Device Drivers: Classes and Services

Introduction

Device drivers represent both:

generalizing abstractions

> gathering a myriad of very different devices together and synthesizing a few general classes (e.g. disks, network interfaces, graphics adaptors) and standard models, behaviors and interfaces to be implemented by all drivers for a given class.

simplifying abstractions

> providing an implemention of standard class interfaces while opaquely encapsulating the details of how to effectively and efficiently use a particular device.

For reasons of performance and control, Operating Systems tend not to be implemented in object oriented languages (does anybody remember JavaOS?). Yet despite being implemented in simpler langauges (often C), Operating Systems, in device drivers, offer highly evolved examples of a different realization of class interfaces, derivation, and inheritance.

Whether we are talking about storage, networking, video, or human-interface, the number of available devices is huge and growing. The number and diversity of these devices creates tremendous demands for object oriented code reuse:

- We want the system to behave similarly, no matter what the underlying devices were being used to provide storage, networking, etc. To ensure this, we would like most of the higher level functionality to be implemented in common, higher level modules.
- We would like to minimize the cost of developing drivers to support new devices. This is most easily done if the majority of the functionality is implemented in common code that can be inherrited by the individual drivers.
- As system functionality and performance are improved, we would like to ensure that those benefits accrue not only to new device drivers, but also to older device drivers.

These needs can be satisfied by implementing the higher level functionality (associated with each general class of device) in common code that uses per-device implementations of a standard sub-class driver to operate over a particular device. This requires:

- deriving device-driver sub-classes for each of the major classes of device.
- defining sub-class specific interfaces to be implemented by the drivers for all devices in each of those classes.
- creating per-device implementations of those standard sub-class interfaces.

Major Driver Classes

In the earliest versions of Unix, all devices were divided into two fundamental classes, which are still present in all Unix derivatives:

block devices

These are random-access devices, addressable in fixed size (e.g. 512 byte, 4K byte) blocks. Their drivers implement a *request* method to enqueue asynchronous DMA requests. The request descriptor included information about the desired operation (e.g. byte count, target device, disk address and in-memory buffer address) as well as completion information (how much data was transferred, error indications) and a condition variable the requestor could use to await the eventual completion of the request.

A read or write request could be issued for any number of blocks, but in most cases a large request would be broken into multiple single-block requests, each of which would be passed, block at a time, through the system buffer cache. For this reason block device drivers also implement a *fsync* method to flush out any buffered writes.

character devices

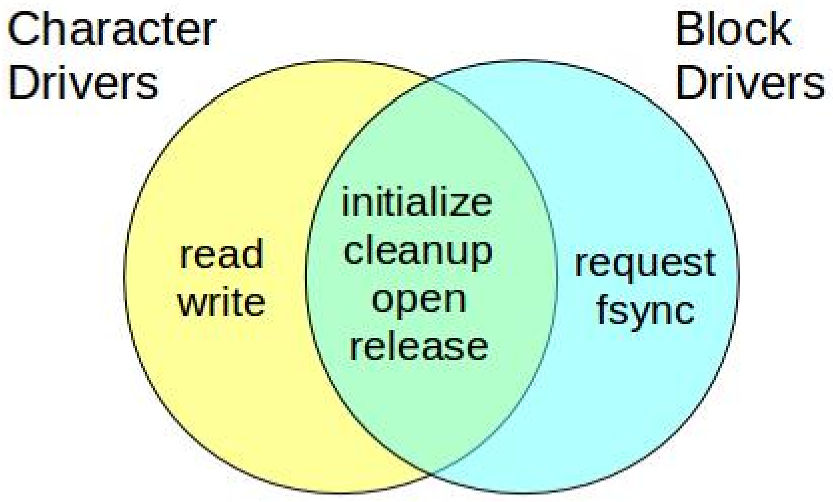
These devices may be sequential access, or may be byte-addressable. They support the standard synchronous *read(2)*, *write(2)* and (indirectly) *seek(2)* operations.

For devices that supported DMA, read and write operations were expected to be done as a single (potentially very large) DMA transfer between the device and the buffers in user address space.

A key point here is that, even in the oldest Unix systems, device drivers were divided into distinct classes (implementing different interfaces) based on the needs of distinct classes of clients:

- Block devices were designed to be used, within the operating system, by file systems, to access disks. Forcing all I/O to go through the system buffer cache is almost surely the right thing to do with file system I/O.
- Character devices were designed to be used directly by applications. The potential for large DMA transfers directly between the device and user-space buffers meant that character (or *raw*) I/O operations might be much more efficient than the corresponding requests to a block device.

