

## Device Driver Specifics

We now consider some of the details of device drivers. Each device is different with different purposes, different implementations, and different worldviews. We will consider a representative sample of several kinds of device drivers and the specifics of how they work.

### Disk Drives

Disk drivers control one or more physical magnetic disks. The most common use of disks is for file systems, but they are also used for swap space, raw backups, databases and assorted other uses. The device driver is responsible for logical → physical translations, multiplexing and demultiplexing data transfers, and for error handling.

### Block Naming

In the simplest case, logical → physical mapping is just a matter of placing an ordering on the physical disk sectors that matches the logical disk sectors, which is pretty simple. Differences between logical and physical block sizes can confuse this.

The two may differ because multiple disks managed by the same OS may have different physical block sizes. Also, because the file system overhead may depend strongly on the size of the OS's logical blocks, sometimes a logical block size much in excess of the disks sectors is chosen to keep the size of the OS tables small. One example of this is large disks using a FAT file system. The size of the FAT is directly dependent on the number of sectors (and is bounded by the size of the entry in a FAT cell). To address a large disk partition may require using large block sizes.

Error handling may also interfere with a straightforward logical → physical mapping. Some smart disk systems leave a few blocks unassigned on the disk when it is formatted for use when other blocks go bad. When a bad block is detected, the data is moved from the bad block to one of the set-aside blocks and the logical → physical mapping changed so that the logical block maps to the set-aside block rather than the bad block. From the OS point of view, this transparently repairs bad blocks.

Originally, such block-shuffling shenanigans were all done in the software, but as disk controllers (and drives) get smarter, these remappings are often done in the hardware. This can make life much trickier.

### Multiplexing and Arm Scheduling

The first issue in multiplexing multiple requests is doing so efficiently. Efficiently in this case means with returning data to the users as quickly as possible. Most of the latency in serving a disk request is in seek time, i.e., moving the disk arm over the proper track. Scheduling the requests to minimize seek time is often handled by the device driver.

Scheduling user disk requests is called arm scheduling, because the problem is essentially scheduling accesses by the disk arm. Some familiar algorithms appear, as well as some new ones. We will illustrate the scheduling algorithms by using them to order requests for the following tracks (in the order they were queued): 11, 1, 36, 16, 34, 9, 12

- **FIFO:** The requests are served in the order they appear : 11, 1, 36, 16, 34, 9, 12. This is easy to implement, but almost never used. Without optimizing at all, seek times can vary widely based on the applications making requests.
- **Shortest Seek First (SSF):** This is analogous to shortest job first, the request that requires moving the arm the least distance is served next: 11, 12, 9, 16, 1, 34, 36. The problem with this algorithm is that it encourages access patterns that keep the disk head in one place. Long accesses to distant tracks suffer very long access times, or, in the worst case, never get served at all. A compromise between optimization and fairness is needed.

- **The Elevator Algorithm** The elevator algorithm tries to keep the disk arm moving one direction. On our canonical input, with the head moving toward higher sectors, the access pattern is 11, 12, 16, 34, 36, 9 1. In practice the elevator algorithm strikes a good balance between efficiency and fairness.

There are other issues in multiplexing, mostly related to how intelligent the controller is. An intelligent controller may be able to handle several outstanding requests from the software, in which case the device driver needs to do a little bookkeeping, but can generally leave it to the controller. Of course, if the controller cannot multiplex, the simple arm scheduling above applies.

### **Error Handling**

The disk device is the first element of the OS to see an error. Accordingly, it has to adopt some strategy for which errors to try to correct and which to report. There are a variety of things that can go wrong in the disk, and Tannenbaum discusses quite a few. In most cases, the appropriate response to the error is to reset some confused part of the hardware and retry the operation.

Errors that are resolved by this are transient errors. In some cases, transient errors can be ignored. Some of them are not reproducible, and will never bother the system again. Some, however, are predictors of future woe. When a sector shows a sharp increase in checksum errors, it's likely that the sector or the disk itself is wearing out. A human being or a higher level of the operating system may want to check into the matter further. Good device drivers try to strike a balance between reporting too many and reporting too few errors.

### **Bigger, Faster, Smarter, More**

Disk drivers and controllers are getting smarter by the revision. Functionality that was traditionally in the drivers is now being moved to controllers and disk firmware. As a result, device drivers are less concerned with directly manipulating the devices as they are with programming the controllers to do so.

Some important functionalities that have begun to appear in disk hardware:

- **Interleaving.** The Berkeley fast file system did clever layout of file blocks within a track to reduce the rotational latency when the file was read sequentially. Many disks today encode this layout as the interleaving of sectors on the disk. The disk firmware rennumbers the sectors rather than the OS doing layout.
- **Caching.** Most disks and disk controllers cache one or more tracks of data in memory on the hardware. Sectors read from those tracks are read not from disk, but from the memory, eliminating the rotational and seek delays.
- **Bad blocks.** As mentioned above, some disk firmware locates and remaps bad blocks on disk directly.
- **Arm Scheduling.** Smart disk controllers, for example high-end SCSI controllers, allow several outstanding simultaneous requests for data from the same disk. The controller firmware schedules the requests internally.

