Linux kernel design patterns - part 1

One of the topics of ongoing interest in the kernel community is that of maintaining quality. It is trivially obvious that we need to maintain and even improve quality. It is less obvious how best to do so. One broad approach that has found some real success is to increase the visibility of various aspects of the kernel. This makes the quality of those aspects more apparent, so this tends to lead to an improvement of the quality.

This increase in visibility takes many forms:

- The checkpatch.pl script highlights many deviations from accepted code formatting style. This encourages people (who remember to use the script) to fix those style errors. So, by increasing the visibility of the style guide, we increase the uniformity of appearance and so, to some extent, the quality.
- The "lockdep" system (when enabled) dynamically measures dependencies between locks (and related states such as whether interrupts are enabled). It then reports anything that looks odd. These oddities will not always mean a deadlock or similar problem is possible, but in many cases they do, and the deadlock possibility can be removed. So by increasing the visibility of the lock dependency graph, quality can be increased.
- The kernel contains various other enhancements to visibility such as the "poisoning" of unused areas of memory so invalid access will be more apparent, or simply the use of symbolic names rather than plain hexadecimal addresses in stack traces so that bug reports are more useful.
- At a much higher level, the "git" revision tracking software that is used for tracking kernel changes makes it quite easy to see who did what and when. The fact that it encourages a comment to be attached to each patch makes it that much easier to answer the question "Why is the code this way". This visibility can improve understanding and that is likely to improve quality as more developers are better informed.

There are plenty of other areas where increasing visibility does, or could, improve quality. In this series we will explore one particular area where your author feels visibility could be increased in a way that could well lead to qualitative improvements. That area is the enunciation of kernel-specific design patterns.

Design Patterns

A "design pattern" is a concept that was first expounded in the field of Architecture and was brought to computer engineering, and particularly the Object Oriented Programming field, though the 1994 publication *Design Patterns: Elements of Reusable Object-Oriented Software*. Wikipedia has further useful background information on the topic.

In brief, a design pattern describes a particular class of design problem, and details an approach to solving that class of problem that has proven effective in the past. One particular benefit of a design pattern is that it combines the problem description and the solution description together and gives them a name. Having a simple and memorable name for a pattern is particularly valuable. If both developer and reviewer know the same names for the same patterns, then a significant design decision can be communicated in one or two words, thus making the decision much more visible.

In the Linux kernel code base there are many design patterns that have been found to be effective. However most of them have never been documented so they are not easily available to other developers. It is my hope that by explicitly documenting these patterns, I can help them to be more widely used and, thus, developers will be able to achieve effective solutions to common problems more quickly.

In the remainder of this series we will be looking at three problem domains and finding a variety of design patterns of greatly varying scope and significance. Our goal in doing so is to not only to enunciate these patterns, but also to show the range and value of such patterns so that others might make the effort to enunciate patterns that they have seen.

A number of examples from the Linux kernel will be presented throughout this series as examples are an important part of illuminating any pattern. Unless otherwise stated they are all from 2.6.30-rc4.

Reference Counts

The idea of using a reference counter to manage the lifetime of an object is fairly common. The core idea is to have a counter which is incremented whenever a new reference is taken and decremented when a reference is released. When this counter reaches zero any resources used by the object (such as the memory used to store it) can be freed.

The mechanisms for managing reference counts seem quite straightforward. However there are some subtleties that make it quite easy to get the mechanisms wrong. Partly for this reason, the Linux kernel has (since 2004) a data type known as "kref" with associated support routines (see Documentation/kref.txt, <linux/kref.h>, and lib/kref.c). These encapsulate some of those subtleties and, in particular, make it clear that a given counter is being used as a reference counter in a particular way. As noted above, names for design patterns are very valuable and just providing that name for kernel developers to use is a significant benefit for reviewers.

In the words of Andrew Morton:

I care more about being able to say "ah, it uses kref. I understand that refcounting idiom, I know it's well debugged and I know that it traps common errors". That's better than "oh crap, this thing implements its own refcounting - I need to review it for the usual errors".

This inclusion of kref in the Linux kernel gives both a tick and a cross to the kernel in terms of explicit support for design patterns. A tick is deserved as the kref clearly embodies an important design pattern, is well documented, and is clearly visible in the code when used. It deserves a cross however because the kref only encapsulates part of

the story about reference counting. There are some usages of reference counting that do not fit well into the kref model as we will see shortly. Having a "blessed" mechanism for reference counting that does not provide the required functionality can actually encourage mistakes as people might use a kref where it doesn't belong and so think it should just work where in fact it doesn't.

A useful step to understanding the complexities of reference counting is to understand that there are often two distinctly different sorts of references to an object. In truth there can be three or even more, but that is very uncommon and can usually be understood by generalizing the case of two. We will call the two types of references "external" and "internal", though in some cases "strong" and "weak" might be more appropriate.

An "external" reference is the sort of reference we are probably most accustomed to think about. They are counted with "get" and "put" and can be held by subsystems quite distant from the subsystem that manages the object. The existence of a counted external reference has a strong and simple meaning: This object is in use.

By contrast, an "internal" reference is often not counted, and is only held internally to the system that manages the object (or some close relative). Different internal references can have very different meanings and hence very different implications for implementation.

Possibly the most common example of an internal reference is a cache which provides a "lookup by name" service. If you know the name of an object, you can apply to the cache to get an external reference, providing the object actually exists in the cache. Such a cache would hold each object on a list, or on one of a number of lists under e.g. a hash table. The presence of the object on such a list is a reference to the object. However it is likely not a counted reference. It does not mean "this object is in use" but only "this object is hanging around in case someone wants it". Objects are not removed from the list until all external references have been dropped, and possibly they won't be removed

immediately even then. Clearly the existence and nature of internal references can have significant implications on how reference counting is implemented.

One useful way to classify different reference counting styles is by the required implementation of the "put" operation. The "get" operation is always the same. It takes an external reference and produces another external reference. It is implemented by something like:

```
assert(obj->refcount > 0) ; increment(obj->refcount);
```

or, in Linux-kernel C:

```
BUG_ON(atomic_read(&obj->refcnt)) ; atomic_inc(&obj->refcnt);
```

Note that "get" cannot be used on an unreferenced object. Something else is needed there.