These two major classes of device were not mutually exclusive. A single driver could export both block and character interfaces. A file system would be mounted on top of the block device, while back-up and integrity-checking software might access the disk through its (potentially much more efficient) character device*. All device drivers support *initialize* and *cleanup* methods (for dynamic module loading and unloading), *open* and *release* methods (roughly corresponding to the *open(2)* and *close(2)* system calls), and an optional catch-all *ioctl(2)* method.

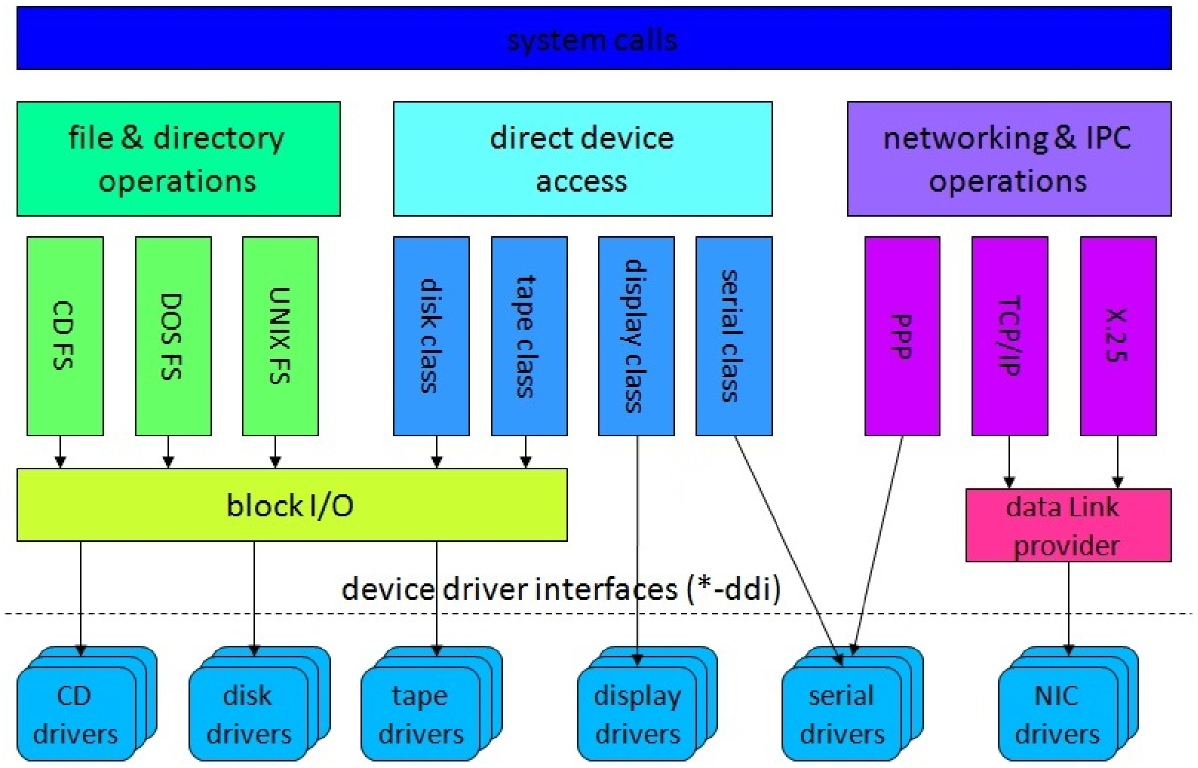


*In more contemporary systems, it is possible for a client to specify that block I/O should not be passed through the system buffer cache.

