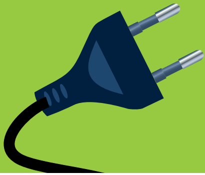
udev Unplugged!



Find out what's up with this geeky utility called udev, and in the process learn how to auto connect to the Internet as soon as you plug in that USB modem. Or take a back-up of your home directory to start automatically as soon as you connect an external hard drive.

You plug your back-up hard disk in! After a few seconds, you get a notification: "Back-up is complete." You then unplug the hard drive and your back-up for the day is ready with you. Now imagine this: you plug your EVDO/CDMA Internet data card in, and within a few seconds you get a notification: "Internet connected." When the device is unplugged, you get a message stating the Net is disconnected. Can you ever think of such a user experience under GNU/Linux?

Of course you can! *udev* helps you achieve this and a lot more. Let's tune into what's so great about *udev!*

What is *udev?*

udev is a device manager for Linux that runs in user space. It deals with device node creation, while taking care of the persistent naming of devices upon the availability of real hardware.

By the UNIX concept, everything is a file. We access our devices via corresponding files in the */dev* directory. As you know, */dev* is a directory containing device nodes for all standard devices. Traditional UNIX systems had static device nodes under the */dev* directory. What happens when you plug your MP3 player in the USB port?

You might have noticed that it is */dev/sda1,* or some other node, through which you access the contents of the filesystem. */dev/sda1* is a device node corresponding to that device. This kind of static device node system worked fine, since there were a limited number of devices in earlier times. The existence of these device nodes was independent of actual devices connected to the hardware. It was a real hassle to decide whether a piece of hardware existed or not, since all possible device nodes existed.

Now, as the number of Linux-supported devices increased, especially USB removable devices and IEEE 1394 (Firewire ports), the number of static nodes required under */dev* increased to a huge number—nearly

18,000—and it became unmanageable. Also, if some device nodes that corresponded to a connected device did not exist under */dev*, you had to Google for the major and minor number for the device, and create the device node manually, using the *mknod* command. Since each device has its unique major and minor numbers, this was a pretty tough situation!

As a result, a pseudo RAM-based filesystem *sysfs,* mounted under */sys,* was introduced. Users now could check whether a device existed or not, by looking into the directory tree of devices under /sys. Still, this wasn't a satisfactory solution, since either of the devices were statically built, or we had to create the device nodes manually, using major and minor numbers for the corresponding device.

Give the following *tree* command a try:

```
[slynux@gnubox ~]$ tree /sys/class/
```

Since our area of interest is making life easier with *udev*, let's move on to hacking *udev*.

udev runs in the memory all the time as a daemon and listens to kernel messages. The kernel always sends a message whenever it notices a hardware change. You can observe it by running the *dmesg* command. The following is the *dmesg* output when I connect an external hard disk:

```
# dmesg | tail

sd 5:0:0:0: [sdb] Attached SCSI disk
sd 5:0:0:0: Attached scsi generic sg2 type 0
kjournald starting.  Commit interval 5 seconds
EXT3 FS on sdb, internal journal
EXT3-fs: recovery complete.
EXT3-fs: mounted filesystem with ordered data mode.
```

So, let's take a look at the duties of *udev*:

- Listen to kernel messages. If some device is connected, create its device nodes according to the order in which it is connected. *udev* has the ability to identify each of the devices uniquely. Device nodes are created only when the device is connected.
- Removal of device nodes when the device is unplugged.
- Create symlinks for device nodes, and execute commands upon *udev* events.
- Follow the *udev* rules. The *udev* daemon is controlled by a set of user-specified rules.

Consider the following scenario, with which I'll try to elaborate the usefulness of *udev*. Let's suppose you have two printers—one an inkjet and the other a laser colour printer. Usually, the one that is connected first is designated as */dev/lp0* and the second one */dev/lp1*. How do you understand which one is laser and which one is inkjet? Is it by looking at which one is switched on first?

udev is brilliant in solving such nonsense. What if you are able to get */dev/laser* for the laser printer and */dev/dotmat* for the dot matrix printer. *udev* can identify each of the devices uniquely by specifying certain parameters through *udev* rules.

Rules explained!

The behaviour of udev on handling each of the devices can be controlled by using udev rules. Most of the newer distros ship with a number of default udev rules meant for hardware detection. When deciding how to name a device and which additional actions to perform, udev reads a series of rule files. These files are kept in the */etc/udev/rules.d* directory, and they all must have the *.rules* suffix. In a rules file, lines starting with "#" are treated as comments. Every other non-blank line is a rule and rules cannot span multiple lines. The default rules file can be seen at /etc/udev/rules.d/50-udev-default.rules

A rule consists of a combination of matching keys for the device and the action to be done on matching the device. In other words, a rule explains how to find the specific device and what to do when it is found.

