USB hotplugging

Linux Hotplugging

In hotpluggable busses like USB (and Cardbus PCI), end-users plug devices into the bus with power on. In most cases, users expect the devices to become immediately usable. That means the system must do many things, including:

- Find a driver that can handle the device. That may involve loading a kernel module; newer drivers can use module-init-tools to publish their device (and class) support to user utilities.
- Bind a driver to that device. Bus frameworks do that using a device driver's probe() routine.
- Tell other subsystems to configure the new device. Print queues may need to be enabled, networks brought up, disk partitions mounted, and so on. In some cases these will be driver-specific actions.

This involves a mix of kernel mode and user mode actions. Making devices be immediately usable means that any user mode actions can't wait for an administrator to do them: the kernel must trigger them, either passively (triggering some monitoring daemon to invoke a helper program) or actively (calling such a user mode helper program directly).

Those triggered actions must support a system's administrative policies; such programs are called "policy agents" here. Typically they involve shell scripts that dispatch to more familiar administration tools.

Because some of those actions rely on information about drivers (metadata) that is currently available only when the drivers are dynamically linked, you get the best hotplugging when you configure a highly modular system.

Kernel Hotplug Helper (`/sbin/hotplug`)

There is a kernel parameter: `/proc/sys/kernel/hotplug`, which normally holds the pathname `/sbin/hotplug`. That parameter names a program which the kernel may invoke at various times.

The /sbin/hotplug program can be invoked by any subsystem as part of its reaction to a configuration change, from a thread in that subsystem. Only one parameter is required: the name of a subsystem being notified of some kernel event. That name is used as the first key for further event dispatch; any other argument and environment parameters are specified by the subsystem making that invocation.

Hotplug software and other resources is available at:

http://linux-hotplug.sourceforge.net

Mailing list information is also available at that site.

USB Policy Agent

The USB subsystem currently invokes `/sbin/hotplug` when USB devices are added or removed from system. The invocation is done by the kernel hub workqueue [hub_wq], or else as part of root hub initialization (done by init, modprobe, kapmd, etc). Its single command line parameter is the string "usb", and it passes these environment variables:

ACTION	`add`, `remove`
PRODUCT	USB vendor, product, and version codes (hex)
TYPE	device class codes (decimal)
INTERFACE	interface 0 class codes (decimal)

If "usbdevfs" is configured, DEVICE and DEVFS are also passed. DEVICE is the pathname of the device, and is useful for devices with multiple and/or alternate interfaces that complicate driver selection. By design, USB hotplugging is independent of `usbdevfs`: you can do most essential parts of USB device setup without using that filesystem, and without running a user mode daemon to detect changes in system configuration.

Currently available policy agent implementations can load drivers for modules, and can invoke driver-specific setup scripts. The newest ones leverage USB module-init-tools support. Later agents might unload drivers.

USB Modutils Support

Current versions of module-init-tools will create a `modules.usbmap` file which contains the entries from each driver's `MODULE_DEVICE_TABLE`. Such files can be used by various user mode policy agents to make sure all the right driver modules get loaded, either at boot time or later.

See `linux/usb.h` for full information about such table entries; or look at existing drivers. Each table entry describes one or more criteria to be used when matching a driver to a device or class of devices. The specific criteria are identified by bits set in "match_flags", paired with field values. You can construct the criteria directly, or with macros such as these, and use driver_info to store more information:

```
USB_DEVICE (vendorId, productId)
    ... matching devices with specified vendor and product ids
USB_DEVICE_VER (vendorId, productId, lo, hi)
    ... like USB_DEVICE with lo <= productversion <= hi
USB_INTERFACE_INFO (class, subclass, protocol)
    ... matching specified interface class info
USB_DEVICE_INFO (class, subclass, protocol)
    ... matching specified device class info
```

A short example, for a driver that supports several specific USB devices and their quirks, might have a MODULE_DEVICE_TABLE like this:

```
static const struct usb_device_id mydriver_id_table[] = {
    { USB_DEVICE (0x9999, 0xaaaa), driver_info: QUIRK_X },
    { USB_DEVICE (0xbbbb, 0x8888), driver_info: QUIRK_Y|QUIRK_Z },
    ...
    { } /* end with an all-zeroes entry */
};
MODULE_DEVICE_TABLE(usb, mydriver_id_table);
```

Most USB device drivers should pass these tables to the USB subsystem as well as to the module management subsystem. Not all, though: some driver frameworks connect using interfaces layered over USB, and so they won't need such a struct usb_driver.

Drivers that connect directly to the USB subsystem should be declared something like this:

```
static struct usb_driver mydriver = {
    .name           = "mydriver",
    .id_table       = mydriver_id_table,
    .probe          = my_probe,
    .disconnect     = my_disconnect,

    /*
    if using the usb chardev framework:
        .minor              = MY_USB_MINOR_START,
        .fops               = my_file_ops,
    if exposing any operations through usbdevfs:
        .ioctl              = my_ioctl,
    */
};
```

When the USB subsystem knows about a driver's device ID table, it's used when choosing drivers to probe(). The thread doing new device processing checks drivers' device ID entries from the `MODULE_DEVICE_TABLE` against interface and device descriptors for the device. It will only call `probe()` if there is a match, and the third argument to `probe()` will be the entry that matched.

If you don't provide an `id_table` for your driver, then your driver may get probed for each new device; the third parameter to `probe()` will be `NULL`.