Dynamically Loadable Kernel Modules

Introduction

We often design systems to provide a common framework, but that expect problem-specific implementations to be provided at a later time. This approach is embraced by several standard design patterns (e.g. Strategy, Factory, Plug-In). These approaches have a few key elements in common:

- all implementations provide similar functionality in accordance with a common *interface*.
- the selection of a particular implementation can be deffered until run time.
- decoupling the overarching service from the plug-in implementations makes it possible for a system to work with a potentially open-ended set of of algorithms or adaptors.

In most programs the set of available implementations is locked in at build-time. But many systems have the ability to select and load new implementations at run time as *dynamically loadable modules*. We see this with browser plug-ins. Operating systems may also support many different types of dynamically loadable modules (e.g., file systems, network protocols, device drivers). Device drivers are a particularly important and rich class of dynamically loadable modules ... and therefore a good example to study.

There are several compelling reasons for wanting device drivers to be dynamically loadable:

- the number of possible I/O devices is far too large to build all of them in to the operating system.
- if we want an operating system to automatically work on any computer, it must have the ability to automatically identify and load the required device drivers.
- many devices (e.g., USB) are hot-pluggable, and we cannot know what drivers are required until the the associated device is plugged in to the computer.
- new devices become available long after the operating system has been shipped, and so must be *after market* addable.
- most device drivers are developed by the hardware manufacturers, and delivered to customers independently from the operating system.

Choosing Which Module to Load

In the abstract, a program needs an implementation and calls a *Factory* to obtain it. But how does the Factory know what implementation class to instantiate?

- for a browser plugin, there might be a MIME-type assocated with the data to be handled, and the browser can consult a registry to find the plug-in associated with that MIME-type. This is a very general mechanism ... but it presumes that data is tagged with a type, and that somebody is maintaining a tag-to-plugin registery.
- at the other extreme, the Factory could load all of the known plug-ins and call a *probe* method in each to see which (if any) of the plug-ins could identify the data and claim responsibility for handling it.

Long ago, dynamically loadable device drivers used the probing process, but this was both unreliable (might incorrectly accept the wrong device) and dangerious (touching random registers in random devices). Today most I/O busses support self-identifying devices. Each device has type, model, and even serial number information that can be queried in a standard way (e.g., by *walking the configuration space*). This information can be used, in combination with a device driver registry, to automatically select a driver for a given device. These registries may support precedence rules that can chose the best from among multiple competing drivers (e.g., a generic VGA driver, a GeForce driver, and a GeForce GTX 980 driver).

Loading a New Module

In many cases, the module to be loaded may be entirely self-contained (it makes no calls outside of itself) or uses only standard shared libraries (which are expected to be mapped in to the address space at well known locations). In these cases loading a new module is as simple as allocating some memory (or address space) and reading the new module into it.

In many cases (including device drivers and file systems) the dynamically loaded module may need to make use of other functions (e.g., memory allocation, synchronization, I/O) in the program into which it is being added. This means that the module to be loaded will (like an object module) have unresolved external references, and requires a *run-time loader* (a simplified linkage editor) that can look up and adjust all of those references as the new module is being loaded.

Note, however, that these references can only be from the dynamically loaded module into the program into which it is loaded (e.g., the operating system). The main program can never have any direct references into the dynamically loaded module ... because the dynamically loaded module may not always be there.

Initialization and Registration

If the operating system is not allowed to have any direct references into a dynamically loadable module, how can the new module be used? When the run-time loader is invoked to load a new dynamically loadable module, it is common for it to return a vector that contains a pointer to at least one method: an initialization function.

After the module has been loaded into memory, the main program calls its initialization method. For a dynamically loaded device driver, the initialization method might:

- allocate memory and initialize driver data structures.
- allocate I/O resources (e.g., bus addresses, interrupt levels, DMA channels) and assign them to the devices to be managed.
- register all of the device instances it supports. Part of this registration would involve providing a vector (of pointers to standard device drvier entry points) that client software could call in order to use these device instances.