In some sense, this is all positive news. Hardware is getting smarter, the software has less to do and life is great. There are two problems: the software and hardware may be unaware of each other, and burning algorithms in hardware makes them hard to change.

For an example of the hardware and software being at cross purposes, consider the disk interleaving case. The file system spends some additional time when blocks are allocated to ensure that they're placed to minimize latency, and then the hardware moves them again because of its interleaving. The result is a block layout that is almost certainly suboptimal. You can find similar problems with the other helpful features above - transparently repaired bad blocks may show up as a performance penalty; caching sectors both in memory and on disk is wasteful and degrades the value of one of the caches; spending CPU time to schedule the disk arm in the device driver only to have the controller do the same caching algorithm in hardware is a waste of CPU time.

The solution is to make sure that the hardware and OS are aware of what the other is doing. Ideally, the OS should detect hardware features and either disable the ones that are replicated in the OS, or disable

the OS routines that do work done by the controller. In practice this may be less straightforward.

The other problem, lack of flexibility, exists primarily if the features of the hardware cannot be disabled. As we have seen, for every scheduling algorithm there is a counter scheduling strategy that confuses it. If your workload is a counter strategy for the algorithms wired into your hardware, performance will suffer. Frequently its; faster to tuen or recode an algorithm in the OS than in hardware, but if you can't disable the smart feature of the hardware, you're sunk.

## Terminals

Terminal is a generic term for the keyboard/screen pair through which much of the computer input in the world today occurs. In times past, this was largely through serial line (RS-232) terminals that passed data a bit at a time, although these days a large number of terminals are intelligent or memory mapped (or both). The console keyboard, screen and mouse of a PC or workstation fall into the latter category.

Serial terminals process data conceptually process data a character or line at a time. (In reality, data may be transmitted a bit at a time down the serial wire, but the device generally collects 7 or 8 bit words to work with.) Particularly intelligent terminals may have data stream editing capabilities built in, or they may be provided by the device driver. When we say character editing capabilities, we mean everything from cursor control to simple character erasures.

To effect such editing, the device driver often collects characters as the keyboard transmits them, only committing ther characters to the standard input of the running process when the enter key is sent. Smart terminals may do the same thing in the hardware.

For output terminals have a simple language for output functions. A particular string of non-printing characters may serve to move the output cursor (the point at which the next character will be output) to a given position or to clear or scroll areas of the screen<sup>1</sup>. The device driver is responsible for arranging for canonical output control sequences to be translated into the specifi c sequences that the terminal hardware understands.

Even in a world of windowing systems and GUIs, the concept of a terminal is useful for its simplicity and power. The concepts carry over to line-driven modem systems and other simple devices. For example Coke machines.<sup>2</sup>

On the other end of the spectrum are bit-mapped displays and modern graphics processors. A bit-mapped display has its drawing memory directly accessible to the diver in a way that allows the driver to draw graphics on the screen directly. Coupled with a pointing device this allows completely graphically-oriented user interfaces (GUIs).

The screen driver directly draws the windows and other screen cues that allows a user to navigate the GUI. There are usually routines either in the kernel or in user libraries to facilitate such constructs. The interface contains simple interface elements (called Widgets or Gadgets or other things depending on whether you're on an X machine, an Amiga or some lesser machine). These libraries create the visual cues for the user and respond to events generated by the pointer driver.

The pointer driver keeps track of the user's reference point into the bit-mapped display used to manipulate GUI elements. The driver receives interrupts from the pointer whenever the pointer device is moves changing the location of the reference point. Generally any motion results in an interrupt, so the ISRs to follow the pointer must be quick. Most pointers supply only relative motion events; the device driver must track the reference point and update the gui element indicating its location.<sup>3</sup>

The pointer and drawinf routines work together to provide an event stream to the OS and applications which the applications can then use to get user input. Events are asynchronous notifi cations that contain information like "the use has activated button number 5" or "the reference pointer is over slider 2." The exact mechanisms of delivering events vary widely, and this class won't really discuss them in detail. Hopefully your understanding of IPC already has given you some ideas: a record-based fi le of events; signals that carry additional information; monitors with procedures defi ned for various events.

---

<sup>1</sup> Some terminals allow individual pixels to be addressed or vectors drawn for graphics capabilities.

<sup>2</sup> <http://www.isi.edu/~faber/coke.html>

<sup>3</sup> Some devices like touch screens and graphics pads do give absolute coordinate locations.

Beyond bit mapped displays are terminals with graphics co-processors. These are dedicated processors that do nothing but render detailed images on the terminal, perhaps employing sophisticated lighting and texture effects. The language used to communicate with such processors may be very intricate and more resembles programming a multiprocessor than running a peripheral. We won't discuss these in detail either, but again your experiences with interprocess communication should give you some ideas of the interfaces in use here. Locks and semaphores to control access to the lists of polygons to be drawn by the co-processor but arranged by the main CPU, for example. Context switching a color map on the co-processor when the reference point moves from one window to another, etc.

There are other devices in the world, too that we won't have time to investigate. Sound recording and playback systems that input and output sampled streams of data that have to be filtered in real time. Network adapters that need to fragment and reassemble large data blocks into small ones to be transferred across a network (and that may have to determine a route across such a network). Other stranger things.