How to not invent kernel interfaces

Arnd Bergmann
arnd@arndb.de

July 31, 2007

Contents

1 Abstract **2**

2 Concepts **2**
2.1 Simplicity 2
2.2 Consistency 3
2.3 Fitness 3
2.4 ABI stability 4

3 Syscalls **5**
3.1 ioctl 6
3.2 Sockets 7
3.2.1 Netlink 7

4 ABI Emulation **9**
4.1 Other operating systems 9
4.2 compat_ioctl 9

5 File systems **11**
5.1 procfs 12
5.2 sysfs 12
5.3 debugfs 13

6 Conclusion **13**

1 Abstract

Every piece of kernel code needs some form of user space interface. Examples for this include system calls, ioctl based character devices, virtual file systems, netlink or procfs files.

Choosing the right one is often hard and people tend to get flamed for their choices no matter what they do. The truth is that there usually is not a single right solution for a new problem, just different degrees on how wrong it gets, and coming up with a simple interface tends to be the most complicated task in any new subsystem.

This paper explores different interfaces and their pros and cons by looking at examples from existing kernel code.

2 Concepts

In Finding the right interface for a new functionality should take into account different aspects. The most important ones are simplicity, consistency and fitness for the given problem. Together, they form what can best be described as 'good taste'. Taste is the ultimate feature of a good interface, but it doesn't have a clear definition, so we need to rely on more concrete indications for whether an interface is good enough.

The importance of getting an interface right in the beginning comes from the problem of changing it later. It is always possible to completely replace all of the kernel code implementing a given functionality, but you cannot change an interface in fundamental ways without breaking user applications.

2.1 Simplicity

The most important aspect for a kernel interface is simplicity. A complex interface is hard to implement correctly and hard to understand, which means application programmers will introduce bugs when trying to use it.

Interestingly, it is much harder to come up with a simple interface than it is to create a complex mess. Even if you start out with a simple interface, you tend to forget some aspect of the problem and later add complexity to it in order to cover all the corner cases. When that happens, it may be better to start over from scratch and come up with a better model of what needs to be done.

Since the interface needs to be implemented on both kernel and user side, both of these need to be as simple as possible. Making the user side as simple as possible is more important though, because there are typically many users of each kernel interface, so each one of them is prone to errors when it is not obvious how the interface is used.

Example 1 (nfsservctl) Sometimes an existing interface can be replaced with something simpler. One such example is the obscure nfsservctl system call that is used for a specific subsystem, namely the NFS server in the kernel. Instead of the multiplexing system call, there is now a special file system called 'nfsd', each file in it representing one of the subfunctions of the older system call.

Using the simple_fill_super() and simple_transaction helpers, a write() followed by a read() on a file in there can be used the same way the system call was before.

This would be a good example of how to do a simple interface, if we didn't have to implement the original system call on top of it to maintain the stable ABI. In the end, user applications still use the old syscall, and the complexity remains in both kernel and user space.

2.2 Consistency

When adding a new interface to an existing kernel, you should take into account how other similar subsystems achieve the same functionality. Most of the time, multiple possible solutions exist, and if they are equally simple, you should choose the one that most closely resembles existing interfaces. This gives users and reviewers the least surprises and thereby avoids bugs.

Example 2 (infiniband uverbs) The infiniband driver subsystem has an interface for direct interaction of user applications with the hardware through 'verbs', which would typically be implemented as ioctl functions.

For some reason, the author must have been scared of adding a large amount of ioctl numbers in his driver and decided to use an interface based on read/write operations on a character device.

While this is not a bad idea in principle, the end result is a violation of the consistency principle. The ib_uverbs_write() function basically acts as an ioctl() replacement, where data buffers are passed in using write(), but contain command codes and structured data, in some cases accessing user data outside of the write buffer and passing data back to the user, or waiting for events from inside of the write() function, depending on the command code.

Using a real ioctl() function instead of write() here would been consistent with other interfaces and avoid potential problems with debuggability and 32 bit emulation.

2.3 Fitness

Third, an interface must be suited for the purpose it's used for. Just because others use a particular interface does not mean that it is also a good idea for something new.

Example 3 (Wext ioctl over netlink) One particularly sad story is configuration of wireless network devices in Linux. The wireless extensions in Linux use an ioctl interface over a socket, which is a bad idea to start with, for various reasons, but is somewhat consistent with what is used for ethernet devices and others.

