# Stacking Blocks
## Problems and Solutions with stacked block device drivers in Linux

Neil Brown
SuSE Labs, Novell Inc

July 31, 2007

## 1 Introduction

For most people, a block device is typically a rectangular prism of some sort. Most often it is a disk drive, whether SCSI or SATA, or increasingly some sort of Flash storage media where the emphasis on 'rectangular' is giving way to more rounded shapes.

Stacking of these devices might mean putting them in a rack of some sort, either one above the other, or in a row. And hopefully some magic behind the scenes will make them appear to be one big piece of storage instead of lots of smaller pieces.

It is this magic for stacking blocks that I will be discussing in this paper, though the precise meaning of both the words 'block' and 'stack' will slightly different to the usage above.

In Linux as in Unix, a 'block device' is a storage device that is most conveniently accessed in fixed sized units larger than one byte. The classic examples are magnetic disks, optical discs, and magnetic tape and the classic unit size is 512 bytes, though larger sizes have always been around and are becoming more common. So a 'block' is not the physical shape, but rather the required style of data access.

Building blocks are often stacked to gain greater heights with each block being supported by the blocks beneath it. Similarly with block devices, we stack them so that each is supported by the services available below it, and the whole stack together can reach greater heights of functionality, performance, capacity, or reliability (sometimes in combination). Unlike classic building blocks, the blocks at different levels are not usually the same. We may have — as depicted in Figure 1 — a collection of similar disk drives at the bottom, though with two small ones placed under a RAID0 device to match the size of the others. Then a RAID5 device may be placed on top to provide aggregation and some redundancy, then a Logical Volume manager may split this up into logical devices to be used for different purposes, one of which might have an encrypting block device placed atop it for added security.

The Linux kernel contains a module of code to support the use of block devices. Knows as the 'block layer' it is primarily an interface definition together with a bunch of useful routines — like IO schedulers and scatter/gather list mapping — that a block driver can use to help with it's task.

Block devices that are designed to stack are unique in that they are both a producer and a consumer of the interface defined by the block layer. That places them in an position to see clearly some strengths and weaknesses.

The remainder of this paper will look a the history of the interface defined by the block layer with particular reference to how that interface affects stacking devices, at the current difficulties faced by various stacking devices in their interactions with the block layer and with each other, and at recent work and future possibilities to make sure all the children are happily playing with their blocks.

## 2 History

The current kernel development model de-emphasises the strong distinction between a 'stable' branch and a 'development' branch in favour of a graded range of branches (-mm, -linus, -stable, -vendor). While this is in many ways a good thing, it makes it harder for the historian to make sweeping

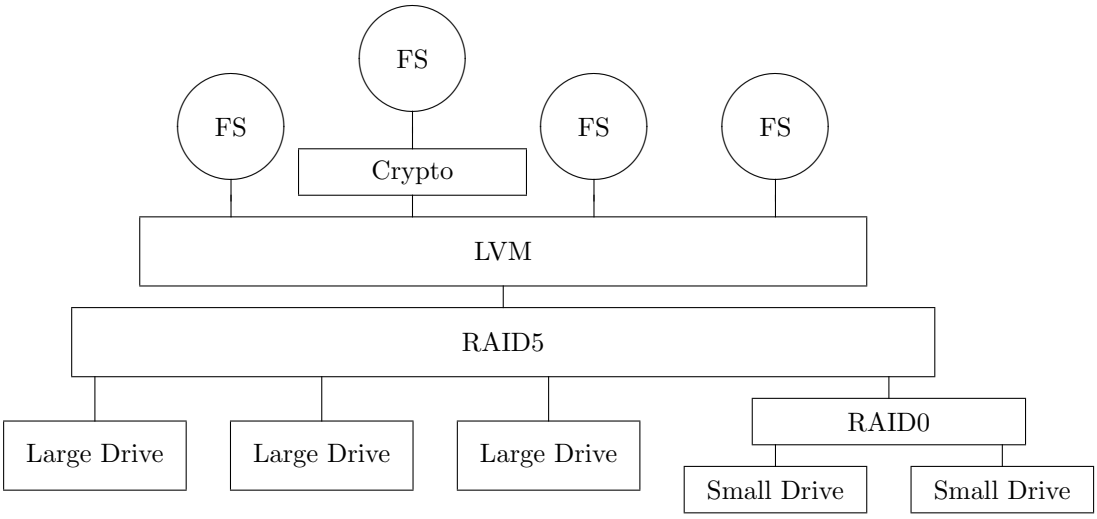


Figure 1: Sample Stack of Block Devices

statements about different epochs of kernel development. Fortunately this new model is still new and much of the history of the block layer can be found in earlier 'stable' branches, so we will be making good use of some sweeping generalisations.

## 2.1 Beginnings — The 2.2 days

It has been said that to every difficult and complex problem there is a solution that is simple, elegant, and wrong. This could well describe the block layer during the 2.2 series of Linux kernels (and earlier).