The following is the basic syntax of a rule:

```
KEY1="value", KEY3="value", KEY4=="value"...SYMLINK+="link"
```

The following line is a simple *udev* rule. It tells the *udev* daemon to create */dev/cdrom* and */dev/cdrom0* softlinks to */dev/hdc* whenever it finds */dev/hdc*.

```
KERNEL=="hdc", SYMLINK+="cdrom cdrom0"
```

It is to be remembered that we can specify multiple rules for a single device and it can be written in multiple *.rules* files. When a device is plugged in or unplugged, the *udev* daemon looks through all the *.rules* files in the */etc/udev/rules.d* directory until all matching rules are read and executed.

The following are some of the keys or parameters that can be used for device matching and the actions in a *udev* rule:

- BUS: matches the bus type of the device; examples of this include PCI, USB or SCSI.
- KERNEL: matches the name the kernel gives the device.
- ID: matches the device number on the bus; for example, the PCI bus ID or the USB device ID.
- PLACE: matches the topological position on the bus, such as the physical port a USB device is plugged in to.
- SYSFS_filename, SYSFS{filename}: allows *udev* to match any *sysfs* device attribute, such as the label, vendor, USB serial number or SCSI UUID. Up to five different *sysfs* files can be checked in a single rule, with all of the values being required in order to match the rule.
- PROGRAM: allows *udev* to call an external program and check the result. This key is valid if the program returns successfully. The string returned by the

TABLE 1: OPERATIONS FOR UDEV KEYS	
Operator	**Meaning**
==	For matching. Eg: KERNEL=="ttyUSB0"
=	Setting a parameter. Eg: NAME="my_disk"
+=	Adding to list. Eg: SYMLINK+="cd1 cd2"

program additionally may be matched with the RESULT key.

- ATTR: different attributes for the device like size, product ID, vendor, etc.
- RESULT: matches the returned string of the last PROGRAM call. This key may be used in any rule following a PROGRAM call.
- RUN: it can be set to some external program that can be executed when a device is detected.
- SYMLINK: for creating symlinks for the matching device.
- ACTION: permits two match conditions 'add' and 'remove' when a new device is added or removed.

In addition to this, Table 1 lists different operators you can use with each of the keys.

Now the question is: how do we collect information about devices?

Writing a rule is, in turn, matching the device by specifying unique bytes about the device. The unique information about the device can be grabbed from *sysfs*:

```
[slynux@gnubox tmp]$ cat /sys/block/sda/sda1/size
14336000
```

Here I have retrieved an attribute size for the device *sda1*. Now I can use ATTR{size}=="14336000" to match the device */dev/sda1*.

To make the job easier, we have a *udev* utility called *udevinfo,* which can be used to collect details about devices and write rules in a very handy way. The following is the *udevinfo* output for the same */dev/sda1*:

```
# udevinfo -a -p /sys/block/sda/sda1
 looking at device '/devices/pci0000:00/0000:00:1f.2/host0/
target0:0:0/0:0:0:0/block/sda/sda1':
    KERNEL=="sda1"
    SUBSYSTEM=="block"
    DRIVER==""
    ATTR{dev}=="8:1"
    ATTR{start}=="2048"
    ATTR{size}=="14336000"
    ATTR{stat}=="    190    59    2162    1655    30    31    488    836
0    1652    2491"
```

As you can see, it returned a lot of information about the device. We will take some of the above lines to make a *udev* rule:

```
KERNEL=="sda1" , SUBSYSTEM=="block" , ATTR{dev}=="8:1", 2048"
```

Now the match is ready! You can even create a symlink for the device as */dev/musicdrive*:

```
KERNEL=="sda1" , SUBSYSTEM=="block" , ATTR{dev}=="8:1", 2048" ,
SYMLINK+="musicdrive"
```

Alternatively, you can use the following to obtain information about any device name:

```
# udevinfo -a -p `udevinfo -q path -n /dev/devicename`
```

Setting up an automatic Internet connection

I depend on BSNL EVDO/CDMA for Internet access. I have configured the dialling by using the *wvdial* PPP utility, and I issue the *wvdial* command to connect under Fedora 9. I found it interesting to write *udev* rules to auto connect Internet whenever I plug in the EVDO USB modem.