Some of the extensions are generic to all wireless network drivers, but other are only implemented by some of them, so a 'private' range of ioctl numbers was assigned to these. In order to reply to ioctls asynchronously, an additional netlink interface was introduced. When it became apparent that device private numbers are not a good idea because of drivers using the same calls with different numbers and vice versa, a registration facility for them was added using special ioctls. Still adding to the complexity, some drivers started running out of private ioctl numbers, so another indirection was added for 'sub-ioctls'.

With this being the most complex configuration interface in Linux (possibly aside from the history s390 'chandev'), people called out for a rewrite. With all the best intentions, a new netlink based interface was added that unified the existing event mechanism with the configuration interfaces. Unfortunately, the new netlink interface had all the same warts as the ioctl interface, and was later removed again after a series of protests.

2.4 ABI stability

Linux defines interface stability in terms of the interface between kernel and user space. We happily break the interface between kernel components all the time (see Documentation/stable-API-nonsense.txt for details), but that does not mean that we break user interfaces. The idea is to guarantee every user program that has ever run on a mainline kernel to keep running on every future version. In particular, the interface stability includes all device drivers, since they are distributed together with the kernel, but not together with the programs using these drivers.

There have been cases in the past where we broke an ABI, whether intentionally or not. In general, this results in an endless stream of bug reports from users that all fall into the same traps, and public humiliation for the responsible developers.

Example 4 (Module loader) The 2.6. version of Linux introduced a complex reimplementation of loadable modules. There was only one program using the system calls for module loading (modutils) and that was replaced by module-init-tools. While almost everyone agrees that the 2.6 method of loading modules is much preferrable to the old one, the transition caused an endless amount of pain for users and distributors alike.

Example 5 (udev) The udev tool is a replacement for the a set of functionality, including the devfs file system, the hotplug tools and module autoloading. During the development, a lot of care was taken to ensure compatiblity with the older mechanism, so users that were switching from older kernels were not forced to upgrade to udev. Unfortunately, not enough care was taken to ensure compatiblity between udev versions. Since udev made use of a lot of sysfs files, it broke when some of these files went away. As a result, distributions shipping with an older version of udev could not be upgraded to a newer kernel without upgrading the udev version first.

Example 6 (KVM) The KVM hypervisor was introduced into the kernel in late 2006. For the sake of being extremely useful, it was allowed into the kernel with an unstable ioctl ABI, which then got broken in every each of the following kernel versions, and even more often in the development tree. As long as the user base consisted of only a small group of developers, that went fine, since everyone capable of using the code was aware of this problem.

With 2.6.22, the interface was formally frozen and will have to be maintained in a compatible way, but it is still likely to require changes in the near future. If this happens, it is guaranteed to be an unpleasant surprise for a number of users.

3 Syscalls

System calls are the low-level interface that everything else builds upon. Every system call has its own uniquely assigned number on each architecture, which makes it relatively hard to add a new one. More often than not, a new syscall gets discussed for months before the exact arguments and semantic details are nailed down enough and people agree that its time to finally add it to the kernel.

Still, it has become much easier to add system calls to the kernel than it used to be, and for many purposes, a system call feels more natural as an interface than any of the alternatives.

Example 7 (Historic System Calls) The oldest publically available source code of Unix, Version 4 from 1973, contained 38 system calls (exit, fork, read, write, open, close, wait, creat, link, unlink, exec, chdir, time, mknod, chmod, chmod, break, stat, seek, mount, umount, setuid, getuid, stime, fstat, stty, gtty, nice, sleep, sync, kill, csw, dup, pipe, times, setgid, getgid, signal).

This was enough to make a lot of user space code run, and one could argue that not much more is needed even today, because all other syscalls can be implemented using read/write on special file systems. This approach is taken for example by the Plan 9 operating system, which defines 50 system calls in its latest version, and like historic Unix versions, it does not include an ioctl() call.

When Linux 0.1 was released, it already contained 66 System calls, and at time of writing this, the i386 architecture defines 323 system call numbers.

3.1 ioctl

ioctl is the generic multiplexor system call. Originally introduced to control tty emulation devices, its use has spread to practically every class of device driver over the years. The biggest advantage of ioctl is that it becomes extremely easy to add another device specific call to it without having to go through an extremely tiring review process as with system calls.