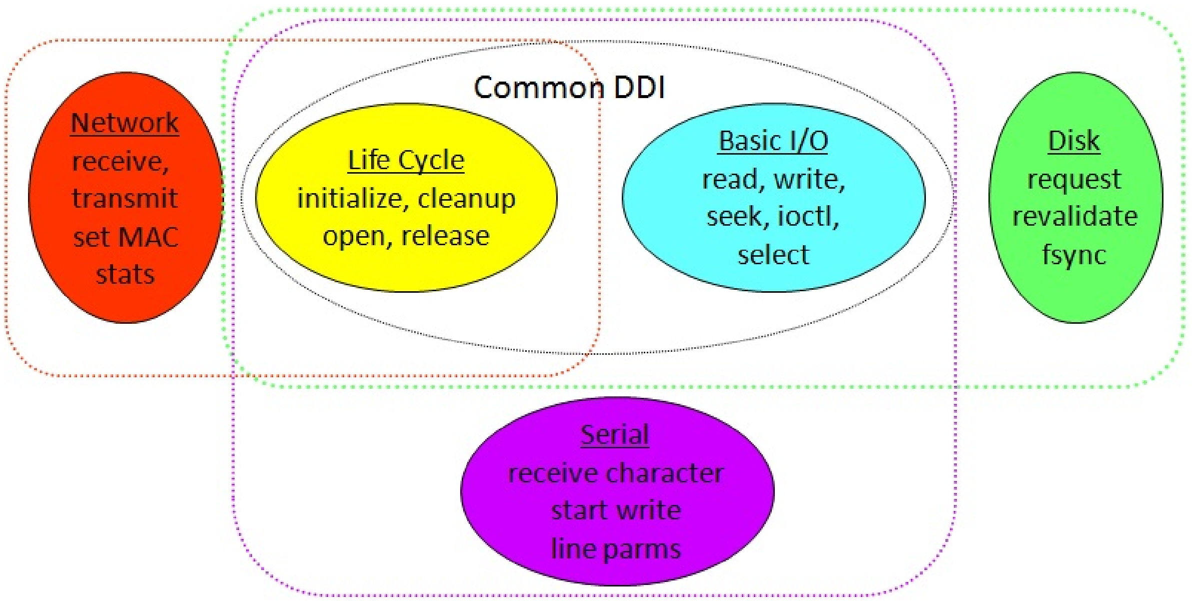
Driver sub-classes

Given the fundamental importance of file systems and disks to operating systems, it is not surprising that block device drivers would have been singled out as a special sub-class in even the earliest versions of Unix. But as system functionality evolved, the operating system began to implement higher level services for other sub-classes of devices:

- input editing and outut translation for terminals and virtual terminals
- address binding and packet sending/receipt for network interfaces
- display mapping and window management for graphics adaptors
- character-set mapping for keyboards
- cursor positioning for pointing devices
- sample mixing, volume and equalization for sound devices



And as each of these sub-systems evolved, new device driver methods* were defined to enable more effective communication between the higher level frameworks and the lower level device drivers in each sub-class. Each of these sub-class specific interfaces is referred to as a *Device Driver Interface* (DDI).



*It should be noted that in some cases, the higher frameworks have been implemented in user-mode, so that some of the new interfaces have been specified as behavior rather than new methods.

The rewards for this structure are:

- Sub-class drivers become easier to implement because so much of the important functionality is implemented in higher level software.
- The system behaves identically over a wide range of different devices.
- Most functionality enhancements to will be in the higher level code, and so should automatically work on all devices within that sub-class.

But the price for these rewards is that all device drivers must implement exactly the same interfaces:

- if a driver does not (correctly) implement the standard interfaces for its device sub-class, it will not work with the higher level software.
- if a driver implements additional functionality (not defined in the standard interfaces for its device sub-class), those features will not be exploited by the standard higher level software.

Services for Device Drivers

It is nearly impossible to implement a completely self-contained device driver. Most device drivers are likely to require a range of resources and services from the operating system:

- dynamic memory allocation
- I/O and bus resource allocation and management
- condition variable operations (wait and signal)
- mutual exclusion
- control of, and being called to service interrupts
- DMA target pin/release, scatter/gather map management
- configuration/registry services

The collection of services, exposed by the operating system for use by device drivers is sometimes referred to as the *Driver-Kernel Interface* (DKI). Interface stability for DKI functions is every bit as important as it is for the DDI entry points. If an operating system eliminates or incompatibly changes a DKI function, device drivers that depend on that function may cease working. The requirement to maintain stable DKI entry points may greatly constrain our ability to evolve our operating system implementation. Similar issues arrise for other classes of dynamically loadable kernel modules (such as network protocols and file systems).

Conclusion

Device drivers demonstrate an evolution from a basic super-class (character devices) into an ever-expanding hierarchy of derived sub-classes. But unlike traditional class derivation, where sub-class implementations inherit most of their implemntation from their parent, we see a different sort of inheritance. While each new sub-class and instance is likely to be a new implementation, what they inherit is pre-existing higher level frameworks that do do most of their work for them.