Here's how to get started: first, plug in the EVDO device in the USB port; second, run the *dmesg* command at a terminal prompt. I received the following *dmesg* output:

```
usb 6-2: New USB device found, idVendor=05c6, idProduct=6000
usb 6-2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 6-2: Product: ZTE CDMA Tech
usb 6-2: Manufacturer: ZTE, Incorporated
```

But there was no suitable kernel module loaded to create */dev/ttyUSB0*, which is the device node for the corresponding device. You might try manually loading the USB serial module specifying the Product ID and vendor ID parameters—that is, idVendor=05c6, idProduct=6000. Run the following command as the root user:

```
/sbin/modprobe usbserial product=0x6000 vendor=0x05c6
```

Executing the *dmesg* command again brings up the following:

```
usb 6-2: configuration #1 chosen from 1 choice
usbserial_generic 6-2:1.0: generic converter detected
usb 6-2: generic converter now attached to ttyUSB0
```

As you can see, this time */dev/ttyUSB0* is created and made available. [Actually when the module *usbserial* is loaded using the *modprobe* command, it is required to manually create */dev/ttyUSB0* using the *mknod* command. But there is a default *udev* rule that creates the device.]

Now we have to dial *wvdial* as the root in order to connect. How do we transform this manual process to a *udev* rule? Run the following command to collect appropriate parameters to match the device:

```
udevinfo -a -p $(udevinfo -q path -n /dev/ttyUSB0)
```

Now, create a file called */etc/udev/rules.d/100-bsnl.*

rules and enter the following rules in it:

```
ATTRS{idVendor}=="05c6" , ATTRS{idProduct}=="6000", RUN+="/
sbin/modprobe usbserial product=0x6000 vendor=0x05c6",
SYMLINK+="netdevice"

ACTION=="add", SUBSYSTEM=="tty",KERNEL=="ttyUSB0",
ATTRS{idVendor}=="05c6" , ATTRS{idProduct}=="6000", RUN+="/usr/bin/
evdo_connect"

ACTION=="remove", SUBSYSTEMS=="usb", KERNEL=="ttyUSB0", RUN+="/
usr/bin/msg_connection"
```

The first rule instructs *udevd* to listen to devices with parameters idVendor=05c6 and idProduct=6000. If found, load the corresponding *usbserial* kernel module. The second rule instructs *udevd* to execute the *evdo_connect* script when the above parameters match for a newly added device */dev/ttyUSB0*. ACTION="add" means, when the device was added.

The parameter value for RUN is an executable command. But it should be noted that the executable should be something that runs finite times rather than something that contains an infinite loop or infinite conditions.

/usr/bin/evdo_connect is made to run for a finite number of times by sending *wvdial* and *msg_connection* to the background.

Now, create two files. In the first file named */usr/bin/evdo_connect* enter the following text:

```
#!/bin/bash
/usr/bin/wvdial &
/usr/bin/msg_connection con &
```

…and in the second file named */usr/bin/msg_connection,* enter the following:

```
#!/bin/bash
user=slynux ; # Specify the user to which notification is to be shown

if [ $# -eq 0 ];
then
        DISPLAY=:0 su $user -c 'notify-send -u critical "Internet
Disconnected :("' ;
else

while true;
do

if [[ -n $(/sbin/ifconfig ppp0 2>&1 | grep "inet addr") ]];
    then
        DISPLAY=:0 su $user -c 'notify-send "Connected to Internet :)"' ;
    exit 0;
fi
    sleep 1;
```

Figure 1: 'Connected to Internet' notification

Figure 2: 'Internet disconnected' notification

```
done

fi
```

In this script, we have used the *notify-send* utility to display messages to the user. *notify-send* comes default with Fedora 9. You may have to install it separately on Ubuntu or other distributions.

Now, set executable permissions to both the scripts since *udev* is going to execute them upon finding the device:

```
# chmod +x /usr/bin/evdo_connect
# chmod +x /usr/bin/msg_connection
```

Voila! The auto dialling is configured and ready to run. As soon as I plug or unplug EVDO now, I get notifications as shown in Figures 1 and 2, in real time.

The procedure is the same while using any other mobile/CDMA Net connection. You have to modify the *udev* rules slightly, according to your device parameters.