The "put" operation comes in three variations. While there can be some overlap in use cases, it is good to keep them separate to help with clarity of the code. The three options, in Linux-C, are:

```
1    atomic_dec(&obj->refcnt);


2    if (atomic_dec_and_test(&obj->refcnt)) { ... do stuff ... }


3    if (atomic_dec_and_lock(&obj->refcnt, &subsystem_lock)) {
         ..... do stuff ....
                  spin_unlock(&subsystem_lock);
     }
```

The "kref" style

Starting in the middle, option "2" is the style used for a kref. This style is appropriate when the object does not outlive its last external reference. When that reference count becomes zero, the object needs to be freed or otherwise dealt with, hence the need to test for the zero condition with atomic_dec_and_test().

Objects that fit this style often do not have any internal references to worry about, as is the case with most objects in sysfs, which is a heavy user of kref. If, instead, an object using the kref style does have internal references, it cannot be allowed to create an external reference from an internal reference unless there are known to be other external references. If this is necessary, a primitive is available:

```
atomic_inc_not_zero(&obj->refcnt);
```

which increments a value providing it isn't zero, and returns a result indicating success or otherwise. atomic_inc_not_zero() is a relatively recent invention in the linux kernel, appearing in late 2005 as part of the lockless page cache work. For this reason it isn't widely used and some code that could benefit from it uses spinlocks instead. Sadly, the kref package does not make use of this primitive either.

An interesting example of this style of reference that does not use kref, and does not even use atomic_dec_and_test() (though it could and arguably should) are the two ref counts in struct super: s_count and s_active.

s_active fits the kref style of reference counts exactly. A superblock starts life with s_active being 1 (set in alloc_super()), and, when s_active becomes zero, further external references cannot be taken. This rule is encoded in grab_super(), though this is not immediately clear. The current code (for historical reasons) adds a very large value (S_BIAS) to s_count whenever s_active is non-zero, and grab_super() tests for s_count exceeding S_BIAS rather than for s_active being zero. It could just as easily do the latter test using atomic_inc_not_zero(), and avoid the use of spinlocks.

s_count provides for a different sort of reference which has both "internal" and "external" aspects. It is internal in that its semantic is much weaker than that of s_active-counted references. References counted by s_count just mean "this superblock cannot be freed just now" without asserting that it is actually active. It is external in that it is much like a kref starting life at 1 (well, 1*S_BIAS actually), and when it becomes zero (in __put_super()) the superblock is destroyed.

So these two reference counts could be replaced by two krefs, providing:

- S_BIAS was set to 1
- grab_super() used atomic_inc_not_zero() rather than testing against S_BIAS

and a number of spinlock calls could go away. The details are left as an exercise for the reader.

The "kcref" style

The Linux kernel doesn't have a "kcref" object, but that is a name that seems suitable to propose for the next style of reference count. The "c" stands for "cached" as this style is very often used in caches. So it is a Kernel Cached REFerence.

A kcref uses atomic_dec_and_lock() as given in option 3 above. It does this because, on the last put, it needs to be freed or checked to see if any other special handling is needed. This needs to be done under a lock to ensure no new reference is taken while the current state is being evaluated.

A simple example here is the i_count reference counter in struct inode. The important part of iput() reads:

```
if (atomic_dec_and_lock(&inode->i_count, &inode_lock))
        iput_final(inode);
```

where iput_final() examines the state of the inode and decides if it can be destroyed, or left in the cache in case it could get reused soon.

Among other things, the inode_lock prevents new external references being created from the internal references of the inode hash table. For this reason converting internal references to external references is only permitted while the inode_lock is held. It is no accident that the function supporting this is called iget_locked() (or iget5_locked()).

A slightly more complex example is in struct dentry, where d_count is managed like a kcref. It is more complex because two locks need to be taken before we can be sure no new reference can be taken - both dcache_lock and de->d_lock. This requires that either we hold one lock, then atomic_dec_and_lock() the other (as in prune_one_dentry()), or that we atomic_dec_and_lock() the first, then claim the second and retest the refcount - as in dput(). This is good example of the fact that you can never assume you have encapsulated all possible reference counting styles. Needing two locks could hardly be foreseen.

An even more complex kcref-style refcount is mnt_count in struct vfsmount. The complexity here is the interplay of the two refcounts that this structure has: mnt_count, which is a fairly straightforward count of external references, and mnt_pinned, which counts internal references from the process accounting module. In particular it counts the number of accounting files that are open on the filesystem (and as such could use a more meaningful name). The complexity comes from the fact that when there are only internal references remaining, they are all converted to external references. Exploring the details of this is again left as an exercise for the interested reader.

The "plain" style

The final style for refcounting involves just decrementing the reference count (atomic_dec()) and not doing anything else. This style is relatively uncommon in the

kernel, and for good reason. Leaving unreferenced objects just lying around isn't a good idea.

One use of this style is in struct buffer_head, managed by fs/buffer.c and <linux/buffer_head.h>. The put_bh() function is simply:

```
static inline void put_bh(struct buffer_head *bh)
{
    smp_mb__before_atomic_dec();
    atomic_dec(&bh->b_count);
}
```

This is OK because buffer_heads have lifetime rules that are closely tied to a page. One or more buffer_heads get allocated to a page to chop it up into smaller pieces (buffers). They tend to remain there until the page is freed at which point all the buffer_heads will be purged (by drop_buffers() called from try_to_free_buffers()).

In general, the "plain" style is suitable if it is known that there will always be an internal reference so that the object doesn't get lost, and if there is some process whereby this internal reference will eventually get used to find and free the object.

Anti-patterns

To wrap up this little review of reference counting as an introduction to design patterns, we will discuss the related concept of an anti-pattern. While design patterns are approaches that have been shown to work and should be encouraged, anti-patterns are approaches that history shows us do not work well and should be discouraged.

