Hotpluggable devices and the Linux kernel

Greg Kroah-Hartman
greg@kroah.com

Abstract

Hotpluggable devices are becoming more common for portable computers, desktop computers, and embedded systems. Linux has had support for PCMCIA devices for quite a while, but with the advent of USB and Firewire devices today, and the needed support for hot plug PCI in the future, the Linux kernel has had to change to handle these new requirements. In older kernels, devices were determined at boot time, or module load time, but now the kernel has to handle devices coming and going at any moment. It also needs to have a mechanism for loading and unloading the drivers for those devices automatically.

1 Introduction

Hotpluggable devices have been created to solve a number of user needs. On laptop computers, PCMCIA devices were designed to allow the user to swap cards while the computer was still running. This allowed people to change network adaptors, memory cards, and even disc drives without shutting down the machine.

The success of this led to the creation of the USB and IEEE-1394 (Firewire) buses. These designs allow for peripherals to be attached and removed at any point. They were also created to try to move systems away from the ISA bus, and to a fully "Plug and Play" type system.

From the operating system's point of view, there are many problems with hotplugging devices. In the past, the operating system has to only search for the various devices connected to it on power up, and once seen the device never goes away. From the view of the device driver, it never expects to have the hardware that it is trying to control disappear. But with hotpluggable devices, all of this changes. Now the operating system has to have a mechanism to constantly detect if a new device appears. This is usually done by a bus specific manager. This manager handles the scanning for new devices, and recognizes when a device has disappeared. It must be able to create system resources for the new device, and pass control off to a specific driver. The device driver for a hotpluggable device has to be able to gracefully recover when the hardware is removed and be able to bind itself to new hardware at any moment.

This paper describes the new framework in the Linux kernel for supporting USB and other hotpluggable devices. It will cover how the past implementation of PCMCIA loaded its drivers, and the problems of that system. It will present the current method of loading USB and PCI drivers, and how it handles the user configuration issues better than PCMCIA.

2 The Past

Linux has had support for PCMCIA since 1995. In order for the PCMCIA core to be able to load drivers when a new device was inserted, it had a userspace program called `cardmgr`. The `cardmgr` program would receive notification from the kernel's PCMCIA core when a device had been inserted or removed and use that information to load or unload the proper driver for that card. It used a configuration file located at `/etc/pcmcia/config` to determine which driver should be used for which card. This configuration file needed to be kept up to date with the what driver supported which card, or ranges of cards, and has grown to be over 1500 lines long. Whenever a driver author added support for a new device, they have to modify two different files to enable the device to work properly.

As the USB core code became mature, the group realized that it too needed something like the PCM-

CIA system to be able to dynamically load and unload drivers when devices were inserted and removed. The group also noted that since USB and PCMCIA both needed this system, and that other kernel hotplug subsystems also would use such a system, a generic hotplug core would be useful. David Brownell posted an initial patch to the kernel[2], enabling it to call out to a userspace program called `/sbin/hotplug`. This patch was eventually accepted, and other subsystems were modified to take advantage of it. Then, as the maintaining of an external configuration file describing which drivers worked for which devices started to become burdensome, a method of automatically creating the configuration data from the drivers themselves was implemented.

3 /sbin/hotplug

The kernel hotplug core provides a method for the kernel to notify userspace that something has happened. It does that by calling the executable listed in the global variable `hotplug_path`. When the kernel starts, `hotplug_path` is set to `/sbin/hotplug` but this can be changed by the user by modifying the value at `/proc/sys/kernel/hotplug`. `/sbin/hotplug` is executed by the kernel function `call_usermodehelper`[3].

As of kernel 2.4.4, this `/sbin/hotplug` method is being used by the PCI, USB and Network core subsystems. As time goes on, more subsystems will be converted to use it (patches are already available for SCSI.)

The PCI, USB, and Networking subsystems all call `/sbin/hotplug` with different environment variables set, depending on what action has just occurred.