Unfortunately, making it so easy to add new calls this way is also the main disadvantage of ioctl, because that means that developers get it wrong all the time and then have to live with the problems arising from having to remain backwards compatible. In the best case, you can simply add another ioctl number that gets the interface right and leaving of using the old or the new number to user space. In the worst case, all existing users are so horribly broken that they stop working once the kernel is fixed and end users need to upgrade their destroy packages at the same time as upgrading their kernels.

The most common problems with ioctl handlers are:

- complex data structures get passed into the kernel, and they are not 32/64 bit interface compatible, see the secion on ABI emulation for this problem.
- The ioctl number range is not properly assigned and registered in Documentation/ioctl-number.txt. This can lead to identical numbers assigned to different subsystems, which complicates debugging and 32 bit emulation.
- The data structures are defined in private headers so that the 'strace' tool does not recognize them, even when it is built against the latest kernel headers.
- The _IO/_IOW/_IOR macros are used incorrectly, meaning that the size of the data is not encoded properly.
- Drivers add another level of indirection and use a single ioctl number for multiple purposes, adding extra complexity in debugging and emulation.
- ioctl is used in place of a different interface that would be more appropriate for a given purpose, like a new syscall or a sysfs attribute.

An ioctl is probably the right interface if you already have a character device node for each physically available device, and you have understood all the potential pitfalls. Some rules for making the best of ioctl are

- Use the smallest possible set of ioctl numbers.
- Use only simple data structures, preferable only one integer per ioctl, but no pointers or variable-length arguments
- use only _IO, _IOC and _IOW, but not _IOWR if possible
- understand the rules for 32 bit emulation, and make sure you don't need a compat_ioctl handler.
- don't use ioctl for what can be easily expressed with mmap(), poll(), read(), write() seek().

3.2 Sockets

The socket interface in Linux was inherited from BSD Unix. While it uses file descriptor for communication, it is not really file-like in the way it works, and an attempt to submit another interface like this as a patch in the linux kernel would probably not get very far.

One immediate showstopper is the fact that on most architectures, the 15 system calls used for sockets are multiplexed over a common 'socketcall' entry point. Also, the method of establishing a connection is more complicated than necessary and inconsistent with all other interfaces used in Linux or Unix.

The one big reason why we still have the BSD socket interface despite all these drawbacks is that almost every other operating system on the planet has it as well. Compatibility with open standards can sometimes, but not always, override good taste.

Also, in the early days of Linux, adding system calls was more about making existing applications work without changes, while nowadays most new system calls that get added are specific to Linux.

3.2.1 Netlink

Netlink is one of the network protocol families supported by Linux which stands out from the others by not being a network protocol at all, but instead an interface between user and kernel space. It has been advertised as 'a flexible, robust wire-format communications channel typically used for kernel to user communication although it can also be used for user to user and kernel to kernel communications' by some people.

It is one of the few interfaces that provides a way to send asynchronous messages to user space from the kernel, and was seen as the answer to the problems that come with ioctl handlers for some time.

Interestingly, netlink never found much friends aside from network configuration and recently the 'uevent' system, even though it does not have properties that make it more useful for network configuration than for other purposes.

The most significant problem with netlink is the complexity it adds to applications using it, most of which it inherits from the socket interface that netlink is built on. Other problems are the limited extensibility, the primitive permission handling, and the inability to deal with 32 bit emulation on 64 bit hosts if the interface was defined in an incompatible way.

Some of these problems are addressed by the newer 'genetlink' helper that builds on netlink, but in reality, most code is better off using something else.

Example 8 (Ancillary data) Ancillary data on Unix domain sockets is one of the least intuitive interfaces offered by Linux. Just to scare off readers who have never heard of it, here is the example from the cmsg(3) man page describing how to pass an open file descriptor through a socket from one task to another:

```
struct msghdr msg = {0};
struct cmsghdr *cmsg;
int myfds[NUM_FD]; /* Contains the file descriptors to pass. */
char buf[CMSG_SPACE(sizeof myfds)];  /* ancillary data buffer */
int *fdptr;

msg.msg_control = buf;
msg.msg_controllen = sizeof buf;
cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_level = SOL_SOCKET;
cmsg->cmsg_type = SCM_RIGHTS;
cmsg->cmsg_len = CMSG_LEN(sizeof(int) * NUM_FD);
/* Initialize the payload: */
fdptr = (int *)CMSG_DATA(cmsg);
memcpy(fdptr, myfds, NUM_FD * sizeof(int));
/* Sum of the length of all control messages in the buffer: */
msg.msg_controllen = cmsg->cmsg_len;
```

4 ABI Emulation

One aspect of binary interfaces that is getting increasingly important is compatibility between 32 and 64 bit user applications running on a 64 bit kernel. Data structures and syscall arguments often differ in subtle ways, and the kernel needs to ensure that the arguments passed by a 32 bit process get interpreted correctly by the kernel.