Your author would like to suggest that the use of a "bias" in a refcount is an example of an anti-pattern. A bias in this context is a large value that is added to, or subtracted from, the reference count and is used to effectively store one bit of information. We have

already glimpsed the idea of a bias in the management of s_count for superblocks. In this case the presence of the bias indicates that the value of s_active is non-zero, which is easy enough to test directly. So the bias adds no value here and only obscures the true purpose of the code.

Another example of a bias is in the management of struct sysfs_dirent, in fs/sysfs/sysfs.h and fs/sysfs/dir.c. Interestingly, sysfs_dirent has two refcounts just like superblocks, also called s_count and s_active. In this case s_active has a large negative bias when the entry is being deactivated. The same bit of information could be stored just as effectively and much more clearly in the flag word s_flags. Storing single bits of information in flags is much easier to understand that storing them as a bias in a counter, and should be preferred.

In general, using a bias does not add any clarity as it is not a common pattern. It cannot add more functionality than a single flag bit can provide, and it would be extremely rare that memory is so tight that one bit cannot be found to record whatever would otherwise be denoted by the presence of the bias. For these reasons, biases in refcounts should be considered anti-patterns and avoided if at all possible.

Conclusion

This brings to a close our exploration of the various design patterns surrounding reference counts. Simply having terminology such a "kref" versus "kcref" and "external" versus "internal" references can be very helpful in increasing the visibility of the behaviour of different references and counts. Having code to embody this as we do with kref and could with kcref, and using this code at every opportunity, would be a great help both to developers who might find it easy to choose the right model first time, and to reviewers who can see more clearly what is intended.

The design patterns we have covered in this article are:

- **kref**: When the lifetime of an object extends only to the moment that the last external reference is dropped, a kref is appropriate. If there are any internal reference to the object, they can only be promoted to external references with atomic_inc_not_zero(). Examples: s_active and s_count in struct super_block.
- **kcref**: With this the lifetime of an object can extend beyond the dropping of the last external reference, the kcref with its atomic_dec_and_lock() is appropriate. An internal reference can only be converted to an external reference will the subsystem lock is held. Examples: i_count in struct inode.
- **plain**: When the lifetime of an object is subordinate to some other object, the plain reference pattern is appropriate. Non-zero reference counts on the object must be treated as internal reference to the parent object, and converting internal references to external references must follow the same rules as for the parent object. Examples: b_count in struct buffer_head.
- **biased-reference**: When you feel the need to use add a large bias to the value in a reference count to indicate some particular state, don't. Use a flag bit elsewhere. This is an anti-pattern.

Next week we will move on to another area where the Linux kernel has proved some successful design patterns and explore the slightly richer area of complex data structures. (Part 2 and part 3 of this series are now available).

Exercises

As your author has been reminded while preparing this series, there is nothing like a directed study of code to clarify understanding of these sorts of issues. With that in mind, here are some exercises for the interested reader.

1. Replace s_active and s_count in struct super with krefs, discarding S_BIAS in the process. Compare the result with the original using the trifecta of Correctness, Maintainability, and Performance.
2. Choose a more meaningful name for mnt_pinned and related functions that manipulate it.
3. Add a function to the kref library that makes use of atomic_inc_not_zero(), and using it (or otherwise) remove the use of atomic_dec_and_lock() on a kref in net/sunrpc/svcauth.c - a usage which violates the kref abstraction.
4. Examine the _count reference count in struct page (see mm_types.h for example) and determine whether it behaves most like a kref or a kcref (hint: it is not "plain"). This should involve identifying any and all internal references and related locking rules. Identify why the page cache (struct address_space.page_tree) owns a counted reference or explain why it should not. This will involve understanding page_freeze_refs() and its usage in __remove_mapping(), as well as page_cache_{get,add}_speculative().

Bonus credit: provide a series of minimal self-contained patches to implement any changes that the above investigations proved useful.

Linux kernel design patterns - part 2

Last week we discussed the value of enunciating kernel design patterns and looked at the design patterns surrounding reference counts. This week we will look at a very different aspect of coding and see why the kernel has special needs, and how those needs have been addressed by successful approaches. The topic under the microscope today is complex data structures.

By "complex data structures" we mean objects that are composed of a variable number of simpler objects, possibly a homogeneous collection, possibly a heterogeneous collection. In general it is a structure to which objects can be added, from which objects can be

removed, and in which objects can be found. The preferred way to work with such data structures when we study them in an introductory programming course is to use Abstract Data Types.

Abstract Data Types

The idea behind Abstract Data Types is to encapsulate the entire implementation of a data structure, and provide just a well defined interface for manipulating it. The benefit of this approach is that it provides a clean separation. The data type can be implemented with no knowledge of the application which might end up using it, and the application can be implemented with no knowledge of the implementation details of the data type. Both sides simply write code based on the interface which works like a contract to explicitly state what is required and what can be expected.

On the other hand, one of the costs of this approach is tightly connected with the abstract nature of the interface. The point of abstracting an interface is to remove unnecessary details. This is good for introductory computing students, but is bad for kernel programmers. In kernel programming. performance is very important, coming as a close third after correctness and maintainability, and sometimes taking precedence over maintainability. Not all code paths in the kernel are performance-critical, but many are, and the development process benefits from the same data structures being using in both performance critical and less critical paths. So it is essential that data types are not overly abstracted, but that all details of the implementation are visible so that the programmer can make optimal choices when using them.

So the first principle of data structures in the kernel is not to hide detail. To see how this applies, and to discover further principles from which to extract patterns, we will explore a few of the more successful data types used in the Linux kernel.

Linked Lists

Starting simply, the first data type we will explore are doubly linked lists. These are implemented by a single include file, <linux/list.h>. There is no separate ".c" file with any library of support routines. All of the code for handling linked lists is simple enough to be implemented using inline functions. Thus it is very easy to use this implementation in any other (GPLv2-licensed) project.