3.1 PCI

PCI devices call `/sbin/hotplug` with the following arguments:

```
argv [0] = hotplug_path
argv [1] = "pci"
argv [2] = 0
```

And the environment is set to the following:

```
HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
PCI_CLASS=class_code
PCI_ID=vendor:device
PCI_SUBSYS_ID=subsystem_vendor:subsystem_device
PCI_SLOT_NAME=slot_name
ACTION=action
```

Where `action` is "`add`" or "`remove`" depending on if the device is being inserted or removed from the system, and `class_code`, `vendor`, `subsystem_vendor`, `subsystem_device`, and `slot_name` represent the numerical values for the PCI device's information.

3.2 USB

USB devices call `/sbin/hotplug` with the following arguments:

```
argv [0] = hotplug_path
argv [1] = "usb"
argv [2] = 0
```

And the environment is set to the following:

```
HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
ACTION=action
PRODUCT=idVendor/idProduct/bcdDevice
TYPE=device_class/device_subclass/device_protocol
```

Where `action` is "`add`" or "`remove`" depending on if the device is being inserted or removed from the system, and `idVendor`, `idProduct`, `bcdDevice`, `device_class`, `device_subclass` and `device_protocol` are filled in with the information from the USB device's descriptors.

If the USB device's deviceClass is 0 then the environment variable `INTERFACE` is set to:

```
INTERFACE=class/subclass/protocol
```

If the USB subsystem is compiled with the `usbdevfs` filesystem enabled, the following environment variables are also set:

```
DEVFS=/proc/bus/usb
DEVICE=/proc/bus/usb/bus_number/device_number
```

Where `bus_number` and `device_number` are set to the bus number and device number that this specific USB device is assigned.

3.3 Network

The network core code also calls `/sbin/hotplug` whenever a network device is registered or unregistered with the network subsystem[6]. `/sbin/hotplug` is called with the following arguments when called from the network core:

```
argv [0] = hotplug_path
argv [1] = "net"
argv [2] = 0
```

And the environment is set to the following:

```
HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
INTERFACE=interface
ACTION=action
```

Where `action` is "`register`" or "`unregister`" depending on what happened in the network core, and `interface` is the name of the interface that just had the action applied to itself.

3.4 simple example

`/sbin/hotplug` can be a very simple script if you only want it to control a small number of devices. For example, if you have a HandSpring Visor that you do not want to compile the module into the kernel to save memory, yet you would like the module to be automatically loaded whenever the device is plugged in and unloaded whenever the device is removed, the following script would be sufficient:

```
#!/bin/sh
if [ "$1" = "usb" ]; then
    if [ "$PRODUCT" = "82d/100/0" ]; then
        if [ "$ACTION" = "add" ]; then
            /sbin/modprobe visor
        else
            /sbin/rmmod visor
        fi
    fi
fi
```

If you want to add support for a USB Bluetooth device the script could be modified to look like:

```
#!/bin/sh
if [ "$1" = "usb" ]; then
    if [ "$ACTION" = "add" ]; then
        PROGRAM="/sbin/modprobe"
    else
        PROGRAM="/sbin/rmmod"
    fi
    if [ "$PRODUCT" = "82d/100/0" ]; then
        $PROGRAM visor
        exit 0;
    fi
    if [ "$INTERFACE" = "e0/01/01" ]; then
        $PROGRAM bluetooth
        exit 0;
    fi
fi
```

4 Need for automation

The previous small example shows the limitations of being forced to manually enter in all of the different device ids, product ids, and such in order to keep a `/sbin/hotplug` script up to date with all of the different devices that the kernel knows about. Instead, it would be better for the kernel itself to specify the different types of devices that it supports in such a way that any userspace tools could read them. Thus was born a series of complex macros that are used by all USB and PCI drivers. These macros describe which devices each specific driver can support. At compilation time, the build process extracts this information out of the driver, and builds a table. The table is called `modules.pcimap` and `modules.usbmap` for all PCI and USB devices respectively. How to use these table's data will be described in section 5.