At this time there was a 'buffer cache' made of in-memory buffers that stored an image of a block of data on disk. Each buffer in the cache had a 'buffer_head' structure that defined the address and state of the buffer and included various other house keeping information.

The block layer interface was simple. You passed it one of these buffer heads and said 'read' or 'write', and it did.

Buffers were created per-device and for each device there was a fixed block size ranging from one sector (512 bytes) to one page (e.g. 4096 bytes). All buffers for a given device had the same size and were aligned on that size. The buffer size of a device could be changed, but that required flushing all buffers for that device from the cache and starting afresh.

A buffer head stored both a device number and block number, as well as a 'real device' and 'real sector' number, thus allowing non-trivial mapping between block number and storage locations.

This all made life of a stacking driver quite easy:

- Alignment was guaranteed, so a striping driver just needed to set the chunk size to be at least one page, and there would be no alignment problems.
- Block mapping was easy – just update'real_device' and 'real_sector' and pass the buffer_head back to the block layer
- As the buffer cache was physically addressed, it was even possible for stacking devices to flush blocks out early to improve efficiency. E.g. when RAID5 needed to write, it could look in the buffer cache for other blocks in the same stripe, and write them all at once.

However life was not quite so good for other members of the ecosystem. A physically addressed cache, while simple, is wrong for file-systems as they need to use their own internal indexing to find memory buffers. Such indexing is probably designed to make best use of disk drive capacities and speed, and is not ideal for in-memory accesses. As an extreme example: for ext2 to perform **fsync**

against a large file it would have to look at all the index blocks, find the physical addresses of all the data blocks, find those data blocks in the cache (if they were there) and then check if they were dirty and — if so – flush them to disk. For a very large file with only a few dirty blocks at the end, this was very inefficient.

Fixed sized, fixed alignment buffers were not much good for anyone (except stacking devices). More sophisticated filesystems like XFS really wanted different block sizes for different usages (journal versus data) and as drives got larger and faster, the overheads of passing data one small block at a time became significant.

So something had to change.

## 2.2 Transition – The 2.4 series

The 2.4 time frame could be seen as the coming-of-age of stacking block devices. **md** has been largely maintained out as an out-of-tree patch during 2.2 with only minimal **linear** and **RAID0** functionality in mainline. With 2.4, **md** had full RAID1 and RAID5 functionality in mainline. Similarly LVM made it into the mainline kernel at this time. Both were a fair way from being mature, but they becoming more widely accepted and used.

One of the big changes to arrive with 2.4 was the page cache. Where the buffer cache was indexed by the address of data on the block device, the page cache is indexed by the address of data in a given file. While this did not affect the block layer directly, it was the beginning of the end for buffer_heads and so their relationship with the block layer was surely soon to change.

With buffer_heads no longer being tied to the primary disk caching mechanism, the rules on strict uniformity of size and alignment of buffer_heads began to be relaxed. It is probably more accurate to say that the rules had never been firmly stated and file-system implementers discovered that using different sizes would work, so they did. As mentioned eariler, XFS (depending on configuration) can use different block sizes for different sorts of data. This caused problems for **md**/RAID5 which was designed to expect a uniform block size and used that size internally in its caching mechanism. Every time XFS appeared to change the block size, RAID5 had to flush its cache and try again, which wasn't ideal for performance.

Also as mentioned previously, if a stacked block device simply wanted to remap each block to a different location, it simply updated **b_rdev** and **b_rsector** and passed the buffer_head back to the block layer. The block layer had direct support for this in that the block device could indicate that was all it wanted to do, and the block layer would accept the task of looping around, re-interpreting the device number, and re-submitting the request. This had the pleasant consequence that multiple stacked block devices would not add to the usage of the kernel stack space. However with the gradual maturing of functionality in **md** and lvm, the usefulness of this optimisation faded away. While RAID0 or simple LVM can be implemented by a simple block remapping, other levels like RAID5, and more sophisticated LVM functions like encryption cannot. So though 2.4 and into 2.6, the potential for excessive stack usage grew and, with the possibility of 4K stacks, became a real problem.

And still, requests were being passed down to device drives one block at a time. Going down and back up a stack of devices for every block is not as efficient as gathering some related blocks together and passing them down the stack all at once. This even affects the very shallow stack of a single SCSI device. So again, something had to change.

## 2.3 2.6. Maturity?

The development in 2.5 and the arrival of 2.6 saw a major shake up in the block layer as fixed (or nearly fixed) sized buffer_heads were replaced with the variable sized 'bio' structure. A 'bio' can list a number of pages, with differing offsets and sizes to be used for each page. There are no fixed size or alignment restrictions on bios, which is good for filesystems that like flexibility, and good for disk drives that can take very large requests, but not so good for stacking devices that have to start worrying about alignment of requests.

It isn't quite true to say that there are no restrictions. Different drives and controllers have assorted different upper limits on total sectors, and sizes of scatter-gather lists, and how IOMMUs can be used to rearrange pages into IO buffers. So a sophisticated, but limited, set of rules were

created for how big a bio could get, both in terms of number of sectors and the number of contiguous memory ranges (segments).

