Curiosity beyond abstractions: how is bytecode executed? how do device drivers work?

Everything I've seen on *nix has been a set of abstractions off hardware, but I'm curious as to how the hardware works.
I've programmed in assembly, but that's still only a set of abstractions.

How does a processor understand assembly opcodes (as bytecode)?
How do device drivers work (with an explanation at a lower level (of abstraction))?

assembly hardware abstraction device-driver

edited May 23 '10 at 15:35

community wiki
10 revs
Yktula

I haven't read the book myself, but amazon.com/Code-Language-Computer-Hardware-Software/dp/… Code purports to answer some of your questions, at least in a basic way. – Jonathan Feinberg May 16 '10 at 22:29

I think you need to split up this question into two or more, as how a device driver works and how the processor handles bytecode is vastly different subjects, and rather big in and of themselves. Or... you could tack on "And what is the meaning of life" in there to be complete. – Lasse V. Karlsen May 17 '10 at 12:50

Tanenbaum's book is a good resource for the second part of the question: amazon.com/Structured-Computer-Organization-Andrew-Tanenbaum/dp/… I recall it includes building a CPU close to the hardware with its own instruction set etc. – wds May 17 '10 at 14:41

4 Answers

Wow.... huge question! At the very root level the processor can communicate with the hardware through special instructions e.g. IN and OUT to I/O ports on x86 hardware and/or some form of memory mapped I/O regions.

Different hardware then has very different protocols / rules as to how to communicate over these channels and in general will probably fail horribly if these rules are not followed. An example would be an output device that can only handle a limited number of transmits per second, so the driver needs to check whether the hardware is ready to send more data before trying to transmit anything. You also usually need to ensure that there are no concurrent attempts to access the same device, which is one of many good reasons why operating systems do not allow user mode programs to directly access the hardware whenever they feel like it.

Why not have a look at the Linux source code to satisfy your curiosity?

Linux kernel drivers

Note that most of this is written in C, not assembly language. There's no strict requirement to use assembly language to write device drivers providing you have the instructions available to communicate with the hardware (which is true in C, but might not be true in some higher level languages).

edited May 17 '10 at 12:54

community wiki
2 revs
mikera

Device drivers form an interface between an OS's device API and actual hardware registers.

The linux device API model is a continuation of the broader linux concept that everything is a file, and that an application can accomplish anything it needs to with the open(), read(), write(), ioctl(), and close() interface. Under the hood, there's an install() routine, but the OS decides when to call that.

The other side of the coin is hardware. The CPU accesses device registers either with special I/O instructions, or ordinary memory accesses to special memory locations that are connected to hardware. While hardware registers can act like memory, they can do things that memory cannot. Quite frequently, writing to one of a device's registers can change values some of its other registers, and to the point, can change or be changed by electrical activity in the connected hardware.

Device drivers bridge this gap. Since the possibilities for devices types are almost unlimited, it's hard to generalize about how functions are mapped, beyond just a few points. The install() routine is hit at system start up time, configures registers for proper operation, normally this includes setting up interrupt service and handling; the open() routine gives an application a logical connection to the device; there's usually an effort to make read() and write() move data in some sensible way, though sometimes you see these implemented as no-ops; and on-the-fly device settings are operated through ioctl(). And of course, the main job of close() is to undo the work of open(), taking special care to release any system resources grabbed by open().

Well, that's the linux-centric take, anyway. The windows model, at least the one I'm familiar with (probably dated), tends to offer libraries of device-specific function calls.

answered May 22 '10 at 17:38

community wiki
JustJeff

Devices have an "interrupt" -- which is how they signal that they want the processor's attention.

Device drivers have an "interrupt service routine" -- which is the code that gets executed when an interrupt occurs on that device.

Device drivers then read or write data in a low level form that maps to the device -- typically either characters or blocks of data. The higher levels of the device driver manage packing, unpacking, buffering and translating from the lower level data to higher level data, like lines of text characters, for example.

Pretty simple in it's basics, but it gets complicated real fast when you add multiple devices, multiple users, keeping track of state for both, and lots of other abstractions, like a file system, on top of basic block oriented device I/O.

edited May 18 '10 at 16:31

community wiki
2 revs
Steven D. Majewski

> Good points - worth noting though that many devices don't use interrupts, either because they are simple output-only devices or because they require the processor to poll for input. – mikera Jun 13 '10 at 14:24

"The Art of Assembly" is a good, yet kind of outdated book with explanations on pretty much everything hardware and low-level. You should give it a read.

It's available legally online and in print form.

The book, online

On amazon

EDIT: Commenter Samoz mentions a new edition, so now it's probably up to date!

answered May 22 '10 at 18:12

community wiki
Francisco P.

Actually, there's a new edition now, so it's in date! – samoz May 22 '10 at 18:14

Really? Editing. – Francisco P. May 22 '10 at 18:15