Linux Kernel Communication Methods:

Summary [What is this Document]:

In my limited exploits with Linux kernel development, I came across one program design issue that plagued me for quite a while and I tried several different solutions for it. That design issue was how to get data from the kernel to user space on linux and in some cases also from user space back to the kernel, or in other words single-direction communication or bi-directional communication to/from the linux kernel. For those who are just starting out with the kernel here's a description of the problem (feel free to skip it if you wish):

The Linux kernel is considered a closed and protected space to ensure that the kernel doesn't crash even in the most extreme cases. In other words, the kernel should always be separated from any outside influence. Those around us always influence our behavior, after all. To that end, the kernel handles hardware devices, time schedule, and a list of system services, but anything outside of its direct implementation is "user space." Code written and compiled to run in user space doesn't use kernel data structures, kernel devices, or kernel memory. To get info from the kernel, often user space programs must open character devices stored in /dev and /sys and read those in read-only mode, for instance. When two-way communication is required, it's generally for a kernel module loaded as an extension to general kernel behavior, and it is a rare design method and the lack of examples that show it can be daunting. The main kernel generally only accepts calls from user space via system calls of various types, and as a general rule, typically only modules use methods outside of system calls to communicate with user space.

A kernel module is a device driver, or driver for a system service, such as a new filesystem, that's loaded into the kernel as an expansion module. Modules are required to follow very closely defined setup criteria to compile using existing kernel code. A module can be loaded with the command insmod into he running linux kernel, and removed with rmmod. Inside the module's code, you generally will define the system calls necessary to begin and end the module and communicate with the kernel. These system calls are, for instance, sometimes interrupts generated by hardware devices the module acts as a driver for, or they can be system interfaces such as the linux vfs layer which acts as a go-between for filesystems to talk to the kernel. Using these methods, a module can act as part of the system without directly altering kernel code, and often a module may have layers of protection between itself and most of the running kernel code.

So, with everything geared toward protecting the kernel, it can sometimes be challenging to get data out and into the secure areas without delays, bottlenecks, bugs, and outright roadblocks. This document assumes that the reader may (or may not) be a beginner kernel developer and is looking for some method of communicating data to or from the Linux kernel. It goes over several methods of communication, expounds on their good and bad points, notes what situations are good or bad uses for that method, and where possible example code is given that can get a new developer up and running. One area this doc pays special attention to is the speed/performance of these communication methods and what data profiles they would best fit.

The Legend of communication Traits:

In my experience, it's easy to find documentation on kernel communication, but nearly impossible to know when each method is/is not a good fit. To that end, this document is arranged with methods of communicating small messages at the beginning and progresses to the better methods of communicating large amounts of data later in the document. Further, each method is marked with several icons which will make it obvious what each communication method is good at with a glance. This section provides the legend for these icons to allow the reader to skim the methods quickly.

Single-direction Communication:

Two-way Communication: 

Amount of Data the method is intended for:

Methods:

Perror and other kernel macros.

Netlink

I basically made a label here so I could tell you that under no circumstances should you ever consider using this one. See below for the explanation and the right way to do this.

Netlink Generic

When you search for netlink on the web, you should use the term netlink generic. Netlink is a general protocol for kernel communication that gives 30 standard IDs for system communication, and those 30 are allocated to kernel systems, so don't touch them. You can. But don't. You're going to eventually run into conflicts with other kernel modules, and that's a very bad idea. Luckily, several years after the original Netlink release the kernel developers realized this was a concept with limited scope and viability

and released an update that included Netlink Generic. Netlink Generic allows the protocol to be used with a service name rather than a simple integer from 1 to 30. So, unless you're a module developer that owns one of the 30 golden module ID's, Netlink Generic is the only netlink that exists. I mean, 30? Really? I guess around the year 2000 when it was first invented no one thought there'd ever be more than 30 kernel modules. However, these day every device has an embedded version of linux, so there are literally automated feeding bins for cats that have more than 30 modules running.

Good:

1. Netlink is fast.
2. It's standard and intended for messaging systems mainly. So, for logging especially it's probably the right choice, despite its annoying bugs.

Bad:

1. It's standard. It's inflexible and basically impossible to make extensible without some serious coding and trouble.

Shared Memory

The gotcha here and in the sections below is that there are many ways to used shared memory to pass information. When looking for information on the web about kernel communication, one of the issues you'll run into is distinguishing which type a particular example is using, which an author might be talking about, and trying to pull apart the spaghetti of information available on shared memory. To further complicate that topic, you need some type of lock on the memory to use it, and there are at least 3 main types that are typically used in Linux. To make this topic easier, we'll first talk about shared memory, and it's uses in kernel communication. Then later, we'll introduce locks and how to protect your shared memory access. This is especially helpful since your method of sharing data may determine which lock types you can use.

Character and Block Devices

Character and Block devices are communication pipes setup under the /dev directory under Linux. However, don't be fooled. These are not actually files and are not subject to the delay of using the VFS filesystem layer in Linux. Remember, in Unix anything can be a file. In this case, even a memory segment set aside for communication can be a file. These are the same thing as the shared memory spoken of above. However, there are many ways to access shared memory, and if it has a /dev file associated with it, then one of those ways is by simply opening it like a file from user space and reading/writing it, then closing it. This is meant to be a super-simple communication method that treats the memory like a file and eliminates the complexity for user-space developers. The catch here, from a module developer perspective, is that this method is a little mind-bending. All reads and writes of kernel data is initiated from the user side of the communication in almost all cases for any communication method. /dev

devices is no exception. So, if a kernel module wishes to communicate something to a user space program, then it must initiate an interrupt or set a poll to notify user space to open the /dev file. This is not difficult at all to do; however, it has an unfortunate side-effect. From a kernel perspective, writing data to user space requires a user program to open the file and read data, so for the module developer, a write to user space is a read. Conversely, a read from module space is a write. For the person developing the module, these terms are exactly opposite of what you mentally think they should be. I'm not sure about other kernel developers, but I remember this being particularly brain-deforming for me when I began

RelayFS (Now name changed to Relay Interface)

Relay is an interface that does one-way communication only, from the running kernel to user space. It is the module which sets up the /sys directory in Linux and under there are files maintained by various modules which can be opened only for reading. The files contain whatever information the module deems it wishes to communicate with user space. If you're looking at a one-way communication method and your data can be easily parse-able in a file, then this is probably the method you want.

File Read/Write