For example, the following code snippet from `drivers/usb/uhci.c`[8]:

```
static const struct pci_device_id
  __devinitdata uhci_pci_ids[] = { {

    /* handle any USB UHCI controller */
    class:          ((PCI_CLASS_SERIAL_USB << 8) | 0x00),
```

```
    class_mask:     ~0,

    /* no matter who makes it */
    vendor:         PCI_ANY_ID,
    device:         PCI_ANY_ID,
    subvendor:      PCI_ANY_ID,
    subdevice:      PCI_ANY_ID,

    }, { /* end: all zeroes */ }
};
MODULE_DEVICE_TABLE (pci, uhci_pci_ids);
```

causes this line to be added to the `modules.pcimap` file:

```
uhci 0xffffffff 0xffffffff 0xffffffff 0xffffffff 0x000c0300 0xffffffff 0x00000000
```

As the example shows, a PCI device can be specified by any of the same paramaters that is passed to the `/sbin/hotplug` program.

A USB device can specify that it can accept only specific devices such as this example from `drivers/usb/serial/whiteheat.c`[9]:

```
static __devinitdata struct
  usb_device_id id_table_combined [] = {
    { USB_DEVICE(CONNECT_TECH_VENDOR_ID,
                 CONNECT_TECH_WHITE_HEAT_ID) },
    { USB_DEVICE(CONNECT_TECH_VENDOR_ID,
                 CONNECT_TECH_FAKE_WHITE_HEAT_ID) },
    { }  /* Terminating entry */
};

MODULE_DEVICE_TABLE (usb, id_table_combined);
```

which causes the following lines to be added to the `modules.usbmap` file:

```
whiteheat 0x0003 0x0710 0x8001 0x0000 0x0000 0x00 0x00 0x00 0x00 0x00 0x00 0x00000000
whiteheat 0x0003 0x0710 0x0001 0x0000 0x0000 0x00 0x00 0x00 0x00 0x00 0x00 0x00000000
```

or it can specify that it accepts any device that matches a specific USB class code, as in this example from `drivers/usb/bluetooth.c`[1]:

```
static struct usb_device_id
  usb_bluetooth_ids [] = {
    { USB_DEVICE_INFO(WIRELESS_CLASS_CODE,
                      RF_SUBCLASS_CODE,
                      BT_PROTOCOL_CODE) },
    { }  /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, usb_bluetooth_ids);
```

which causes the following line to be added to the `modules.usbmap` file:

```
bluetooth 0x0070 0x0000 0x0000 0x0000 0x0000 0xe0 0x01 0x01 0x00 0x00 0x00 0x00000000
```

Again these USB examples show that the information in the `modules.usbmap` file matches the information provided to `/sbin/hotplug` by the kernel, enabling `/sbin/hotplug` to determine which driver to load without relying on a hand generated table, like PCMCIA relies apon.

5 How automation works

The macro `MODULE_DEVICE_TABLE`[5] automatically creates two variables. For the example:

```
MODULE_DEVICE_TABLE (usb, usb_bluetooth_ids);
```

the variables `__module_usb_device_size` and `__module_usb_device_table` are created. `__module_usb_device_size` contains the value of the size of the `struct usb_id` structure, and `__module_usb_device_table` points to the `usb_bluetooth_ids` structure.

The `usb_bluetooth_ids` variable is an array of `usb_id` structures, with a terminating NULL structure at the end of the list. The individual structure is filled with one of the following macros:

```
USB_DEVICE(vendor_id, product_id)
USB_DEVICE_VER(vendor_id,product_id, low, high)
USB_DEVICE_INFO(class, subclass, protocol)
USB_INTERFACE_INFO(class, subclass, protocol)
```

which fills up the `usb_id` with the proper device, class, or interface class information, depending on what the driver supports.

