Object-oriented design patterns in the kernel, part 1

Despite the fact that the Linux Kernel is mostly written in C, it makes broad use of some techniques from the field of object-oriented programming. Developers wanting to use these object-oriented techniques receive little support or guidance from the language and so are left to fend for themselves. As is often the case, this is a double-edged sword. The developer has enough flexibility to do really cool things, and equally the flexibility to do really stupid things, and it isn't always clear at first glance which is which, or more accurately: where on the spectrum a particular approach sits.

Instead of looking to the language to provide guidance, a software engineer must look to established practice to find out what works well and what is best avoided. Interpreting established practice is not always as easy as one might like and the effort, once made, is worth preserving. To preserve that effort on your author's part, this article brings another installment in an occasional series on Linux Kernel Design Patterns and attempts to set out - with examples - the design patterns in the Linux Kernel which effect an object-oriented style of programming.

Rather than providing a brief introduction to the object-oriented style, tempting though that is, we will assume the reader has a basic knowledge of objects, classes, methods, inheritance, and similar terms. For those as yet unfamiliar with these, there are plenty of resources to be found elsewhere on the web.

Over two weeks we will look for patterns in just two areas: method dispatch and data inheritance. Despite their apparent simplicity they lead to some rich veins for investigation. This first article will focus on method dispatch.

Method Dispatch

The large variety of styles of inheritance and rules for its usage in languages today seems to suggest that there is no uniform understanding of what "object-oriented" really means. The term is a bit like "love": everyone thinks they know what it means but when you get down to details people can find they have very different ideas. While what it means to be "oriented" might not be clear, what we mean by an "object" does seem to be uniformly agreed upon. It is simply an abstraction comprising both state and behavior. An object is like a record (Pascal) or struct (C), except that some of the names of members refer to functions which act on the other fields in the object. These function members are sometimes referred to a "methods".

The most obvious way to implement objects in C is to declare a "struct" where some fields are pointers to functions which take a pointer to the struct itself as their first argument. The calling convention for method "foo" in object "bar" would simply be: bar-

>foo(bar, ...args); While this pattern is used in the Linux kernel it is not the dominant pattern so we will leave discussion of it until a little later.

As methods (unlike state) are not normally changed on a per-object basis, a more common and only slightly less obvious approach is to collect all the methods for a particular class of objects into a separate structure, sometimes known as a "virtual function table" or vtable. The object then has a single pointer to this table rather than a separate pointer for each method, and consequently uses less memory.

This then leads to our first pattern - a **pure vtable** being a structure which contains only function pointers where the first argument of each is a pointer to some other structure (the object type) which itself contains a pointer to this vtable. Some simple examples of this in the Linux kernel are the file_lock_operations structure which contains two function pointers each of which take a pointer to a struct file_lock, and the seq_operations vtable which contains four function pointers which each operate on a struct seq_file. These two examples display an obvious naming pattern - the structure holding a vtable is named for the structure holding the object (possibly abbreviated) followed by "_operations". While this pattern is common it is by no means universal. Around the time of 2.6.39 there are approximately 30 "*_operations" structures along with well over 100 "*_ops" structures, most if not all of which are vtables of some sort. There are also several structs such as struct mdk_personality which are essentially vtables but do not have particularly helpful names.

Among these nearly 200 vtable structures there is plenty of variability and so plenty of scope to look for interesting patterns. In particular we can look for common variations from the "pure vtable" pattern described above and determine how these variations contribute to our understanding of object use in Linux.

NULL function pointers

The first observation is that some function pointers in some vtables are allowed to be NULL. Clearly trying to call such a function would be futile, so the code that calls into these methods generally contains an explicit test for the pointer being NULL. There are a few different reasons for these NULL pointers. Probably easiest to justify is the incremental development reason. Because of the way vtable structures are initialized, adding a new function pointer to the structure definition causes all existing table declarations to initialise that pointer to NULL. Thus it is possible to add a caller of the new method before any instance supports that method, and have it check for NULL and perform a default behavior. Then as incremental development continues those vtable instances which need it can get non-default methods.

A recent example is commit 77af1b2641faf4 adding set_voltage_time_sel() to struct regulator_ops which acts on struct regulator_dev. Subsequent commit 42ab616afe8844 defines that method for a particular device. This is simply the most recent example of a very common theme.

Another common reason is that certain methods are not particularly meaningful in certain cases so the calling code simply tests for NULL and returns an appropriate error when found. There are multiple examples of this in the virtual filesystem (VFS) layer. For instance, the create() function in inode_operations is only meaningful if the inode in question is a directory. So inode_operations structures for non-directories typically have NULL for the create() function (and many others) and the calling code in vfs_create() checks for NULL and returns -EACCES.

A final reason that vtables sometimes contain NULL is that an element of functionality might be being transitioned from one interface to another. A good example of this is the ioctl() operation in file_operations. In 2.6.11, a new method, unlocked_ioctl() was added which was called without the big kernel lock held. In 2.6.36, when all drivers and filesystems had been converted to use unlocked_ioctl(), the original ioctl() was finally removed. During this transition a file system would typically define only one of two, leaving the other defaulting to NULL.

