The Linux Device Model

Many Linux subsystems, such as the /dev filesystem, hotplug, module autoload, and microcode download have undergone significant changes with the introduction of the new device model. Learn about udev, sysfs, kobjects, classes, and more.

By Sreekrishnan Venkateswaran
Tuesday, August 15th, 2006

The Linux 2.6 kernel features a new device
model built around concepts like sysfs,
kobjects, device classes, and udev. The new model makes it easier to develop device drivers by introducing C++- like abstractions that distill commonalities from device drivers into bus and core layers. It also weeds policies out of kernel space and pushes them to user space, which has resulted in a total revamp of many features, including /dev node management, hotplug, coldplug, module autoload and firmware download.
This month, let's look at the different pieces that constiture the device model and how each piece affects key kernel subsystems. But let's start by learning about udev with the help of some examples.

Tinkering with Udev

Years ago, when Linux was young, it wasn't fun to administer device nodes. All the needed nodes (which could run into thousands) had to be statically created under the /dev directory. With the advent of the 2.4 kernels came devfs, which introduced dynamic device node creation. Devfs provided kernel interfaces to request generation of device nodes in an in-memory filesystem, but the onus of naming the nodes still rested with device drivers. However, device naming policy is administrative and doesn't mix well with the kernel. Udev arrived on the scene to push device management entirely to user space.
Udev depends on the following to do its work:
1.Kernel sysfs support, which is an important part of the Linux device model. We'll look at sysfs later on, but for now, take the corresponding sysfs file accesses for granted.
2.A set of user space daemons and utilities, such as udevd and udevinfo.
3.User-specified rules located in the /etc/udev/rules.d directory. You can write rules to get a consistent view of your devices.
To understand how to use udev, let's start off with an example. Assume that you have a USB DVD drive and a USB CD-RW drive. Depending on the order in which you hotplug these devices, one of them is assigned the name sr0, while the other gets the name sr1. During pre-udev days, you had to figure out the associated names before you could use the devices. But with udev, you can consistently view the DVD (say, as /dev/usbdvd) and the CD-RW (say, as /dev/usbcdrw) irrespective of the order in which they are plugged in or out.
First, pull product attributes from corresponding files in sysfs. Use udevinfo to collect device information:

```
$ udevinfo –a –p /sys/block/sr0
…
looking at the device chain at
'/sys/devices/pci0000:00/0000:00:1d.7/usb1/1-4':
  BUS="usb"
  ID="1-4"
  SYSFS{bConfigurationValue}="1"
  …
  SYSFS{idProduct}="0701"
  SYSFS{idVendor}="05e3"
  SYSFS{manufacturer}="Genesyslogic"
  SYSFS{maxchild}="0"
  SYSFS{product}="USB Mass Storage Device"
  …
$ udevinfo –a –p /sys/block/sr1
…
looking at the device chain at
'/sys/devices/pci0000:00/0000:00:1d.7/usb1/1-3':
  BUS="usb"
  ID="1-3"
  SYSFS{bConfigurationValue}="2"
  …
  SYSFS{idProduct}="0302"
  SYSFS{idVendor}="0dbf"
  SYSFS{manufacturer}="Addonics"
  SYSFS{maxchild}="0"
  SYSFS{product}="USB to IDE Cable"
  …
```

Now let's use the product information gleaned above to identify the devices and add udev naming rules. Create a file called /etc/udev/rules.d/40-cdvd.rules and add the rules shown in Listing One to it.
LISTING ONE: A new file for udev configuration

Software

Stick a Fork in Flock: Why it Failed

CentOS 5.6 Finally Arrives: Is It Suitable for Business Use?

Rooting a Nook Color: Is it Worth It?

System Administration

Scripting, Part Two: Looping for Fun and Profit

Command Line Magic: Scripting, Part One

Making the Evolutionary Leap from Meerkat to Narwhal

Storage

Extended File Attributes Rock!

Checksumming Files to Find Bit-Rot

What's an inode?