There are two aspects of the "list.h" lists which are worth noting as they point to possible patterns. The first is struct list_head, which serves not only as the head of a list, but also as the anchor in items that are on a list. Your author has seen other linked list implementations which required that the first and second element in any data structures meant to be stored in lists be the "next" and "previous" pointers, so that common list-walking code could be used on a variety of different data structures. Linux kernel lists do not suffer from this restriction. The list_head structure can be embedded anywhere in a data structure, and the list_heads from a number of instances of that structure can be linked together. The containing structure can be found from a ->next or ->prev pointer using the list_entry() macro.

There are at least two benefits of this approach. One is that the programmer still has full control of placement of fields in the structure in case they need to put important fields close together to improve cache utilization. The other is that a structure can easily be on two or more lists quite independently, simply by having multiple struct list_head fields.

This practice of embedding one structure inside another and using container_of() (which is the general form of list_entry()) to get the parent from the child structure is quite common in the Linux kernel and is somewhat reminiscent of object oriented programming. The container is like a subtype of the embedded structure.

The other noteworthy aspect of list.h is the proliferation of "for_each" macros - the macros that make it easy to walk along a list looking at each item in turn. There are 20 of

them (and that isn't counting the four more in rculist.h which I'll choose to ignore in the hope of brevity).

There are a few different reasons for this. The simplest are that

- We sometimes want to walk the list in the "reverse" direction (following the "prev" link). There are five macros that go backward, and 15 that go forward.
- We sometimes want to start in the middle of a list and "continue" on from there, so we have four "continue" macros and three "from" macros which interpret that starting point slightly differently.
- We sometimes want to work with the struct list_head embedded in the target structure, but often we really want to use the list_entry() macro to get the enclosing structure; we find it easiest if the "for_each" macro does that for us. This provides the "entry" versions of the "for_each" macro, of which there are 13 (more than half).

Getting to the more subtle reasons, we sometimes want to be able to delete the "current" item without upsetting the walk through the list. This requires that a copy of the "next" pointer be taken before providing "this" entry to be acted upon, thus yielding the eight "safe" macros. An "ADT" style implementation of linked lists would quite likely only provide "safe" versions so as to hide these details. However kernel programmers don't want to waste the storage or effort for that extra step in the common case were it isn't needed.

Then there is the fact that we actually have two subtly different types of linked lists. Regular linked lists use struct list_head as the head of the list. This structure contains a pointer to the start and to the end. In some use cases, finding the end of the list is not needed, and being able to halve the size of the head of the list is very valuable. One typical use case of that kind is a hash table where all these heads need to go in an array. To meet this need, we have the hlist, which is very similar to the regular list, except that

only one pointer is needed in struct hlist_head. This accounts for six of the different "for_each" macros.

If we had every possibly combination of forward or reverse, continue or not, entry or not, safe or not, and regular or hlist, we would have 32 different macros. In fact, only 19 of these appear to have been needed and, thus, coded. We certainly could code the remaining eleven, but as having code that is never used tends to be frowned upon, it hasn't happened.

The observant reader will have noticed a small discrepancy in some of the above numbers. Of the 20 macros, there is one that doesn't fit the above patterns, and it drives home the point that was made earlier about kernel programmers valuing performance. This final "for_each" macro is __list_for_each(). All of the other macros use the "prefetch" function to suggest that the CPU starts fetching the ->next pointer at the start of each iteration so that it will already be available in cache when the next iteration starts (though the "safe" macros actually fetch it rather than prefetch it). While this will normally improve performance, there are cases when it will slow things down. When the walk of the list will almost always abort very early - usually only considering the first item - the prefetch will often be wasted effort. In these cases (currently all in the networking code) the __list_for_each() macro is available. It does not prefetch anything. Thus people having very strict performance goals can have a better chance of getting the performance they want.

So from this simple data structure we can see two valuable patterns that are worth following.

- **Embedded Anchor**: A good way to include generic objects in a data structure is to embed an anchor in them and build the data structure around the anchors. The object can be found from the anchor using container_of().

- **Broad Interfaces**: Don't fall for the trap of thinking that "one size fits all". While having 20 or more macros that all do much the same thing is uncommon, it can be a very appropriate way of dealing with the complexity of finding the optimal solution. Trying to squeeze all possibilities into one narrow interface can be inefficient and choosing not to provide for all possibilities is counter-productive. Having all the permutations available encourages developers to use the right tool for the job and not to compromise. In 2.6.30-rc4, there are nearly 3000 uses of list_for_each_entry(), about 1000 of list_for_each_entry_safe(), nearly 500 of list_for_each(), and less than 1000 of all the rest put together. The fact that some are used rarely in no way reduces their importance.

RB-trees

Our next data structure is the RB-Tree or red-black tree. This is a semi-balanced, binary search tree that generally provides order "log(n)" search, insert, and delete operations. It is implemented in <linux/rbtree.h> and lib/rbtree.c. It has strong similarities to the list.h lists in that it embeds an anchor (struct rb_node) in each data structure and builds the tree from those.

The interesting thing to note about rbtree is that there is no search function. Searching an rbtree is really a very simple operation and can be implemented in just a few lines as shown by the examples at the top of rbtree.h. A search function certainly could be written, but the developer chose not to. The main reason, which should come as no surprise, is performance. To write a search function, you need to pass the "compare" function into that search function. To do that in C, you would need to pass a function pointer. As compare functions are often very simple, the cost of following the function pointer and making the function call would often swamp the cost of doing the comparison itself. It turns out that having the whole search operation compiled as one function makes for more efficient code. The same performance could possibly be achieved using inline

functions or macros, but given that the search function itself is so short, it hardly seems worthwhile.

Note also that rbtree doesn't exactly provide an insert function either. Rather, the developer needs to code a search; if the search fails, the new node must be inserted at the point where it was found not to exist and the tree must be rebalanced. There are functions for this final insertion and rebalancing as they are certainly complex enough to deserve separate functions.

By giving the developer the responsibility for search and for some of insert, the rbtree library actually is giving a lot of valuable freedom. The pattern of "search for an entry but if it isn't there, insert one" is fairly common. However the details of what happens between the "search" and "add" phases is not always the same and so not something that can easily be encoded in a library. By providing the basic tools and leaving the details up to the specific situation, users of rbtree find themselves liberated, rather than finding themselves fighting with a library that almost-but-not-quite does what they want.