Auto syncing a back-up drive

Let's look at a typical problem: I have a back-up hard drive. I used to back up my home directory everyday in this hard disk. This is normally done manually so, again, let's use *udev* to automate the procedure. Again, as we did with the EVDO modem, first plug in the external hard drive. Then

run *dmesg* to identify the device. The following is the *dmesg* output in my case:

```
usb-storage: device scan complete
scsi 7:0:0:0: Direct-Access     HITACHI_ DK23DA-20        00J2 PQ: 0 ANSI: 0
sd 7:0:0:0: [sdb] 39070079 512-byte hardware sectors (20004 MB)
sd 7:0:0:0: [sdb] Write Protect is off
sd 7:0:0:0: [sdb] Mode Sense: 03 00 00 00
sd 7:0:0:0: [sdb] Assuming drive cache: write through
sd 7:0:0:0: [sdb] 39070079 512-byte hardware sectors (20004 MB)
sd 7:0:0:0: [sdb] Write Protect is off
```

Now, collect suitable keys to match the device using the following command:

```
# udevinfo -a -p $(udevinfo -q path -n /dev/sdb) | more
```

The output in my case was:

```
 looking at device '/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-1:1.0/
host7/tar
get7:0:0/7:0:0:0/block/sdb':
    KERNEL=="sdb"
    SUBSYSTEM=="block"
    DRIVER==""
    ATTR{dev}=="8:16"
    ATTR{range}=="16"
    ATTR{removable}=="0"
    ATTR{size}=="39070079"
    ATTR{capability}=="12"
    ATTR{stat}=="     51     285     456     340       1       0       8
       9       0     278     349"

 looking at parent device '/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-
1:1.0/ho
st7/target7:0:0/7:0:0:0/block':
    KERNELS=="block"
    SUBSYSTEMS==""
    DRIVERS==""

 looking at parent device '/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-
1:1.0/ho
st7/target7:0:0/7:0:0:0':
    KERNELS=="7:0:0:0"
    SUBSYSTEMS=="scsi"
    DRIVERS=="sd"
    ATTRS{device_blocked}=="0"
    ATTRS{type}=="0"
    ATTRS{scsi_level}=="0"
    ATTRS{vendor}=="HITACHI_"
    ATTRS{model}=="DK23DA-20       "
```

Now formulate a matching rule as the following and write to a rule file [we'll call it */etc/udev/rules.d/100-backupdisk.rules*]:

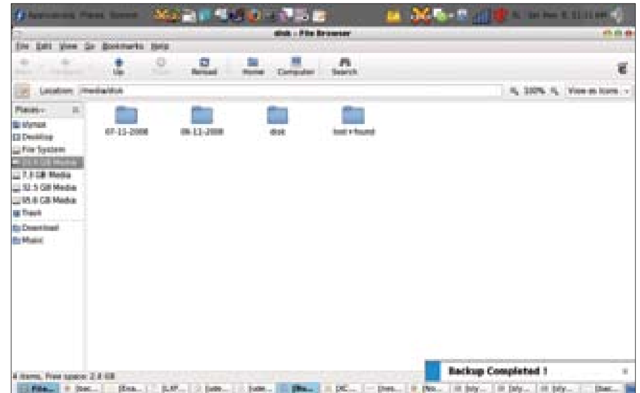


Figure 3: Back-up completed notification

```
SUBSYSTEM=="block", ATTR{removable}=="0", ATTR{size}=="39070079",
SYMLINK+="backupdisk", RUN+="/usr/bin/backup"
```

We have an action script */usr/bin/backup*, which is called when a match is found. Write a bash script with the following contents:

```
#!/bin/bash

backup_dir=/home/slynux  # Specify the directory to backup
user=slynux # The user to whom which the message is to be displayed

mount /dev/backupdisk /mnt/backups;

rsync -a $backup_dir /mnt/backups/$(date +%d-%m-%Y)/ ;

umount /mnt/backups ;

DISPLAY=:0 su $user-c 'notify-send "Backup Complete"';
```

Notice that the script mounts the external disk under */mnt/backup*. So, make sure you create that directory as well.

Following this, make the script executable as follows:

```
# chmod +x /usr/bin/backup
```

That's it! Now, every time you connect the external disk, it starts the back-up procedure using *rsync* automatically. Once the procedure ends, you will get a pop-up notification on your desktop as well (Figure 3).

You can tweak around a bit to make this back-up drive encrypted as well. However, I'll leave you to try it out yourself.

So, that's all for now. Have fun with *udev*, and happy hacking! END

***By: Sarath Lakshman** is an 18 year old hacker and free software enthusiast from Kerala. He loves working on the GNU/Linux environment and contributes to the PiTiVi video editor project. He is also the developer of SLYNUX, a distro for newbies. He is currently studying at Model Engineering College, Cochin. He blogs at www.sarathlakshman.info*