Mobile

Putting Text to Speech to Work

Look Who's Talking: Android Edition

Upgrading Android: A Guided Tour

HPC

A Little (q)bit of Quantum Computing

Emailing HPC

Chasing The Number

```
BUS="usb", SYSFS{idProduct}="0701", SYSFS{idVendor}="05e3",
KERNEL="sr[0-9]*", NAME="%k", SYMLINK="usbdvd"

BUS="usb", SYSFS{idProduct}="0302", SYSFS{idVendor}="0dbf",
KERNEL="sr[0-9]*", NAME="%k", SYMLINK="usbcdrw"
```

The first rule tells udev that whenever it finds a USB device with a product ID of 0×0701, vendor ID of 0x05e3, and a device name starting with sr, it has to create a node of the same name under /dev and produce a symbolic link named usbdvd to the created node. The second rule orders the creation of a symbolic link named usbcdrw for the CD-RW drive.
To test for syntax errors in your rules, run udevtest on /sys/block/sr1. To turn on verbose messages to /var/log/messages, set udev_log to yes in /etc/udev/udev.conf. To repopulate the /dev directory with the newly-added rules, restart udev using udevstart.
Once all of this is done, your DVD will consistently appear as /dev/usbdvd to the system, while your CD-RW drive will always appear as /dev/usbcdrw. Moreover, you can confidently mount them from shell scripts using commands like mount /dev/usbdvd /mnt/dvd.
Consistent naming of device nodes (and network interfaces) is not the sole capability of udev. Indeed, udev has metamorphosed into the Linux hotplug manager as well. Udev is also in charge of automatically loading modules on demand, and downloading microcode onto devices that need them. But before digging into those capabilities, let's get a basic understanding of the innards of the device model.

Kobjects, Sysfs, and Device Classes

Kobjects, sysfs, and device classes are the building blocks of the device model, but are publicity shy and prefer to remain behind the scene. Each is almost exclusively in the usage domain of bus and core implementations, and hide inside APIs that provide services to device drivers.
Kobjects introduce an encapsulation of common object properties, like usage reference counts. They are usually embedded inside larger structures. If you look at the definition of a kobject in include/linux/kobject.h, you'll notice two interesting fields:
1.A pointer to a kset, which is an object set to which the kobject belongs.
2.A ktype, which is an object type that describes the kobject.
Kobjects are intertwined with an in-memory filesystem called sysfs, which is mounted under /sys at boot time (look at /etc/fstab). Every kobject instantiated within the kernel has a representation inside sysfs. Sysfs is similar to the process filesystem (procfs) in that both are in-memory filesystems containing information about kernel data structures. While procfs is a generic window into kernel internals, sysfs is specific to the device model. Sysfs is hence not a replacement for procfs- information such as sysctl parameters and process descriptors belong to procfs and not sysfs.
Browse through /sys looking for entries that associate with say, your network card, to get a feel for its hierarchical organization. As will be apparent when you read the rest of this article, udev depends on sysfs for most of its extended functions. Sysfs is also the user space manifestation of the kernel's structured device model.
The concept of device classes is another feature of the device model, and is an interface that you're more likely to use in your device driver than the others. The class interface abstracts the idea that each device falls under a broader class (or category) of devices. A USB mouse, a PS/2 keyboard, and a joystick, all fall under the input class and own entries under /sys/class/input/.
The device class programming interface is built on top of kobjects and sysfs, so it's good place to start digging to understand the end-to-end interactions between the components of the device model. Let's take the Real Time Clock (RTC) driver as an example. The RTC driver is a miscellaneous (or "misc") driver. Misc drivers are simple character drivers that have common characteristics.
First, let's insert the RTC driver module and look at the nodes created in /sys and /dev:

```
$ modprobe rtc
$ ls –lR /sys/class/misc
drwr-xr-x 2 root root 0 Jan 15 01:23 rtc
/sys/class/misc/rtc:
total 0
-r--r--r-- 1 root root 4096 Jan 15 01:23  dev
--w------- 1 root root 4096 Jan 15 01:23  uevent
$ ls –l /dev/rtc
crw-r--r-- 1 root root 10, 135 Jan 15 01:23 /dev/rtc
```