A slightly more subtle example of this is read() and aio_read(), also in file_operations, and the corresponding write() and aio_write(). aio_read() was introduced to support asynchronous IO, and if it is provided the regular synchronous read() is not needed (it is effected using do_sync_read() which calls the aio_read() method). In this case there appears to be no intention of ever removing read() - it will remain for cases where async IO is not relevant such as special filesystems like procfs and sysfs. So it is still the case that only one of each pair need be defined by a filesystem, but it is not simply a transition, it is a long-term state.

Though there seem to be several different reasons for a NULL function pointer, almost every case is an example of one simple pattern - that of providing a default implementation for the method. In the "incremental development" examples and the non-meaningful method case, this is fairly straightforward. e.g. the default for inode->create() is simply to return an error. In the interface transition case it is only slightly less obvious. The default for unlocked_ioctl() would be to take the kernel lock and then call the ioctl() method. The default for read() is exactly do_sync_read() and some filesystems such as ext3 actually provide this value explicitly rather than using "NULL" to indicate a default.

With that in mind, a little reflection suggests that if the real goal is to provide a default, then maybe the best approach would be to explicitly give a default rather than using the circuitous route of using a default of NULL and interpreting it specially.

While NULL is certainly the easiest value to provide as a default - as the C standard assures us that uninitialized members of a structure do get set to NULL - it is not very much harder to set a more meaningful default. I am indebted to LWN reader wahern for the observation that C99 allows fields in a structure to be initialized multiple times with only the final value taking effect and that this allows easy setting of default values such as by following the simple model:

```
    #define FOO_DEFAULTS  .bar = default_bar, .baz = default_baz
    struct foo_operations my_foo = { FOO_DEFAULTS,
            .bar = my_bar,
    };
```

This will declare my_foo with a predefined default value for baz and a localized value for bar. Thus for the small cost of defining a few "default" functions and including a "_DEFAULTS" entry to each declaration, the default value for any field can easily be chosen when the field is first created, and automatically included in every use of the structure.

Not only are meaningful defaults easy to implement, they can lead to a more efficient implementation. In those cases where the function pointer actually is NULL it is probably faster to test and branch rather than to make an indirect function call. However the NULL case is very often the exception rather than the rule, and optimizing for an exception is not normal practice. In the more common case when the function pointer is not NULL, the test for NULL is simply a waste of code space and a waste of execution time. If we disallow NULLs we can make all call sites a little bit smaller and simpler.

In general, any testing performed by the caller before calling a method can be seen as an instance of the "mid-layer mistake" discussed in a previous article. It shows that the mid-layer is making assumptions about the behavior of the lower level driver rather than simply giving the driver freedom to behave in whatever way is most suitable. This may not always be an expensive mistake, but it is still best avoided where possible. Nevertheless there is a clear pattern in the Linux kernel that pointers in vtables can sometimes be NULLable, typically though not always to enable a transition, and the call sites should in these cases test for NULL before proceeding with the call.

The observant reader will have noticed a hole in the above logic denouncing the use NULL pointers for defaults. In the case where the default is the common case and where performance is paramount, the reasoning does not hold and a NULL pointer could well be justified. Naturally the Linux kernel provides an example of such a case for our examination.

One of the data structures used by the VFS for caching filesystem information is the "dentry". A "dentry" represents a name in the filesystem, and so each "dentry" has a parent, being the directory containing it, and an "inode" representing the named file. The dentry is separate from the inode because a single file can have multiple names (so an "inode" can have multiple "dentry"s). There is a dentry_operations vtable with a number of operations including, for example, "d_compare" which will compare two names and "d_hash" which will generate a hash for the name to guide the storage of the "dentry" in a hash table. Most filesystems do not need this flexibility. They treat names as uninterpreted strings of bytes so the default compare and hash functions are the common case. A few filesystems define these to handle case-insensitive names but that is not the norm.

Further, filename lookup is a common operation in Linux and so optimizing it is a priority. Thus these two operations appear to be good candidates where a test for NULL and an inlined default operation might be appropriate. What we find though is that when such an optimization is warranted it is not by itself enough. The code that calls d_compare() and d_hash() (and a couple of other dentry operations) does not test these functions for NULL directly. Rather they require that a few flag bits (DCACHE_OP_HASH, DCACHE_OP_COMPARE) in the "dentry" are set up to indicate whether the common default should be used, or whether the function should be called. As the flag field is likely to be in cache anyway, and the dentry_operations structure will often be not needed at all, this avoids a memory fetch in a hot path.

So we find that the one case where using a NULL function pointer to indicate a default could be justified, it is not actually used; instead, a different, more efficient, mechanism is used to indicate that the default method is requested.

Members other than function pointers

While most vtable-like structures in the kernel contain exclusively function pointers, there are a significant minority that have non-function-pointer fields. Many of these appear on the surface quite arbitrary and a few closer inspections suggest that some of them result of poor design or bit-rot and their removal would only improve the code.