When the `depmod` program is run, as part of the kernel installation process, it goes through every module looking for the symbol `__module_usb_device_size` to be present in the compiled module. If it finds it, it copies the data pointed to by the `__module_usb_device_table` symbol into a structure, extracts out all of the information, and writes it out to the `modules.usbmap` file in the module root directory. It does the same thing while looking for the `__module_pci_device_size` in creating the `modules.pcimap` file.

With the kernel module information exported to these files `modules.usbmap` and `modules.pcimap` our version of `/sbin/hotplug` can look like the following example:

```
#!/bin/bash

declare -i usb_idVendor
declare -i usb_idProduct

MAP=/lib/modules/‘uname -r‘/modules.usbmap

usb_map_modules ()
{
  # convert the usb_device_id fields to
  # integers as we read them
  local line module
  declare -i match_flags
  declare -i idVendor idProduct

  # look at each usb_device_id entry
  # collect all matches in $DRIVERS
  while read line
  do
    case "$line" in
      \#*) continue ;;
    esac

    set $line

    module=$1
    match_flags=$2
    idVendor=$3
    idProduct=$4

    : checkmatch $module
    : idVendor $idVendor $usb_idVendor
    if [ 0x0001 -eq $(($match_flags & 0x0001)) ] &&
       [ $idVendor -ne $usb_idVendor ]; then
      continue
    fi

    : idProduct $idProduct $usb_idProduct
    if [ 0x0002 -eq $(($match_flags & 0x0002)) ] &&
       [ $idProduct -ne $usb_idProduct ]; then
      continue
    fi

    # It was a match!
    DRIVERS="$module $DRIVERS"
    : drivers $DRIVERS
  done
}

if [ "$1" = "usb" ]; then
  IFS=/
  set $PRODUCT ’’
  usb_idVendor=$1
  usb_idProduct=$2
  IFS="$DEFAULT_IFS"

  usb_map_modules < $MAP

  if [ "$ACTION" = "add" ]; then
    PROGRAM="/sbin/modprobe"
  else
    PROGRAM="/sbin/rmmod"
  fi

  for MODULE in $DRIVERS
  do
    $PROGRAM $MODULE
  done
fi
```

The Linux-Hotplug project has created a set of scripts that covers all of the different subsystems that can call `/sbin/hotplug` enabling drivers to be automatically loaded, and network subsystems started up and shut down. These scripts are released under the GPL and available at

`http://linux-hotplug.sourceforge.net/`

This package is currently being used in the RedHat and Debian releases.

6 Future

The current `/sbin/hotplug` subsystem needs to be incorporated into other kernel systems, as they develop hotplug capability. SCSI, PCMCIA, IDE, and other systems all have hotplug patches available for kernel support, but need to have script support, kernel macro support, and modutils `depmod` support added in order to provide the user with a consistent experience. Patches for kernel support of hotplug PCI and cPCI drivers need to take advantage of the current `/sbin/hotplug` interface and get integrated into the main kernel tree.

7 Acknowledgments

I would like to thank David Brownell who wrote the original `/sbin/hotplug` kernel patch, and most of the linux-hotplug scripts. Without his persistence, Linux would not have this user friendly feature. I would also like to acknowledge the entire Linux USB development team, who have provided a solid kernel subsystem in a relativly short ammount of time.

References

[1] http://lxr.linux.no/source/drivers/usb/bluetooth.c

[2] David Brownell,
Updated "Kernel USBD" patch,
http://marc.theaimsgroup.com/?l=linux-usb-devel&m=96334011602320

[3] http://lxr.linux.no/ident?i=call_usermodehelper

[4] http://lxr.linux.no/ident?i=hotplug_path

[5] http://lxr.linux.no/ident?i=MODULE_DEVICE_TABLE

[6] http://lxr.linux.no/ident?i=net_run_sbin_hotplug

[7] http://lxr.linux.no/ident?i=net_run_sbin_hotplug

[8] http://lxr.linux.no/source/drivers/usb/uhci.c

[9] http://lxr.linux.no/source/drivers/usb/serial/whiteheat.c