Device instance configuration and initialization is another area where self-identifying devices have made it much easier to implement dynamically loaded device drivers:

- Long ago, devices were configured for particular bus addresses and interrupt levels with mechanical switches, that would be set before the card was plugged in to the bus. These resource assignments would be recorded in a system configuration table, which would be compiled (or read during system start-up) into the operating system, and used to select and configure corresponding device driver instances.
- More contemporary busses (like PCIe or USB) provide mechanisms to discover all of the available devices and learn what resources (e.g., bus addresses, DMA channels, interrupt levels) they require. The device driver can then allocate these resources from the associated bus driver, and assign the required resources to each device.

Using a Dynamically Loaded Module

The operating system will provide some means by which processes can open device instances. In Linux the OS exports a pseudo file system (/dev) that is full of *special files*, each one associated with a registered device instance. When a process attempts to open one of those special files, the operating system creates a reference from the open file instance to the registered device instance. From then on, when ever the process issues a *read(2)*, *write(2)*, or *ioctl(2)* system call on that file descriptor, the operating system forwards that call to the appropriate device driver entry point.

A similar approach is used when higher level frameworks (e.g., terminal sessions, network protocols or file systems) are implemented on top of a device. Each of those services maintains open references to the underlying devices, and when they need to talk to the device (e.g., to queue an I/O request or send a packet) the OS forwards that call to the appropriate device driver entry point.

The system often maintains a table of all registered device instances and the associated device driver entry points for each standard operation. When ever a request is to be made for any device, the operating system can simply index into this table by a device identifier to look up the address of the entry point to be called. Such mechanisms for registering heterogenous implementations and forwarding requests to the correct entry point are often referred to as *federation frameworks* because they combine a set of independent implementations into a single unified framework.

Unloading

When all open file descriptors into those devices have been closed and the driver is no-longer needed, the operating system can call an shut-down method that will cause the driver to:

- un-register itself as a device driver
- shut down the devices it had been managing
- return all allocated memory and I/O resources back to the operating system.

After which, the module can be safely unloaded and that memory freed as well.

The Criticality of Stable Interfaces

All of this is completely dependent on stable and well specified interfaces:

- the set of entry-points for any class of device driver must be well defined, and all drivers must compatably implement all of the relevant interfaces.
- the set of functions within the main program (OS) that the dynamically loaded modules are allowed to call must be well defined, and the interfaces to each of those functions must be stable.

If one device driver did not implement a standard entry point in the standard way, clients of that device would not work. Some functionality may be optional, and it may be acceptable for a device driver to refuse some requests. But this may make the application responsible for dealing with some version incompatabilities.

If an operating system does not implement some standard service function (e.g., memory allocation) in the standard way, a device driver written to that interface standard may not work when loaded into the non-compliant operating system.

There is often a tension between the conflicting needs to support new hardware and software features while retaining compatability with old device drivers.

Hot-Pluggable Devices and Drivers

One of the major advantages of dynamically loadable modules is that they can be loaded at any time; not merely during start-up. Hot-plug busses (e.g., USB) can generate events whenever a device is added to, or removed from the bus. In many systems a hot-plug manager:

- subscribes to hot-plug events.
- when a new device is inserted, walks the configuration space to identify the new device, finds, and loads the appropriate driver.

- when a device is removed, finds the associated driver and calls its *removal* method to inform it that the device is no longer on the bus.

Hot-plugable busses often have multiple power levels, and a newly inserted device may receive only enough power to enable it to be queried and configured. When the driver is ready to start using the device, it can instruct the bus to fully power the device. Some hot-pluggable busses also have mechanical safety interlocks to prevent a device from being removed while it is still in use. In these cases the driver must shut down and release the device before it can be removed.

Summary

This discussion has focused on dynamically loadable device drivers, but most of the issues (selecting the module to be loaded, dependencies of the loaded module on the host program, initializing and shutting down dynamically loaded modules, binding clients to dynamically loaded modules, and defining and managing the interfaces between the loaded and hosting modules) are applicable to a much wider range of dynamically loadable modules.

Stable and well standardized interfaces are critical to any such framework:

- the methods to be implemented by the dynamically loaded modules.
- the services that can be used by the dynamically loaded modules.