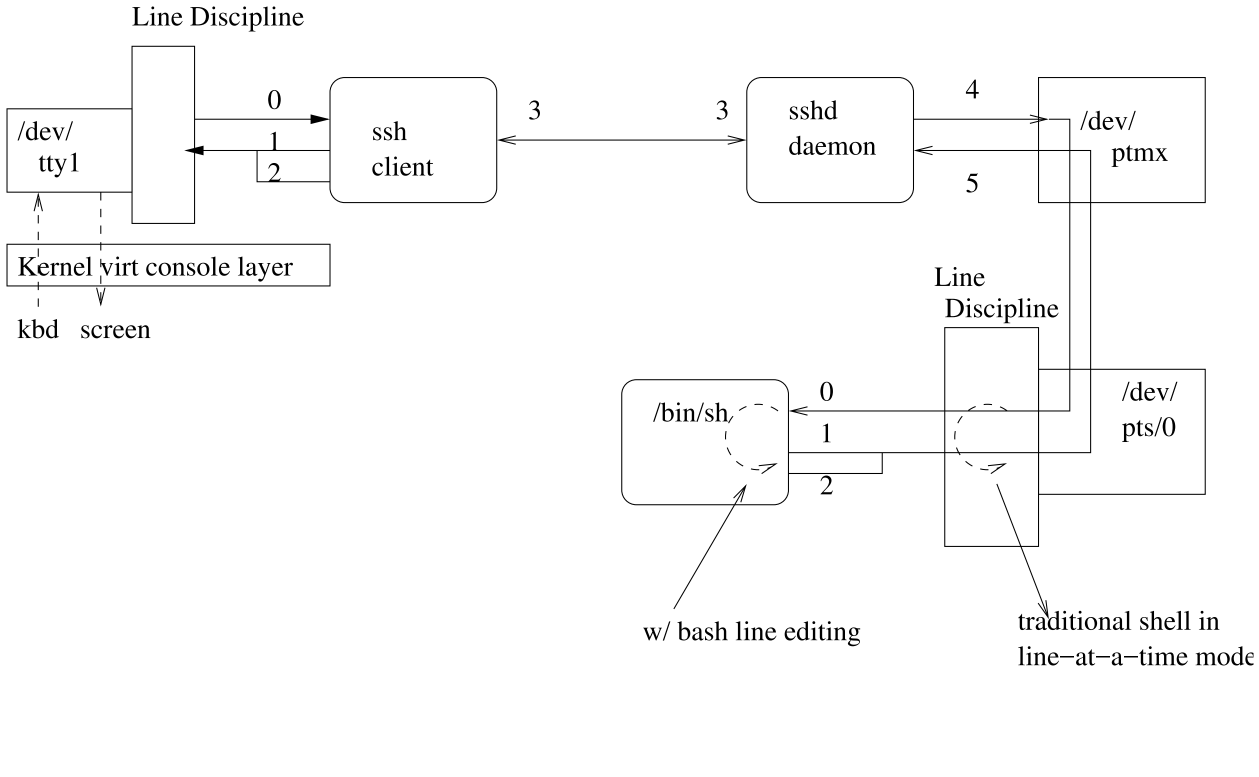
Another use of virtual ttys is the virtual console switching provided by Linux. A group of devices, `/dev/tty0`, `/dev/tty1`, etc. forms the set of virtual consoles. By pressing the Ctrl-Alt-Fn key sequence, the user can change to which virtual console the keyboard and screen are currently attached.

In the illustration below, a user on one system is using virtual screen /dev/tty1 and running the ssh client program. It has established a TCP/IP network connection to a remote system using the SSH protocol. On the remote system, the sshd daemon has handled the SSH connection. It has opened the pseudo-terminal master device `/dev/ptmx`, and has issued an ioctl to cause a pseudo terminal slave, `/dev/pts/0`, to be created.

Keystrokes entered on the left system become readable characters from `/dev/tty1`. This tty device is in non-canonical mode so that as soon as a single character is received, it is available on the file descriptor. Characters are read by the ssh client on its file descriptor 0. These characters are packaged into the SSH protocol and transmitted via the network socket on fd 3. On the right-hand system, the sshd daemon receives the SSH protocol messages and writes the characters into its fd 4. File descriptors 4 and 5, connecting to the master side of the pseudo-tty system, are connected by the driver to the slave side.

If the terminal is in canonical mode, then the line discipline module associated with `/dev/pts/0` performs local echo and line edit functions, and no data are readable on fd 0 of the shell until an entire line has been accumulated. With shells such as `bash` that perform their own line editing, the terminal is in non-canonical mode and the shell reads each character as it is received, and performs echo.

Likewise, any characters echoed back by the line discipline module, or written to file descriptors 1 or 2 of the shell, or to any child processes spawned by the shell, get passed back through the pseudo-terminal pair and are read on fd 5 by sshd, packaged into an SSH message, sent across the network socket, unpacked by the ssh client, and written to fd 1, where the virtual console tty driver displays them on the screen.


Line Discipline
/dev/
tty1
0
1
2
ssh
client
3
3
sshd
daemon
4
5
/dev/
ptmx
Kernel virt console layer
kbd
screen
Line
Discipline
/bin/sh
0
1
2
/dev/
pts/0
w/ bash line editing
traditional shell in
line−at−a−time mode