There is one exception to the "functions only" pattern that occurs repeatedly and provides real value, and so is worth exploring. This pattern is seen in its most general form in struct mdk_personality which provides operations for a particular software RAID level. In particular this structure contains an "owner", a "name", and a "list". The "owner" is the module that provides the implementation. The "name" is a simple identifier: some vtables have string names, some have numeric names, and it is often called something different like "version", "family", "drvname", or "level". But conceptually it is still a name. In the present example there are two names, a string and a numeric "level".

The "list", while part of the same functionality, is less common. The mdk_personality structure has a struct list_head, as does struct ts_ops. struct file_system_type has a simple pointer to the next struct file_system_type. The underlying idea here is that for any particular implementation of an interface (or "final" definition of a class) to be usable, it must be registered in some way so that it can be found. Further, once it has been found it must be possible to ensure that the module holding the implementation is not removed while it is in use.

There seem to be nearly as many styles of registration against an interface in Linux as there are interfaces to register against, so finding strong patterns there would be a difficult task. However it is fairly common for a "vtable" to be treated as the primary handle on a particular implementation of an interface and to have an "owner" pointer which can be used to get a reference on the module which provides the implementation.

So the pattern we find here is that a structure of function pointers used as a "vtable" for object method dispatch should normally contain **only** function pointers. Exceptions require clear justification. A common exception allows a module pointer and possible other fields such as a name and a list pointer. These fields are used to support the registration protocol for the particular interface. When there is no list pointer it is very likely that the entire vtable will be treated as read-only. In this case the vtable will often be declared as a const structure and so could even be stored in read-only memory.

Combining Methods for different objects

A final common deviation from the "pure vtable" pattern that we see in the Linux kernel occurs when the first argument to the function is not always the same object type. In a pure vtable which is referenced by a pointer in a particular data structure, the first argument of each function is exactly that data structure. What reason could there be for deviating from that pattern? It turns out that there are few, some more interesting than others.

The simplest and least interesting explanation is that, for no apparent reason, the target data structure is listed elsewhere in the argument list. For example all functions in struct fb_ops take a struct fb_info. While in 18 cases that structure is the first argument, in five cases it is the last. There is nothing obviously wrong with this choice and it is unlikely to confuse developers. It is only a problem for data miners like your author who need to filter it out as an irrelevant pattern.

A slight deviation on this pattern is seen in struct rfkill_ops where two functions take a struct rkfill but the third - set_block() - takes a void *data. Further investigation shows that this opaque data is exactly that which is stored in rfkill->data, so set_block() could easily be defined to take a struct rfkill and simply to follow the ->data link itself. This deviation is sufficiently non-obvious that it could conceivably confuse developers as well as data miners and so should be avoided.

The next deviation in seen for example in platform_suspend_ops, oprofile_operations, security_operations and a few others. These take an odd assortment of arguments with no obvious pattern. However these are really very different sorts of vtable structures in that the object they belong to are singletons. There is only one active platform, only one profiler, only one security policy. Thus the "object" on which these operations act is part of the global state and so does not need to be included in the arguments of any functions.

Having filtered these two patterns out as not being very interesting we are left with two that do serve to tell us something about object use in the kernel.

quota_format_ops and export_operations are two different operations structures that operate on a variety of different data structures. In each case the apparent primary object (e.g. a struct super_block or a struct dentry) already has a vtable structure dedicated to it (such as super_operations or dentry_operations) and these new structures add new operations. In each case the new operations form a cohesive unit providing a related set of functionality - whether supporting disk quotas or NFS export. They don't all act on the same object simply because the functionality in question depends on a variety of objects.

The best term from the language of object-oriented programming for this is probably the "mixin". Though the fit may not be perfect - depending on what your exact understanding of mixin is - the idea of bringing in a collection of functionality without using strict hierarchical inheritance is very close to the purpose of quota_format_ops and export_operations.

Once we know to be on the lookout for mixins like these we can find quite a few more examples. The pattern to be alert for is not the one that led us here - an operations structure that operates on a variety of different objects - but rather the one we found where the functions in an "operations" structure operate on objects that already have their own "operations" structure. When an object has a large number of operations that are relevant and these operations naturally group into subsets, it makes a lot of sense to divide them into separate vtable-like structures. There are several examples of this in the networking code where for instance both tcp_congestion_ops and inet_connection_sock_af_ops operate (primarily) on a struct sock, which itself has already got a small set of dedicated operations.

So the pattern of a "mixin" - at least as defined as a set of operations which apply to one or more objects without being the primary operations for those objects - is a pattern that is often found in the kernel and appears to be quite valuable in allowing better modularization of code.

The last pattern which explains non-uniform function targets is probably the most interesting, particularly in its contrast to the obvious application of object-oriented programming style. Examples of this pattern abound with ata_port_operations, tty_operations, nfs_rpc_ops and atmdev_ops all appearing as useful examples. However we will focus primarily on some examples from the filesystem layer, particularly super_operations and inode_operations.