4.1 Other operating systems

Traditionally, there was also support for running binaries from other operating systems like Xenix or OSF/1 on Linux, but that has become less interesting now that far more applications exist for Linux than for any other Unix and most other OSs include code for running Linux binaries natively.

Fragments of the old emulation code in Linux can still be found in the SysV IPC code that contains hacks for old x86 Unix versions, even on architectures that never ran any Unix, and in some ioctl definitions that are defined to match the respective native Unix variant on the alpha and sparc architectures.

Example 9 (The s390 problem) On the s390 architecture, emulating pointers is particularly tricky, because an 'unsigned long' integer is 32 bit, while a pointer is only 31 bit in legacy user applications, while the kernel uses 64 bit for both.

For reasons that are too ugly to explain here, the most significant bit in a pointer is always ignored on 32 bit s390, so in practice it can become '1' any system call or ioctl argument. When interpreting the pointer as a 32 bit integer and converting it into a 64 bit pointer, this may accidentally point into the second 2GB of memory, which are by definition invalid in a 32 bit s390 process. Consequently, every conversion of a user pointer value into a kernel pointer that can get passed to copy_from_user needs to go through a conversion function that is defined as a simple typecast all all architectures except s390, where it becomes

```
static inline void __user *compat_ptr(compat_uptr_t uptr)
{
        return (void __user *)(unsigned long)(uptr & 0x7fffffffUL);
}
```

4.2 compat_ioctl

Emulation of ioctl calls is probably the trickiest part in emulating foreign operating systems and 32 bit native applications. Every ioctl number has its own calling conventions and many of them are incompatible between 32 and

64 bit user programs. Traditionally, they were all handled by the architecture code for sparc64, parisc, mips64, s390x, ppc64, ia64 and later x86_64, but that has been consolidated into a single implementation in fs/compat_ioctl.c.

For Linux 2.6, a method was introduced to have ioctl argument conversion handled by every device driver for itself. Code from the generic compat_ioctl implementation is slowly being moved into the device drivers as appropriate, and new drivers must handle the conversion themselves, or (better) not need conversion at all.

The easiest way to ensure that an ioctl argument does not need conversion is make the argument a single 32 or 64 bit integer argument. If a data structure is absolutely necessary, it should only contain members of fixed size, i.e. no pointers or 'long' integers, and not need padding.

For a device driver that only has compatible ioctl arguments, ensuring compatibility is as easy as assigning the same function pointer to both the 'unlocked_ioctl' and 'compat_ioctl' members in the file_operations structure.

For drivers where it's already too late for this, because they use incompatible ioctl definitions, the recommended practice is to have separate functions for 'unlocked_ioctl' and 'compat_ioctl', and make each of them access their respective data structures in user space, but to call the same internal handlers with kernel native data structures, that typically match the 64 bit user version.

Example 10 (The i386 Problem) The x86 family of architectures (called i386, ia64 and x86_64 in the kernel) have another unique problem that causes endless pain for the implementation of compat_ioctl functions and system call handlers.

Unlike all other architectures that support both 32 and 64 bit code, i386 uses 32 bit alignment for 64 bit data types like 'long long' or '__u64', while the two 64 bit architectures that can run i386 binaries natively use 64 bit alignment. Therefore, some data structures that do not require special treatment on the other 64 bit architecture actually do need to be converted on x86_64 and ia64, and many developers are surprised when they see that their data structures are actually not compatible as they had intended.

As an example, a data structure like

```
struct foo {
     char a;
     unsigned long long b;
     int c;
     int d;
}
```

is padded to 20 bytes on i386, but to 24 bytes on most other architectures, including x86_64, and therefore needs to be converted when a driver accesses it from user space. If you add an '__attribute__((aligned(8)))' to the structure, it will be 24 bytes on all supported architectures, but the location of the members 'b', 'c' and 'd' in the structure is still different on i386. Adding '__attribute__((packed))' would solve this problem, but make access unaligned and therefore slow on many processors.