/sys/class/misc/rtc/dev contains the major and minor numbers assigned to this device; /sys/class/misc/rtc/uevent is used for coldplugging (discussed later), while /dev/rtc is used by applications to access the RTC driver.
Now let's understand the code flow through the device model. Misc devices utilize the services of misc_register() during initialization, which looks like Listing Two if you strip off some code:
LISTING TWO: So-called "misc" devices utilize the services of misc_register() during initialization

```
/* … */
dev = MKDEV(MISC_MAJOR, misc->minor);

misc->class = class_device_create(misc_class, NULL, dev, misc->dev,
"%s", misc->name);
if (IS_ERR(misc->class))
{
err = PTR_ERR(misc->class);
goto out;
}
/* … */
```

Figure One continues to peel off more layers to get to the bottom of the device modeling. It illustrates the transitions that ripple through classes, kobjects, sysfs and udev, which result in the generation of the /sys and /dev files listed above.
FIGURE ONE: The many layers of the Linux device model

The Bus-Device-Driver Model

Another abstraction that is part of the device model is the bus/device/driver programming interface. Kernel device support is structured cleanly into buses, devices, and drivers. This renders the individual driver implementations simpler and more general. Bus implementations can, for example, search for drivers that can handle a particular device.
Consider the kernel Inter-Integrated Circuit (I2C) subsystem. The I2C core layer registers each detected I2C bus adapter using bus_register(). When an I2C client device, say an EEPROM, is probed and detected, its existence is recorded via device_register(). Finally, the I2C EEPROM client driver registers itself using driver_register().
bus_register() adds a corresponding entry to /sys/bus/, while device_register() adds entries under sys/devices/. struct bus_type, struct device, and struct device_driver are the main datastructures used respectively by buses, devices and drivers. Take a peek inside include/linux/device.h for their definitions.

Hotplug and Coldplug

Linux hotplug has recently changed considerably. Earlier, the kernel used to notify user space about hotplug events by invoking a helper program registered via the /proc filesystem. But when the latest 2.6 kernels detect hotplug, they dispatch uevents to user space via netlink sockets. Netlink sockets are an efficient mechanism to communicate between kernel space and user space via socket APIs. At the user space end, udevd receives the uevents and manages hotplug.
It's interesting to see how hotplug handling has evolved recently. A Fedora Core 3 system running a udev-039 package and a 2.6.9 kernel has a symbolic link to a udev helper under the default hotplug directory, /etc/hotplug.d/default:

```
$ ls –l /etc/hotplug.d/default/
…
lrwcrwxrwx 1 root root 14 May 11 2005 10-udev.hotplug -> /sbin/udevsend
…
```

When the kernel detects a hotplug event, it invokes a user space helper registered with /proc/sys/kernel/hotplug, which defaults to /sbin/hotplug. /sbin/hotplug receives the attributes of the hotplugged device in its environment. It runs /etc/hotplug.d/default/10-udev.hotplug after executing other scripts under /etc/hotplug/. When /sbin/udevsend thus gets executed, it passes on the hotplugged device information to udevd, which manages subsequent node creation and removal.
On a Fedora Core 4 system running udev-058 and a 2.6.11 kernel, the story changes somewhat. udevsend itself replaces /sbin/hotplug:

```
$ cat /proc/sys/kernel/hotplug
/sbin/udevsend
```

On a Fedora Core 5 machine running udev-084 and a 2.6.15 kernel, udevd assumes full responsibility of managing hotplug without depending on udevsend. Udevd now pulls hotplug events directly from the kernel via netlink sockets (see Figure One). /proc/sys/kernel/hotplug is now empty:

```
$ cat /proc/sys/kernel/hotplug
$
```

Udev also handles coldplug. Device that are connected prior to system boot are said to be "coldplugged." Since udev is part of user space and is started only after the kernel boots, a special mechanism is needed to support coldplug. The kernel creates a uevent file under sysfs for all devices and emits coldplug events to those files. When udev starts, it reads all the uevent files from /sys and emulates hotplug over the coldplugged devices.