There is a strong hierarchy of objects in the implementation of a filesystem where the filesystem - represented by a "super_block" - has a number of files (struct inode) which may have a number of names or links (struct dentry). Further each file might store data in the page cache (struct address_space) which comprises a number of individual pages (struct page). There is a sense in which all of these different objects belong to the filesystem as a whole. If a page needs to be loaded with data from a file, the filesystem knows how to do that, and it is probably the same mechanism for every page in every file. Where it isn't always the same, the filesystem knows that too. So we could conceivably store every operation on every one of these objects in the struct super_block, as it represents the filesystem and could know what to do in each case.

In practice that extreme is not really helpful. It is quite likely that while there are similarities between the storage of a regular file and a directory, there are also important differences and being able to encode those differences in separate vtables can be helpful. Sometimes small symbolic links are stored directly in the inode while larger links are stored like the contents of a regular file. Having different readlink() operations for the two cases can make the code a lot more readable.

While the extreme of every operation attached to the one central structure is not ideal, it is equally true that the opposite extreme is not ideal either. The struct page in Linux does not have a vtable pointer at all - in part because we want to keep the structure as small as possible because it is so populous. Rather the address_space_operations structure contains the operations that act on a page. Similarly the super_operations structure contains some operations that apply to inodes, and inode_operations contains some operations that apply to dentries.

It is clearly possible to have operations structures attached to a parent of the target object - providing the target holds a reference to the parent, which it normally does - though it is not quite so clear that it is always beneficial. In the case of struct page which avoids having a vtable pointer altogether the benefit is clear. In the case of struct inode which has its own vtable pointer, the benefit of having some operations (such as destroy_inode() or write_inode()) attached to the super_block is less clear.

As there are several vtable structures where any given function pointer could be stored, the actual choice is in many cases little more than historical accident. Certainly the proliferation of struct dentry operations in inode_operations seems to be largely due to the fact that some of them used to act directly on the inode, but changes in the VFS eventually required this to change. For example in 2.1.78-pre1, each of link(), readlink(), followlink() (and some others which are now defunct) were changed from taking a struct inode to take a struct dentry instead. This set the scene for "dentry" operations to be in inode_operations, so when setattr and getattr were added for 2.3.48, it probably seemed completely natural to include them in inode_operations despite the fact that they acted primarily on a dentry.

Possibly we could simplify things by getting rid of dentry_operations altogether. Some operations that act on dentries are already in inode_operations and super_operations - why not move them all there? While dentries are not as populous as struct page there are still a lot of them and removing the "d_op" field could save 5% of the memory used by that structure (on x86-64).

With two exceptions, every active filesystem only has a single dentry operations structure in effect. Some filesystem implementations like "vfat" define two - e.g. one with case-sensitive matching and one with case-insensitive matching - but there is only one active per super-block. So it would seem that the operations in dentry_operations could be moved to super_operations, or at least accessed through "s_d_op". The two exceptions are ceph and procfs. These filesystems use different d_revalidate() operations in different parts of the filesystem and - in the case of procfs - different d_release() operations. The necessary distinctions could easily be made in per-superblock versions of these operations. Do these cases justify the 5% space cost? Arguably not.

Directly embedded function pointers

Finally it is appropriate to reflect on the alternate pattern mentioned at the start, where function pointers are stored directly in the object rather than in a separate vtable structure. This pattern can be seen in struct request_queue which has nine function pointers, struct efi which has ten function pointers, and struct sock which has six function pointers.

The cost of embedded pointers is obviously space. When vtables are used, there is only one copy of the vtable and multiple copies of an object (in most cases) so if more than one function pointer is needed, a vtable would save space. The cost of a vtable is an extra memory reference, though cache might reduce much of this cost in some cases. A vtable also has a cost of flexibility. When each object needs exactly the same set of operations a vtable is good, but if there is a need to individually tailor some of the operations for each object, then embedded function pointer can provide that flexibility. This is illustrated quite nicely by the comment with "zoom_video" in struct pcmcia_socket

```
/* Zoom video behaviour is so chip specific its not worth adding
   this to _ops */
```

So where objects are not very populous, where the list of function pointers is small, and where multiple mixins are needed, embedded function pointers are used instead of a separate vtable.

Method Dispatch Summary

If we combine all the pattern elements that we have found in Linux we find that:

Method pointers that operate on a particular type of object are normally collected in a vtable associated directly with that object, though they can also appear:

- In a mixin vtable that collects related functionality which may be selectable independently of the base type of the object.
- In the vtable for a "parent" object when doing so avoids the need for a vtable pointer in a populous object
- Directly in the object when there are few method pointers, or they need to be individually tailored to the particular object.

These vtables rarely contain anything other than function pointers, though fields needed to register the object class can be appropriate. Allowing these function pointers to be NULL is a common but not necessarily ideal technique for handling defaults.

So in exploring the Linux Kernel code we have found that even though it is not written in an object-oriented language, it certainly contains objects, classes (represented as vtables), and even mixins. It also contains concepts not normally found in object-oriented languages such as delegating object methods to a "parent" object.

Hopefully understanding these different patterns and the reasons for choosing between them can lead to more uniform application of the patterns across the kernel, and hence make it easier for a newcomer to understand which pattern is being followed. In the second part of our examination of object oriented patterns we will explore the various ways that data inheritance is achieved in the Linux kernel and discuss the strengths and weaknesses of each approach so as to see where each is most appropriate.