So this example of rbtrees re-enforces the "embedded anchors" pattern and suggests a pattern that providing tools is sometimes much more useful than providing complete solutions. In this case, the base data structures and the tools required for insert, remove, and rebalance are provided, but the complete solution can still be tailored to suit each case.

This pattern also describes the kernel's approach to hash tables. These are a very common data structure, but there is nothing that looks even vaguely like a definitive implementation. Rather the basic building blocks of the hlist and the array are available along with some generic functions for calculating a hash (<linux/hash.h>). Connecting these to fit a given purpose is up to the developer.

So we have another pattern:

- **Tool Box**: Sometimes it is best not to provide a complete solution for a generic service, but rather to provide a suite of tools that can be used to build custom solutions.

Radix tree

Our last data structure is the Radix tree. The Linux kernel actually has two radix tree implementations. One is in <linux/idr.h> and lib/idr.c, the other in <linux/radix-tree.h> and lib/radix-tree.c. Both provide a mapping from a number (unsigned long) to an arbitrary pointer (void *), though radix-tree also allows up to two "tags" to be stored with each entry. For the purposes of this article we will only be looking at one of the implementations (the one your author is most familiar with) - the radix-tree implementation.

Radix-tree follows the pattern we saw in list.h of having multiple interfaces rather than trying to pack lots of different needs into the one interface. list.h has 20 "for_each" macros; radix-tree has six "lookup" functions, depending on whether we want just one item or a range (gang lookups), or whether we want to use the tags to restrict the search (tag lookups) and whether we want to find the place where the pointer is stored, rather than the pointer that is stored there (this is needed for the subtle locking strategies of the page cache).

However radix-tree does not follow the embedded anchor pattern of the earlier data structures, and that is why it is interesting. For lists and rbtree, the storage needed for managing the data structure is exactly proportional to the number of items in the data structure on a one-to-one basis, so keeping this storage in those item works perfectly. For a radix-tree, the storage needed is a number of little arrays, each of which refers to a number of items. So embedding these arrays, one each, in the items cannot work. This means that, unlike list.h and rbtree, radix-tree will sometimes need to allocate some

memory in order to be able to add items to the data structure. This has some interesting consequences.

In the previous data structures (lists and rbtrees), we made no mention of locking. If locking is needed, then the user of the data structure is likely to know the specific needs so all locking details are left up to the caller (we call that "caller locking" as opposed to "callee locking". Caller locking is more common and generally preferred). This is fine for lists and rbtrees as nothing that they do internally is affected particularly by locking.

This is not the case if memory allocation is needed, though. If a process needs to allocate memory, it is possible that it will need to sleep while the memory management subsystem writes data out to storage to make memory available. There are various locks (such as spinlocks) that may not be held while a process sleeps. So there is the possibility for significant interaction between the need to allocate memory internally to the radix-tree code, and the need to hold locks outside the radix-tree code.

The obvious solution to this problem (once the problem is understood) is to preallocate the maximum amount of memory needed before taking the lock. This is implemented within radix-tree with the radix_tree_preload() function. It manages a per-CPU pool of available radix_tree nodes and makes sure the pool is full and will not be used by any other radix-tree operation. Thus, bracketing radix_tree_insert() with calls to radix_tree_preload() and radix_tree_preload_end() ensures that the radix_tree_insert() call will not fail due to lack of memory (though the radix_tree_preload() might) and that there will be no unpleasant interactions with locking.

Summary

So we can now summarize our list of design patterns that we have found that work well with data structures (and elsewhere) in the kernel. Those that have already been detailed are briefly included in this list too for completeness.

- **Embedded Anchor**. This is very useful for lists, and can be generalized as can be seen if you explore kobjects (an exercise left to the reader).
- **Broad Interfaces**. This reminds us that trying to squeeze lots of use-cases in to one function call is not necessary - just provide lots of function calls (with helpful and (hopefully) consistent names).
- **Tool Box**. Sometimes it is best not to provide a complete solution for a generic service, but rather to provide a suite of tools that can be used to build custom solutions.
- **Caller Locks**. When there is any doubt, choose to have the caller take locks rather than the callee. This puts more control in that hands of the client of a function.
- **Preallocate Outside Locks**. This is in some ways fairly obvious. But it is very widely used within the kernel, so stating it explicitly is a good idea.

Next week we will complete our investigation of kernel design patterns by taking a higher level view and look at some patterns relating to the design of whole subsystems.

Exercises

For those who would like to explore these ideas further, here are some starting points.

1. Make a list of all data structures that are embedded, by exploring all uses of the "container_of" macro. Of these, make a list of pairs that are both embedded in the same structure (modeling multiple inheritance). Comment on how this reflects on the general usefulness of multiple inheritance.
2. Write a implementation of skiplists that would be suitable for in-kernel use. Consider the applicability of each of the patterns discussed in this article. Extra credit for leveraging list.h lists.
3. Linux contains a mempool library which radix-tree chooses not to use, preferring it's own simple pool (in radix_tree_preload). Examine the consequences of changing radix-tree to use mempool, and of changing mempool to be usable by

radix-tree. Provide patches to revert this design choice in radix-tree, or a pattern to explain this design choice.

4. Compare the radix-tree and idr implementations to see if one could be implemented using the other without sacrificing correctness, maintainability or performance. Provide either an explanation of why they should stay separate, or a patch to replace one with the other.

Linux kernel design patterns - part 3

In this final article we will be looking at just one design pattern. We started with the fine details of reference counting, zoomed out to look at whole data structures, and now move to the even larger perspective of designing subsystems. Like every pattern, this pattern needs a name, and our working title is "midlayer mistake". This makes it sounds more like an anti-pattern, as it appears to describe something that should be avoided. While that is valid, it is also very strongly a pattern with firm prescriptive guides. When you start seeing a "midlayer" you know you are in the target area for this pattern and it is time to see if this pattern applies and wants to guide you in a different direction.