Microcode Downloads

You have to feed microcode to some devices before they can get ready for action. A common mechanism used by drivers to access microcode was to store them in static arrays inside header files. But microcode is usually distributed as proprietary binary images by device vendors and doesn't mix homogeneously with the GPL kernel. The solution apparently is to separately maintain microcode in user space and pass them down to the kernel when required. Sysfs and udev provide an infrastructure to achieve this.
Let's take the example of the Intel PRO/Wireless 2100 Wifi adapter to walk through the new steps for downloading microcode:
1.Download the required firmware images from http://ipw2100.sourceforge.net/firmware.php and save them in /lib/firmware on your system.
2.Insert the driver module (modprobe ipw2100). The module initialization routine invokes request_firmware("ipw2100-1.3.fw").
3.This creates a sysfs directory called /sys/class/firmware/your-device/ and dispatches a hotplug uevent to user space along with the identity of the requested firmware image.
4.Udevd receives this hotplug event and responds by invoking /sbin/firmware_helper. You need a line similar to ACTION=="add", SUBSYSTEM=="firmware", RUN="/sbin/firmware_helper" in a rule file under /etc/udev/rules.d.
5./sbin/firmware_helper looks inside /lib/firmware/ to locate the requested firmware image, ipw2100-1.3.fw. It dumps the image to /sys/class/0000:02:02.0/data (0000:02:02:0 is the PCI bus/device/function identifier of the Wifi adapter in this case).
6.The driver downloads the received microcode onto the device and calls release_firmware() to free the corresponding data structures. The corresponding directory under /sys/class/firmware is also erased.
7.The driver goes through the rest of the initializations and the Wifi adapter beacons.

Module Auto-Loading

Automatically loading kernel modules on demand is a convenient feature supported on Linux. To understand how the kernel emits a “module fault” and how udev handles it, let’s insert a Xircom CardBus Ethernet adapter into a laptop:
1.During compile time, device support information is generated in the driver module object. The driver declares the identity of the devices that it knows about. Take a peek at the driver that supports the Xircom CardBus Ethernet combo card (drivers/net/tulip/xircom_cb.c) and find this snippet:

```
static struct pci_device_id xircom_pci_table[] =
{
  {0x115D, 0x0003, PCI_ANY_ID, PCI_ANY_ID,},
  {0,},
};
MODULE_DEVICE_TABLE(pci, xircom_pci_table);
```

This declares that the driver can support any card having a PCI vendor ID of 0x115D and a PCI device ID of 0×0003. When you install the driver module, the depmod utility looks inside the module image and deciphers the IDs present in the device table. It then adds the following entry to /lib/modules/kernel-version/modules.alias (v stands for VendorID, d for DeviceID, sv for subvendorID, and * for wildcard match):

```
alias pci:v0000115Dd00000003sv*sd*bc*sc*i* xircom_cb
```

2.When you hotplug the Xircom card into a CardBus slot, the kernel generates a uevent that announces the identity of the newly added device. You can look at the generated events using udevmonitor:

```
$ udevmonitor ——env
…
MODALIAS=pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00
…
```

3.Udevd receives the uevents via netlink sockets. It invokes modprobe with the MODALIAS information that the kernel passed up to it:

```
modprobe pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00
```

4.modprobe searches for a match in /lib/modules/kernel-version/modules.alias and proceeds to insert xircom_cb:

```
$ lsmod
Module      Size   Used by
xircom_cb   10433  0
…
```

The card is now ready to surf.

Looking at the Sources

The kobject implementation and related programming interfaces live in lib/kobject.c and include/linux/kobject.h. Look at drivers/base/sys.c for the sysfs implementation. The device class APIs can be found in drivers/base/class.c. Dispatching hotplug uevents via netlink sockets is accomplished in lib/kobject_uevent.c). Udev sources and documentation can be downloaded from http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html.

Sreekrishnan Venkateswaran has been working for IBM India for over ten years. His recent projects include porting Linux to a pacemaker programmer and writing firmware for a lung biopsy device. You can reach Krishnan at class="emailaddress">krishhna@gmail.com.

Comments are closed.