Object-oriented design patterns in the kernel, part 2

In the first part of this analysis we looked at how the polymorphic side of object-oriented programming was implemented in the Linux kernel using regular C constructs. In particular we examined method dispatch, looked at the different forms that vtables could take, and the circumstances where separate vtables were eschewed in preference for storing function pointers directly in objects. In this conclusion we will explore a second important aspect of object-oriented programming - inheritance, and in particular data inheritance.

Data inheritance

Inheritance is a core concept of object-oriented programming, though it comes in many forms, whether prototype inheritance, mixin inheritance, subtype inheritance, interface inheritance etc., some of which overlap. The form that is of interest when exploring the Linux kernel is most like subtype inheritance, where a concrete or "final" type inherits some data fields from a "virtual" parent type. We will call this "data inheritance" to emphasize the fact that it is the data rather than the behavior that is being inherited.

Put another way, a number of different implementations of a particular interface share, and separately extend, a common data structure. They can be said to inherit from that data structure. There are three different approaches to this sharing and extending that can be found in the Linux kernel, and all can be seen by exploring the struct inode structure and its history, though they are widely used elsewhere.

Extension through unions

The first approach, which is probably the most obvious but also the least flexible, is to declare a union as one element of the common structure and, for each implementation, to declare an entry in that union with extra fields that the particular implementation needs. This approach was introduced to struct inode in Linux-0.97.2 (August 1992) when

```
union {
        struct minix_inode_info minix_i;
        struct ext_inode_info ext_i;
        struct msdos_inode_info msdos_i;
} u;
```

was added to struct inode. Each of these structures remained empty until 0.97.5 when i_data was moved from struct inode to struct ext_inode_info. Over the years several more "inode_info" fields were added for different filesystems, peaking at 28 different "inode_info" structures in 2.4.14.2 when ext3 was added.

This approach to data inheritance is simple and straightforward, but is also somewhat clumsy. There are two obvious problems. Firstly, every new filesystem implementation needs to add an extra field to the union "u". With 3 fields this may not seem like a problem, with 28 it was well past "ugly". Requiring every filesystem to update this one structure is a barrier to adding filesystems that is unnecessary. Secondly, every inode allocated will be the same size and will be large enough to store the data for any filesystem. So a filesystem that wants lots of space in its "inode_info" structure will impose that space cost on every other filesystem.

The first of these issues is not an impenetrable barrier as we will see shortly. The second is a real problem and the general ugliness of the design encouraged change. Early in the 2.5 development series this change began; it was completed by 2.5.7 when there were no "inode_info" structures left in union u (though the union itself remained until 2.6.19).

Embedded structures

The change that happened to inodes in early 2.5 was effectively an inversion. The change which removed ext3_i from struct inode.u also added a struct inode, called vfs_inode, to struct ext3_inode_info. So instead of the private structure being embedded in the common data structure, the common data structure is now embedded in the private one. This neatly avoids the two problems with unions; now each filesystem needs to only allocate memory to store its own structure without any need to know anything about what other filesystems might need. Of course nothing ever comes for free and this change brought with it other issues that needed to be solved, but the solutions were not costly.

The first difficulty is the fact that when the common filesystem code - the VFS layer - calls into a specific filesystem it passes a pointer to the common data structure, the struct inode. Using this pointer, the filesystem needs to find a pointer to its own private data structure. An obvious approach is to always place the struct inode at the top of the private inode structure and simply cast a pointer to one into a pointer to the other. While this can work, it lacks any semblance of type safety and makes it harder to arrange fields in the inode to get optimal performance - as some kernel developers are wont to do.

The solution was to use the list_entry() macro to perform the necessary pointer arithmetic, subtracting from the address of the struct inode its offset in the private data structure and then casting this appropriately. The macro for this was called list_entry() simply because the "list.h lists" implementation was the first to use this pattern of data structure embedding. The list_entry() macro did exactly what was needed and so it was used despite the strange name. This practice lasted until 2.5.28 when a new container_of() macro was added which implemented the same functionality as list_entry(), though with slightly more type safety and a more meaningful name. With container_of() it is a simple matter to map from an embedded data structure to the structure in which it is embedded.

The second difficulty was that the filesystem had to be responsible for allocating the inode - it could no longer be allocated by common code as the common code did not have enough information to allocate the correct amount of space. This simply involved adding alloc_inode() and destroy_inode() methods to the super_operations structure and calling them as appropriate.

Void pointers

As noted earlier, the union pattern was not an impenetrable barrier to adding new filesystems independently. This is because the union u had one more field that was not an "inode_info" structure. A generic pointer field called generic_ip was added in Linux-1.0.5, but it was not used until 1.3.7. Any file system that does not own a structure in struct inode itself could define and allocate a separate structure and link it to the inode through u.generic_ip. This approach addressed both of the problems with unions as no changes are needed to shared declarations and each filesystem only uses the space that it needs. However it again introduced new problems of its own.