In the Linux world, the term "midlayer" seems (in your author's mind and also in Google's cache) most strongly related to SCSI. The "scsi midlayer" went through a bad patch quite some years ago, and there was plenty of debate on the relevant lists as to why it failed to do what was needed. It was watching those discussions that provided the germ from which this pattern slowly took form.

The term "midlayer" clearly implies a "top layer" and a "bottom layer". In this context, the "top" layer is a suite of code that applies to lots of related subsystems. This might be the POSIX system call layer which supports all system calls, the block layer which supports all block devices, or the VFS which supports all filesystems. The block layer would be the top layer in the "scsi midlayer" example. The "bottom" layer then is a particular implementation of some service. It might be a specific system call, or the driver

for a specific piece of hardware or a specific filesystem. Drivers for different SCSI controllers fill the bottom layer to the SCSI midlayer. Brief reflection on the list of examples shows that which position a piece of code takes is largely a matter of perspective. To the VFS, a given filesystem is part of the bottom layer. To a block device, the same filesystem is part of the top layer.

A midlayer sits between the top and bottom layers. It receives requests from the top layer, performs some processing common to the implementations in the bottom layer, and then passes the preprocessed requests - presumably now much simpler and domain-specific - down to the relevant driver. This provides uniformity of implementation, code sharing, and greatly simplifies that task of implementing a bottom-layer driver.

The core thesis of the "midlayer mistake" is that midlayers are bad and should not exist. That common functionality which it is so tempting to put in a midlayer should instead be provided as library routines which can used, augmented, or ignored by each bottom level driver independently. Thus every subsystem that supports multiple implementations (or drivers) should provide a very thin top layer which calls directly into the bottom layer drivers, and a rich library of support code that eases the implementation of those drivers. This library is available to, but not forced upon, those drivers.

To try to illuminate this pattern, we will explore three different subsystems and see how the pattern specifically applies to them - the block layer, the VFS, and the 'md' raid layer (i.e. the areas your author is most familiar with).

Block Layer

The bulk of the work done by the block layer is to take 'read' and 'write' requests for block devices and send them off to the appropriate bottom level device driver. Sounds simple enough. The interesting point is that block devices tend to involve rotating media, and rotating media benefits from having consecutive requests being close together in address space. This helps reduce seek time. Even non-rotating media can benefit from

having requests to adjacent addresses be adjacent in time so they can be combined into a smaller number of large requests. So, many block devices can benefit from having all requests pass through an elevator algorithm to sort them by address and so make better use of the device.

It is very tempting to implement this elevator algorithm in a 'midlayer'. i.e. a layer just under the top layer. This is exactly what Linux did back in the days of 2.2 kernels and earlier. Requests came in to ll_rw_block() (the top layer) which performed basic sanity checks and initialized some internal-use fields of the structure, and then passed the request to make_request() - the heart of the elevator. Not quite every request went to make_request() though. A special exception was made for "md" devices. Those requests were passed to md_make_request() which did something completely different as is appropriate for a RAID device.

Here we see the first reason to dislike midlayers - they encourage special cases. When writing a midlayer it is impossible to foresee every possible need that a bottom level driver might have, so it is impossible to allow for them all in the midlayer. The midlayer could conceivably be redesigned every time a new requirement came along, but that is unlikely to be an effective use of time. Instead, special cases tend to grow.

Today's block layer is, in many ways, similar to the way it was back then with an elevator being very central. Of course lots of detail has changed and there is a lot more sophistication in the scheduling of IO requests. But there is still a strong family resemblance. One important difference (for our purposes) is the existence of the function blk_queue_make_request() which every block device driver must call, either directly or indirectly via a call to blk_init_queue(). This registers a function, similar to make_request() or md_make_request() from 2.2, which should be called to handle each IO request.

This one little addition effectively turns the elevator from a midlayer which is imposed on every device into a library function which is available for devices to call upon. This was a significant step in the right direction. It is now easy for drivers to choose not to use the elevator. All virtual drivers (md, dm, loop, drbd, etc.) do this, and even some drivers for physical hardware (e.g. umem) provide their own make_request_fn().

While the elevator has made a firm break from being a mid-layer, it still retains the appearance of a midlayer in a number of ways. One example is the struct request_queue structure (defined in <linux/blkdev.h>). This structure is really part of the block layer. It contains fields that are fundamental parts of the block interface, such as the make_request_fn() function pointer that we have already mentioned. However many other fields are specific to the elevator code, such as elevator (which chooses among several IO schedulers) and last_merge (which is used to speed lookups in the current queue). While the elevator can place fields in struct request_queue, all other code must make use of the queuedata pointer to store a secondary data structure.

This arrangement is another tell-tale for a midlayer. When a primary data structure contains a pointer to a subordinate data structure, we probably have a midlayer managing that primary data structure. A better arrangement is to use the "embedded anchor" pattern from the previous article in this series. The bottom level driver should allocate its own data structure which contains the data structure (or data structures) used by the libraries embedded within it. struct inode is a good example of this approach, though with slightly different detail. In 2.2, struct inode contained a union of the filesystem-specific data structure for each filesystem, plus a pointer (generic_ip) for another filesystem to use. In the 2.6 kernel, struct inode is normally embedded inside a filesystem-specific inode structure (though there is still an i_private pointer which seems unnecessary).

One last tell-tale sign of a midlayer, which we can still see hints of in the elevator, is the tendency to group unrelated code together. The library design will naturally provide

separate functionality as separate functions and leave it to the bottom level driver to call whatever it needs. The midlayer will simply call everything that might be needed.

If we look at __make_request() (the 2.6 entry point for the elevator), we see an early call to blk_queue_bounce(). This provides support for hardware that cannot access the entire address space when using DMA to move data between system memory and the device. To support such cases, data sometimes needs to be copied into more accessible memory before being transferred to the device, or to be copied from that memory after being transferred from the device. This functionality is quite independent of the elevator, yet it is being imposed on all users of the elevator.