In order to define data structures that don't have this problem, define data structures in a way to avoid padding, both inside and at the end. 'gcc -Wpadded' gives a good hint of potential problems.

When writing 32 bit emulation code for a data structure that already has this problem, the best solution is to use the compat_u64 and compat_s64 data types in the conversion handler, like

```
struct compat_foo {
        char a;
        compat_u64 b;
        int c;
        int d;
}
```

This describes the 32 bit data structure from the example above correctly on all architectures.

5 File systems

One idea made famous especially by the Plan 9 operating system is to take the Unix 'Everything is a File' concept much further than the classic Unix style operating systems. Where Unix basically had file systems for files stored on disk, and special files for character and block devices, about half of the eighty file system types in Linux 2.6.22 are for purposes other than storing files.

An important advantage of using a virtual file system for all interaction between kernel and user space is that you don't need to introduce new control structures in your user applications the way you would have to do for system calls or ioctls. This is especially useful in scripting languages when they interact with the kernel.

One important problem however does not go away by using a file system: Every file you introduce in a virtual file system and every string you can read and write in such a file becomes part of the Application Binary Interface, and must never be changed in incompatible ways once a user program has started to rely on it.

5.1 procfs

The /proc file system was the first abstract file system introduced into Linux. Like on some Unix variants, it was originally meant to represent user processes in a simple way. It later developed into a bit of a mess, when an endless amount of interfaces were added for device drivers and other subsystems. Each of these interfaces became part of the kernel ABI until the point where /proc became a maintenance nightmare and developers were basically banned from adding new files to it. Older books about kernel programming still talk a lot about using procfs files, but it is almost always a mistake for any new code.

However, many concepts from procfs have proved valuable and are now used in other file systems like sysfs and debugfs. If you feel the need to add a file to /proc, think about adding it to one of the others instead or to write your own file system for the purpose.

5.2 sysfs

Sysfs found its way into the kernel initially under the name of driverfs, with the intention of making a view of all drivers and devices that the kernel knows about visible to users, in a much cleaner way than what would have been possible with procfs.

Sysfs now shows a hierarchy of 'kobject' data structures, each of them as a directory, along with 'attribute' data structures that are files with typically a single value encoded in a text string. The most important objects here are the devices that are present in the system, and a driver that wishes to externalize some properties of its devices can easily add read-only or read-write attributes to these device objects.

For simple drivers, this can be a nicer alternative to an ioctl function, especially when you would like the functionality to be scriptable.

EXAMPLE 11 (SPUFS) The SPU file system is an example for a special-purpose file system. It was developed by your author for controlling the Synergistic Processing Units on the Cell Broadband Engine processor. The file system interface was chosen because of the flexibility compared to character devices or new system calls, which are the more conventional approaches for driving new hardware. Specific advantages include:

- A directory can be used to represent a context, so it's easy to see what contexts are present and to group their properties as files.
- Besides read/write based interfaces, other system calls like mmap() or poll() work naturally on files.

- Permissions for accessing user resources can be controlled with regular chmod calls.
- With the delete-on-close semantics in spufs, life time rules are straightforward.

In the end, using only a file system for SPUs turned out not to be ideal, so two new system calls were added (spu_run and spu_create) that interoperate with the file system. This violates the consistency rule, but it does make the user side simpler in the end.

5.3 debugfs

Debugfs is different from all other interfaces discussed here, in that it is explicitly meant for non-stable debugging interfaces. It is extremely flexible and easy to use, so it is an ideal candidate for prototyping and ad-hoc debugging, but not for shipping code that is actually meant to be used by end users.

Each file in debugfs can have its own file_operations, so you can do everything with it that can be done with a character device, and there are helpers that allow you to monitor variables without the need to write your own file operations if you don't want to.

For developers that want to have their own file system in the end, debugfs is a good starting point, as it is relatively simple to turn the code for a debugfs directory into a new file system once you are happy with the file interfaces.

6 Conclusion

With all the different interfaces shown here, and more that are not covered, it's clear that there is no easy answer for how to do it right. The best approach is often to avoid adding new interfaces altogether and use what is already there. Where that is not possible, you can still put thought into reusing existing functionality.

Special file systems are currently the most fashionable approach for anything that deals with highly complex subsystems, because they avoid some of the problems of defining binary interfaces, but each of the other alternatives has its own niche.