Using generic_ip, each filesystem required two allocations for each inode instead of one and this could lead to more wastage depending on how the structure size was rounded up for allocation; it also required writing more error-handling code. Also there was memory used for the generic_ip pointer and often for a back pointer from the private structure to the common struct inode. Both of these are wasted space compared with the union approach or the embedding approach.

Worse than this though, an extra memory dereference was needed to access the private structure from the common structure; such dereferences are best avoided. Filesystem code will often need to access both the common and the private structures. This either requires lots of extra memory dereferences, or it requires holding the address of the private structure in a register which increases register pressure. It was largely these concerns that stopped struct inode from ever migrating to broad use of the generic_ip pointer. It was certainly used, but not by the major, high-performance filesystems.

Though this pattern has problems it is still in wide use. struct super_block has an s_fs_info pointer which serves the same purpose as u.generic_ip (which has since been renamed to i_private when the u union was finally removed - why it was not completely removed is left as an exercise for the reader). This is the only way to store filesystem-private data in a super_block. A simple search in the Linux include files shows quite a collection of fields which are void pointers named "private" or something similar. Many of these are examples of the pattern of extending a data type by using a pointer to a private extension, and most of these could be converted to using the embedded-structure pattern.

Beyond inodes

While inodes serve as an effective vehicle to introduce these three patterns they do not display the full scope of any of them so it is useful to look further afield and see what else we can learn.

A survey of the use of unions elsewhere in the kernel shows that they are widely used though in very different circumstances than in struct inode. The particular aspect of inodes that is missing elsewhere is that a wide range of different modules (different filesystems) each wanted to extend an inode in different ways. In most places where unions are used there are a small fixed number of subtypes of the base type and there is little expectation of more being added. A simple example of this is struct nfs_fattr which stores file attribute information decoded out of an NFS reply. The details of these attributes are slightly different for NFSv2 and NFSv3 so there are effectively two subtypes of this structure with the difference encoded in a union. As NFSv4 uses the same information as NFSv3 this is very unlikely to ever be extended further.

A very common pattern in other uses of unions in Linux is for encoding messages that are passed around, typically between the kernel and user-space. struct siginfo is used to convey extra information with a signal delivery. Each signal type has a different type of ancillary information, so struct siginfo has a union to encode six different subtypes. union inputArgs appears to be the largest current union with 22 different subtypes. It is used by the "coda" network file system to pass requests between the kernel module and a user-space daemon which handles the network communication.

It is not clear whether these examples should be considered as the same pattern as the original struct inode. Do they really represent different subtypes of a base type, or is it just one type with internal variants? The Eiffel object-oriented programming language does not support variant types at all except through subtype inheritance so there is clearly a school of thought that would want to treat all usages of union as a form of subtyping. Many other languages, such as C++, provide both inheritance and unions allowing the programmer to make a choice. So the answer is not clear.

For our purposes it doesn't really matter what we call it as long as we know where to use each pattern. The examples in the kernel fairly clearly show that when all of the variants are understood by a single module, then a union is a very appropriate mechanism for variants structures, whether you want to refer to them as using data inheritance or not. When different subtypes are managed by different modules, or at least widely separate pieces of code, then one of the other mechanisms is preferred. The use of unions for this case has almost completely disappeared with only struct cycx_device remaining as an example of a deprecated pattern.

Problems with void pointers

Void pointers are not quite so easy to classify. It would probably be fair to say that void pointers are the modern equivalent of "goto" statements. They can be very useful but they can also lead to very convoluted designs. A particular problem is that when you look at a void pointer, like looking at a goto, you don't really know what it is pointing at. A void pointer called private is even worse - it is like a "goto destination" command - almost meaningless without reading lots of context.

Examining all the different uses that void pointers can be put to would be well beyond the scope of this article. Instead we will restrict our attention to just one new usage which relates to data inheritance and illustrates how the untamed nature of void pointers makes it hard to recognize their use in data inheritance. The example we will use to explain this usage is struct seq_file used by the seq_file library which makes it easy to synthesize simple text files like some of those in /proc. The "seq" part of seq_file simply indicates that the file contains a sequence of lines corresponding to a sequence of items of information in the kernel, so /proc/mounts is a seq_file which walks through the mount table reporting each mount on a single line.

When seq_open() is used to create a new seq_file it allocates a struct seq_file and assigns it to the private_data field of the struct file which is being opened. This is a straightforward example of void pointer based data inheritance where the struct file is the base type and the struct seq_file is a simple extension to that type. It is a structure that never exists by itself but is always the private_data for some file. struct seq_file itself has a private field which is a void pointer and it can be used by clients of seq_file to add extra state to the file. For example md_seq_open() allocates a struct mdstat_info structure and attaches it via this private field, using it to meet md's internal needs. Again, this is simple data inheritance following the described pattern.

However the private field of struct seq_file is used by svc_pool_stats_open() in a subtly but importantly different way. In this case the extra data needed is just a single pointer. So rather than allocating a local data structure to refer to from
the private field, svc_pool_stats_open simply stores that pointer directly in
the private field itself. This certainly seems like a sensible optimization - performing an allocation to store a single pointer would be a waste - but it highlights exactly the source of confusion that was suggested earlier: that when you look at a void pointer you don't really know what is it pointing at, or why.