So we see in the block layer, and its relationship with the elevator a subsystem which was once implemented as a midlayer, but has taken a positive step away from being a midlayer by making the elevator clearly optional. It still contains traces of its heritage which have served as a useful introduction to the key identifiers of a midlayer: code being imposed on lower layer, special cases in that code, data structures storing pointers to subordinate data structures, and unrelated code being called by the one support function.

With this picture in mind, let us move on.

The VFS

The VFS (or Virtual File System) is a rich area to explore to learn about midlayers and their alternatives. This is because there is a lot of variety in filesystems, a lot of useful services that they can make use of, and a lot of work has been done to make it all work together effectively and efficiently. The top layer of the VFS is largely contained in the vfs_ function calls which provide the entry points to the VFS. These are called by the various sys_ functions that implement system calls, by nfsd which does a lot of file system access without using system calls, and from a few other parts of the kernel that need to deal with files.

The vfs_ functions fairly quickly call directly in to the filesystem in question through one of a number of _operations structures which contain a list of function pointers. There are inode_operations, file_operations, super_operations etc, depending on what sort of object is being manipulated. This is exactly the model that the "midlayer mistake" pattern advocates. A thin top layer calls directly into the bottom layer which will, as we shall see, make heavy use of library functions to perform its task.

We will explore and contrast two different sets of services provided to filesystems, the page cache and the directory entry cache.

The page cache

Filesystems generally want to make use of read-ahead and write-behind. When possible, data should be read from storage before it is needed so that, when it is needed, it is already available, and once it has been read, it is good to keep it around in case, as is fairly common, it is needed again. Similarly, there are benefits from delaying writes a little, so that throughput to the device can be evened out and applications don't need to wait for writeout to complete. Both of these features are provided by the page cache, which is largely implemented by mm/filemap.c and mm/page-writeback.c.

In its simplest form a filesystem provides the page cache with an object called an address_space which has, in its address_space_operations, routines to read and write a single page. The page cache then provides operations that can be used as file_operations to provide the abstraction of a file that must be provided to the VFS top layer. If you look at the file_operations for a regular file in ext3, we see:

```
const struct file_operations ext3_file_operations = {
        .llseek         = generic_file_llseek,
        .read           = do_sync_read,
        .write          = do_sync_write,
        .aio_read       = generic_file_aio_read,
```

```
        .aio_write      = ext3_file_write,
        .unlocked_ioctl = ext3_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl   = ext3_compat_ioctl,
#endif
        .mmap           = generic_file_mmap,
        .open           = generic_file_open,
        .release  = ext3_release_file,
        .fsync          = ext3_sync_file,
        .splice_read    = generic_file_splice_read,
        .splice_write   = generic_file_splice_write,
};
```

Eight of the thirteen operations are generic functions provided by the page cache. Of the remaining five, the two ioctl() operations and the release() operation require implementations specific to the filesystem; ext3_file_write() and ext3_sync_file are moderately sized wrappers around generic functions provided by the page cache. This is the epitome of good subsystem design according to our pattern. The page cache is a well defined library which can be used largely as it stands (as when reading from an ext3 file), allows the filesystem to add functionality around various entry points (like ext3_file_write()) and can be simply ignored altogether when not relevant (as with sysfs or procfs).

Even here there is a small element of a midlayer imposing on the bottom layer as the generic struct inode contains a struct address_space which is only used by the page cache and is irrelevant to non-page-cache filesystems. This small deviation from the pattern could be justified by the simplicity it provides, as the vast majority of filesystems do use the page cache.

The directory entry cache (dcache)

Like the page cache, the dcache provides an important service for a filesystem. File names are often accessed multiple times, much more so than the contents of file. So caching them is vital, and having a well designed and efficient directory entry cache is a big part of having efficient access to all filesystem objects. The dcache has one very important difference from the page cache though: it is not optional. It is imposed upon every filesystem and is effectively a "midlayer." Understanding why this is, and whether it is a good thing, is an important part of understanding the value and applicability of this design pattern.

One of the arguments in favor of an imposed dcache is that there are some interesting races related to directory renames; these races are easy to fail to handle properly. Rather than have every filesystem potentially getting these wrong, they can be solved once and for all in the dcache. The classic example is if /a/x is renamed to /b/c/x at the same time that /b/c is renamed to /a/x/c. If these both succeed, then 'c' and 'x' will contain each other and be disconnected from the rest of the directory tree, which is a situation we would not want.

Protecting against this sort of race is not possible if we only cache directory entries at a per-directory level. The common caching code needs to at least be able to see a whole filesystem to be able to detect such a possible loop-causing race. So maintaining a directory cache on a per-filesystem basis is clearly a good idea, and strongly encouraging local filesystems to use it is very sensible, but whether forcing it on all filesystems is a good choice is less clear.

Network filesystems do not benefit from the loop detection that the dcache can provide as all of that must be done on the server anyway. "Virtual" filesystems such as sysfs, procfs, ptyfs don't particularly need a cache at all as all the file names are in memory permanently. Whether a dcache hurts these filesystems is not easy to tell as we don't have a complete and optimized implementation that does not depend on the dcache to compare with.

Of the key identifiers for a midlayer that were discussed above, the one that most clearly points to a cost is the fact that midlayers tend to grow special case code. So it should be useful to examine the dcache to see if it has suffered from this.

The first special cases that we find in the dcache are among the flags stored in d_flags. Two of these flags
are DCACHE_AUTOFS_PENDING and DCACHE_NFSFS_RENAMED. Each is specific to just one filesystem. The AUTOFS flag appears to only be used internally to autofs, so this isn't really a special case in the dcache. However the NFS flag is used to guide decisions made in common dcache code in a couple of places, so it clearly is a special case, though not necessarily a very costly one.

Another place to look for special case code is when a function pointer in an _operations structure is allowed to be NULL, and the NULL is interpreted as implying some specific action (rather than no action at all). This happens when a new operation is added to support some special-case, and NULL is left to mean the 'default' case. This is not always a bad thing, but it can be a warning signal.

In the dentry_operations structure there are several functions that can be NULL. d_revalidate() is an example which is quite harmless. It simply allows a filesystem to check if the entry is still valid and either update it or invalidate it. Filesystems that don't need this simply do nothing as having a function call to do nothing is pointless.

