where is device driver code executed? Kernel space or User space?

Part1:

To the linux/unix experts out there, Could you please help me understanding about device drivers. As i understood, a driver is a piece of code that directly interacts with hardware and exposes some apis to access the device. My question is where does this piece of code runs, User space or Kernel space?

I know that code that is executed in kernel space has some extra privileges like accessing any memory location(pls correct if i'm wrong). If we install a third party driver and if it runs in kernel space, wouldn't this be harmful for the whole system? How any OS handles this?

Part2:

Lets take an example of USB device(camera, keyboard..), How the system recognizes these devices? how does the system know which driver to install? How does the driver know the address of the device to read and write the data?

(if this is too big to answer here, pls provide links of some good documentation or tutorials.., I've tried and couldn't find answers for these. pls help)

linux-kernel operating-system device-driver

edited Feb 3 '13 at 22:13
marko
5,348 4 13 27

asked Nov 15 '12 at 7:02
sujith
1,202 1 7 18

You should split this question to two separate ones. – icepack Nov 15 '12 at 7:20

1 In my old days of Linux device driver development, device drivers compiled inside the kernel and executed 100% kernel space. Nowadays kernels seems protecting itself from buggy device drivers and even interrupt them. But I'll take "kernel space" as an answer if it's a question on academic examinations. – Ken Cheung Nov 15 '12 at 7:26

About USB device detection, VID and PID is used to recognize the device. – Ken Cheung Nov 15 '12 at 7:31

Thanks for comments. this is not an academic exam question..:). I'm just trying to understand how this works. If possible could you please elaborate your answers and provide some links. – sujith Nov 15 '12 at 10:03

1 Answer

Part 1

On linux, drivers run in kernel space. And yes, as you state there a significant security implications to this. Most exceptions in drivers will take down the kernel, potentially corrupt kernel memory (with all manner of consequences). Buggy drivers also have an impact on system security, and malicious drivers can do absolutely anything they want.

A trend seen on MacOSX and Window NT kernels is *user-space drivers*. Microsoft has for some time been pushing the Windows Userspace Driver Framework, and MacOSX has long provided user-space APIs for Firewire and USB drivers, and class-compliant drivers for many USB peripherals. it is quite unusual to install 3rd party kernel-mode device drivers on MacOSX.

Arguably, the bad reputation Windows used to have for kernel panics can be attributed to the (often poor quality) kernel mode drivers that came with just about every mobile phone, camera and printer.

Linux graphics drivers are pretty much all implemented in user-space with a minimal kernel-resident portion, and Fuse allows the implementation of filing systems in user-space.

Part 2

USB, Firewire, MCI (and also PCI-e) all have enumeration mechanisms through which a bus driver can match the device to a driver. In practice this means that all devices expose metadata describing what they are.

Contained within the metadata is a DeviceID, VendorID and a description of functions the device provides and associated ClassIDs. ClassIDs facilitate generic Class Drivers.

Conceptually, the operating system will attempt to find a driver that specifically supports the VendorID and DeviceID, and then fall back to one that supports the ClassID(s).

Matching devices to drivers is a core concept at the heart of the Linux Device Model, and exact matching criteria used for matching is `match()` function in the specific bus driver.

Once device drivers are bound to a device, it uses the bus-driver (or addressing information given by it) to perform read and writes. In the case of PCI and Firewire, this is a memory mapped IO address. For USB it bus addressing information.

The Linux Documentation tree provides some insight into the design of the Linux Device Model, but isn't really entry-level reading.

I'd also recommend reading Linux Device Driver (3rd Edition)

edited Nov 15 '12 at 12:17

answered Nov 15 '12 at 11:57
marko
5,348 4 13 27