To make it a bit clearer what is happening here, it is helpful to imagine "void *private" as being like a union of every different possible pointer type. If the value that needs to be stored is a pointer, it can be stored in this union following the "unions for data inheritance" pattern. If the value is not a single pointer, then it gets stored in allocated space following the "void pointers for data inheritance" pattern. Thus when we see a void pointer being used it may not be obvious whether it is being used to **point to** an extension structure for data inheritance, or being used **as** an extension for data inheritance (or being used as something else altogether).

To highlight this issue from a slightly different perspective it is instructive to examine struct v4l2_subdev which represents a sub-device in a video4linux device, such as a sensor or camera controller within a webcam. According to the (rather helpful) documentation it is expected that this structure will normally be embedded in a larger structure which contains extra state. However this structure still has not just one but two void pointers, both with names suggesting that they are for private use by subtypes:

```
        /* pointer to private data */
        void *dev_priv;
        void *host_priv;
```

It is common that a v4l sub-device (a sensor, usually) will be realized by, for example, an I2C device (much as a block device which stores your filesystem might be realized by an ATA or SCSI device). To allow for this common occurrence, struct v4l2_subdev provides a void pointer (dev_priv), so that the driver itself doesn't need to define a more specific pointer in the larger structure which struct v4l2_subdev would be embedded in. host_priv is intended to point back to a "parent" device such as a controller which acquires video data from the sensor. Of the three drivers which use this field, one appears to follow that intention while the other two use it to point to an allocated extension structure. So both of these pointers are intended to be used following the "unions for data inheritance" pattern, where a void pointer is playing the role of a union of many other pointer types, but they are not always used that way.

It is not immediately clear that defining this void pointer in case it is useful is actually a valuable service to provide given that the device driver could easily enough define its own (type safe) pointer in its extension structure. What is clear is that an apparently "private" void pointer can be intended for various qualitatively different uses and, as we have seen in two different circumstances, they may not be used exactly as expected.

In short, recognizing the "data inheritance through void pointers" pattern is not easy. A fairly deep examination of the code is needed to determine the exact purpose and usage of void pointers.

A diversion into struct page

Before we leave unions and void pointers behind a look at struct page may be interesting. This structure uses both of these patterns, though they are hidden somewhat due to historical baggage. This example is particularly instructive because it is one case where struct embedding simply is not an option.

In Linux memory is divided into pages, and these pages are put to a variety of different uses. Some are in the "page cache" used to store the contents of files. Some are "anonymous pages" holding data used by applications. Some are used as "slabs" and divided into pieces to answer kmalloc() requests. Others are simply part of a multi-page allocation or maybe are on a free list waiting to be used. Each of these different use cases could be seen as a subtype of the general class of "page", and in most cases need some dedicated fields in struct page, such as a struct address_space pointer and index when used in the page cache, or struct kmem_cache and freelist pointers when used as a slab.

Each page always has the same struct page describing it, so if the effective type of the page is to change - as it must as the demands for different uses of memory change over time - the type of the struct page must change within the lifetime of that structure. While many type systems are designed assuming that the type of an object is immutable, we find here that the kernel has a very real need for type mutability. Both unions and void pointers allow types to change and as noted, struct page uses both.

At the first level of subtyping there are only a small number of different subtypes as listed above; these are all known to the core memory management code, so a union would be ideal here. Unfortunately struct page has three unions with fields for some subtypes spread over all three, thus hiding the real structure somewhat.

When the primary subtype in use has the page being used in the page cache, the particular address_space that it belongs to may want to extend the data structure further. For this purpose there is a private field that can be used. However it is not a void pointer but is an unsigned long. Many places in the kernel assume an unsigned long and a void * are the same size and this is one of them. Most users of this field actually store a pointer here and have to cast it back and forth. The "buffer_head" library provides macros attach_page_buffers and page_buffers to set and get this field.

So while struct page is not the most elegant example, it is an informative example of a case where unions and void pointers are the only option for providing data inheritance.

The details of structure embedding

Where structure embedding can be used, and where the list of possible subtypes is not known in advance, it seems to be increasingly the preferred choice. To gain a full understanding of it we will again need to explore a little bit further than inodes and contrast data inheritance with other uses of structure embedding.

There are essentially three uses for structure embedding - three reasons for including a structure within another structure. Sometimes there is nothing particularly interesting going on. Data items are collected together into structures and structures within structures simply to highlight the closeness of the relationships between the different items. In this case the address of the embedded structure is rarely taken, and it is never mapped back to the containing structure using container_of().

The second use is the data inheritance embedding that we have already discussed. The third is like it but importantly different. This third use is typified by struct list_head and other structs used as an embedded anchor when creating abstract data types.

The use of an embedded anchor like struct list_head can be seen as a style of inheritance as the structure containing it "is-a" member of a list by virtue of inheriting from struct list_head. However it is not a strict subtype as a single object can have several struct list_heads embedded - struct inode has six (if we include the similar hlist_node). So it is probably best to think of this sort of embedding more like a "mixin" style of inheritance. The struct list_head provides a service - that of being included in a list - that can be mixed-in to other objects, an arbitrary number of times.