However, we also find d_hash() and d_compare(), which allow the filesystem to provide non-standard hash and compare functions to support, for example, case-insensitive file names. This does look a lot like a special case because the common code uses an explicit default if the pointer is NULL. A more uniform implementation would have every filesystem providing a non-NULL d_hash() and d_compare(), where many filesystems would choose the case-sensitive ones from a library.

It could easily be argued that doing this - forcing an extra function call for hash and compare on common filesystems - would be an undue performance cost, and this is true. But given that, why is it appropriate to impose such a performance cost on filesystems which follow a different standard?

A more library-like approach would have the VFS pass a path to the filesystem and allow it to do the lookup, either by calling in to a cache handler in a library, or by using library routines to pick out the name components and doing the lookups directly against its own stored file tree.

So the dcache is clearly a midlayer, and does have some warts as a result. Of all the midlayers in the kernel it probably best fits the observation above that they could "be redesigned every time a new requirement came along". The dcache does see constant improvement to meet the needs of new filesystems. Whether that is "an effective use of time" must be a debate for a different forum.

The MD/RAID layer

Our final example as we consider midlayers and libraries, is the md driver which supports various software-RAID implementations and related code. md is interesting because it has a mixture of midlayer-like features and library-like features and as such is a bit of a mess.

The "ideal" design for the md driver is (according to the "midlayer mistake" pattern) to provide a bunch of useful library routines which independent RAID-level modules would use. So, for example, RAID1 would be a standalone driver which might use some library support for maintaining spares, performing resync, and reading metadata. RAID0 would be a separate driver which use the same code to read metadata, but which has no use for the spares management or resync code.

Unfortunately that is not how it works. One of the reasons for this relates to the way the block layer formerly managed major and minor device numbers. It is all much more flexible today, but in the past a different major number implied a unique device driver and a unique partitioning scheme for minor numbers. Major numbers were a limited resource, and having a separate major for RAID0, RAID1, and RAID5 etc would have been wasteful. So just one number was allocated (9) and one driver had to be responsible for all RAID levels. This necessity undoubtedly created the mindset that a midlayer to handle all RAID levels was the right thing to do, and it persisted.

Some small steps have been made towards more of a library focus, but they are small and inconclusive. One simple example is the md_check_recovery() function. This is a library function in the sense that a particular RAID level implementation needs to explicitly call it or it doesn't get used. However, it performs several unrelated tasks such as updating the metadata, flushing the write-intent-bitmap, removing devices which have failed, and (surprisingly) checking if recovery is needed. As such it is a little like part of a mid-layer in that it imposes that a number of unrelated tasks are combined together.

Perhaps a better example is md_register_thread() and friends. Some md arrays need to have a kernel thread running to provide some support (such as scheduling read requests to different drives after a failure). md.c provides library routines md_register_thread() and md_unregister_thread(), which can be called by the personality as required. This is all good. However md takes it upon itself to choose to call md_unregister_thread() at times rather than leaving that up to the particular RAID level driver. This is a clear violation of the library approach. While this is not causing any actual problems at the moment, it is exactly the sort of thing that could require the addition of special cases later.

It has often been said that md and dm should be unified in some way (though it is less often that the practical issues of what this actually means are considered). Both md and dm suffer from having a distinct midlayer that effectively keeps them separate. A full

understanding of the fact that this midlayer is a mistake, and moving to replace it with an effective library structure is likely to be an important first step towards any sort of unification.

Wrap up

This ends our exploration of midlayers and libraries in the kernel -- except maybe to note that more recent additions include such things as libfs, which provides support for virtual filesystems, and libata, which provides support for SATA drives. These show that the tendency away from midlayers is not only on the wishlist of your author but is present in existing code.

Hopefully it has resulted in an understanding of the issues behind the "midlayer mistake" pattern and the benefits of following the library approach.

Here too ends our little series on design patterns in the Linux kernel. There are doubtlessly many more that could be usefully extracted, named, and illuminated with examples. But they will have to await another day.

Once compiled, such a collection would provide invaluable insight on how to build kernel code both effectively and uniformly. This would be useful in understanding how current code works (or why it doesn't), in making choices when pursuing new development, or when commenting on design during the review process, and would generally improve visibility at this design level of kernel construction. Hopefully this could lead, in the long term, to an increase in general quality.

For now, as a contribution to that process, here is a quick summary of the Patterns we have found.

- **kref**: Reference counting when the object is destroyed with the last external reference

- **kcref**: Reference counting when the object can persist after the last external reference is dropped
- **plain ref**: Reference counting when object lifetime is subordinate to another object.
- **biased-reference**: An anti-pattern involving adding a bias to a reference counter to store one bit of information.
- **Embedded Anchor**: This is very useful for lists, and can be generalized as can be seen if you explore kobjects.
- **Broad Interfaces**: This reminds us that trying to squeeze lots of use-cases in to one function call is not necessary - just provide lots of function calls (with helpful and (hopefully) consistent names).
- **Tool Box**: Sometimes it is best not to provide a complete solution for a generic service, but rather to provide a suite of tools that can be used to build custom solutions.
- **Caller Locks**: When there is any doubt, choose to have the caller take locks rather than the callee. This puts more control in that hands of the client of a function.
- **Preallocate Outside Locks**: This is in some ways fairly obvious. But it is very widely used within the kernel, so stating it explicitly is a good idea.
- **Midlayer Mistake**: When services need to be provided to a number of low-level drivers, provide them with a library rather than imposing them with a midlayer.

Exercises

1. Examine the "blkdev_ioctl()" interface to the block layer from the perspective of whether it is more like a midlayer or a library. Compare the versions in 2.6.27 with 2.6.28. Discuss.
2. Choose one other subsystem such as networking, input, or sound, and examine it in the light of this pattern. Look for special cases, and imposed functionality.

Examine the history of the subsystem to see if there are signs of it moving away from, or towards, a "midlayer" approach.

3. Identify a design pattern which is specific to the Linux kernel but has not been covered in this series. Give it a name, and document it together with some examples and counter examples.