A key aspect of data inheritance structure embedding that differentiates it from each of the other two is the existence of a reference counter in the inner-most structure. This is an observation that is tied directly to the fact that the Linux kernel uses reference counting as the primary means of lifetime management and so would not be shared by systems that used, for example, garbage collection to manage lifetimes.

In Linux, every object with an independent existence will have a reference counter, sometimes a simple atomic_t or even an int, though often a more explicit struct kref. When an object is created using several levels of inheritance the reference counter could be buried quite deeply. For example a struct usb_device embeds a struct device which embeds struct kobject which has a struct kref. So usb_device (which might in turn be embedded in a structure for some specific device) does have a reference counter, but it is contained several levels down in the nest of structure embedding. This contrasts quite nicely with a list_head and similar structures. These have no reference counter, have no independent existence and simply provide a service to other data structures.

Though it seems obvious when put this way, it is useful to remember that a single object cannot have two reference counters - at least not two lifetime reference counters (It is fine to have two counters like s_active and s_count in struct super_block which count different things). This means that multiple inheritance in the "data inheritance" style is not possible. The only form of multiple inheritance that can work is the mixin style used by list_head as mentioned above.

It also means that, when designing a data structure, it is important to think about lifetime issues and whether this data structure should have its own reference counter or whether it should depend on something else for its lifetime management. That is, whether it is an object in its own right, or simply a service provided to other objects. These issues are not really new and apply equally to void pointer inheritance. However an important difference with void pointers is that it is relatively easy to change your mind later and switch an extension structure to be a fully independent object. Structure embedding requires the discipline of thinking clearly about the problem up front and making the right decision early - a discipline that is worth encouraging.

The other key telltale for data inheritance structure embedding is the set of rules for allocating and initializing new instances of a structure, as has already been hinted at. When union or void pointer inheritance is used the main structure is usually allocated and initialized by common code (the mid-layer) and then a device specific open() or create() function is called which can optionally allocate and initialize any extension object. By contrast when structure embedding is used the structure needs to be allocated by the lowest level device driver which then initializes its own fields and calls in to common code to initialize the common fields.

Continuing the struct inode example from above which has an alloc_inode() method in the super_block to request allocation, we find that initialization is provided for with inode_init_once() and inode_init_always() support functions. The first of these is used when the previous use of a piece of memory is unknown, the second is sufficient by itself when we know that the memory was previously used for some other inode. We see this same pattern of an initializer function separate from allocation in kobject_init(), kref_init(), and device_initialize().

So apart from the obvious embedding of structures, the pattern of "data inheritance through structure embedding" can be recognized by the presence of a reference counter in the innermost structure, by the delegation of structure allocation to the final user of the structure, and by the provision of initializing functions which initialize a previously allocated structure.

Conclusion

In exploring the use of method dispatch (last week) and data inheritance (this week) in the Linux kernel we find that while some patterns seem to dominate they are by no means universal. While almost all data inheritance could be implemented using structure embedding, unions provide real value in a few specific cases. Similarly while simple vtables are common, mixin vtables are very important and the ability to delegate methods to a related object can be valuable.

We also find that there are patterns in use with little to recommend them. Using void pointers for inheritance may have an initial simplicity, but causes longer term wastage, can cause confusion, and could nearly always be replaced by embedded inheritance. Using NULL pointers to indicate default behavior is similarly a poor choice - when the default is important there are better ways to provide for it.

But maybe the most valuable lesson is that the Linux kernel is not only a useful program to run, it is also a useful document to study. Such study can find elegant practical solutions to real problems, and some less elegant solutions. The willing student can pursue the former to help improve their mind, and pursue the latter to help improve the kernel itself. With that in mind, the following exercises might be of interest to some.

Exercises

1. As inodes now use structure embedding for inheritance, void pointers should not be necessary. Examine the consequences and wisdom of removing "i_private" from "struct inode".
2. Rearrange the three unions in struct page to just one union so that the enumeration of different subtypes is more explicit.
3. As was noted in the text, struct seq_file can be extended both through "void pointer" and a limited form of "union" data inheritance. Explain how seq_open_private() allows this structure to also be extended through "embedded structure" data inheritance and give an example by converting one usage in the kernel from "void pointer" to "embedded structure". Consider submitting a patch if this appears to be an improvement. Contrast this implementation of embedded structure inheritance with the mechanism used for inodes.
4. Though subtyping is widely used in the kernel, it is not uncommon for a object to contain fields that not all users are interested in. This can indicate that more fine grained subtyping is possible. As very many completely different things can be represented by a "file descriptor", it is likely that struct file could be a candidate for further subtyping.

   Identify the smallest set of fields that could serve as a generic struct file and explore the implications of embedding that in different structures to implement regular files, socket files, event files, and other file types. Exploring more general use of the proposed open() method for inodes might help here.

5. Identify an "object-oriented" language which has an object model that would meet all the needs of the Linux kernel as identified in these two articles.