This was fine for block device that look like blocks (i.e. disk drives) but wasn't quite enough for stacked device which had extra global alignment restrictions (e.g. RAID0 wants each request to map to just one target device). So the interface was extended to allow the block device to have a chance to veto the addition of each block to a bio. This would allow a RAID0 device to stop a bio from increasing past the size that would fit in a single chunk — thus allowing each bio to be directly mapped to a single underlying device ... almost.

The small hole in this is that if the first block to be added to a bio crossed a chunk boundary — if the first bytes should be mapped to one device while the last bytes should be mapped to anther — there was no way for the stacked RAID0 to prevent this. Thus it needed to be able to split small, single-page requests. Fortunately splitting a single page bio into two smaller single-page bios is not very difficult, and several drivers in 2.6 do this.

However the difficulties don't end there. When a stacked device is asked if it wants to veto adding a block to a bio, it should really proceed to check with the device, or devices, that it will ultimately send the request to, to see if it might want to veto the addition too. There are a few problems with this, a significant one being that it tends to negate one of the main purposes for introducing the bio structure to the block layer.

That purpose, as mentioned eariler, was to allow large requests to be built up without having to bounce back and forth (or is that 'up' and 'down' on the stack) between filesystem code and device driver code. With the veto call-down, the filesystem code needs to call into the device driver for every block, which goes some way to negating the advantages of bios. It is true that this only affects stacked drives, so direct devices (disk drives) still benefit. However nobody likes being a second-class citizen, not even a device driver.

There is also the problem of device reconfiguration. A stacked device which is built on other devices could change which actual devices are used to store data from time to time, whether due to a failed drive in a RAID being reconstructed onto spare, or because LVM was told to migrate data from one place to another. As there is a delay (short, but non-zero) between the filesystem asking for a veto, and the filesystem actually submitting a request, the device that ends up receiving the request could be different from the one that was asked for a veto. This could be fixed with refcounting and locking but that just adds more overhead to a very common kernel code path.

As a consequence of this most, if not all, stacking drivers do not bother to call the veto routine of drivers that are lower in the stack. They simply check if such a routine exist and — if it does — they set their own maximum block size to 1 page so that no large bios ever attempt their way down the stack. This works, but is clearly not ideal.

# 3 Status

So we come to present day Linux, by which we mean release 2.6.20 or there abouts. There have been ups and downs in the life of a stacked block device drives and at this stage there are two main issues.

## 3.1 Kernel Stack Usage

The first is kernel-stack usage. The calling convention requires the driver at each level in the stack to make a subroutine call into the driver and the next lower level to pass requests around in all but the most simple (block remapping) cases. This make is difficult to put firm bounds on stack usage without unfairly curtailing the allowed configurations of stacked block devices.

There are 4 main block device interfaces that can cause recursive calls all the way down the device stack. Two of those: `unplug_fn` and `issue_flush_fn` have not been addressed. They are extremely simple functions and so do not add to stack usage very much and have not been an issue in practice. Nevertheless they should be addressed at some stage.

`merge_bvec_fn`, which is the *veto* function mentioned above, will be discussed in the next subsection.

That leaves the main block device interface: `make_request_fn`. This is the function that is handed a bio and is expected to schedule the IO. Often the `make_request_fn` in a stacked block

device will call the `make_request_fn` in one or more lower devices. As these functions can be quite complex, a lot of kernel stack can be used.

Fortunately there is an easy way around this. The `make_request_fn` does not return any value. It is asynchronous as is expected to schedule the request and signal the completion later via a callback. Also, it has only one argument (well, technically two but the `request_queue` argument can be deduced from the `bio` argument) and this argument contains a pointer field that is unused.

Combining these fortunate facts together we discover that a recursive call the `make_request_fn` can be handled by simply placing the `bio` on a linked list until the parent request completes. Thus instead of the various `make_request_fn` calls being stacked, they happen in sequence, a bit like optimising a tail recursion.

This does put an added requirement of device drives that the `make_request_fn` must not wait for another `make_request_fn` to finish. There were a few cases that did wait such as some cases in **md** where a super-block update was required before a write could continue. Moving the management of that out into a separate thread not only avoided the problem, but made the code somewhat cleaner too.

The `make_request_fn` interface is never called directly, but instead is called though the `generic_make_request` function. This means that the handling of the list of delayed requests can all be done in one place. So with 2.6.23 came a wrapper for `generic_make_request` which ensured that recursive calls would always be serialised. This should go a long way to making an end to kernel stack problems caused by stacked devices.

## 3.2 Request size restrictions

The other big problem is managing the various restrictions on request sizes. Each layer is a stack of block devices may impose it own restrictions on the size of a request. If we want to be able to build large requests and have them pass down through the multiple layers without needing to be broken into multiple smaller requests, then the top level that assembles the requests needs to know about all those restrictions. While that is quite manageable for simple disk drive devices, there are a number of problems with doing it generally for a stack of devices.

The obvious solution would appear to be to accept that drives are different layer will need to split requests. Some devices such as RAID0 already need to do this when they receive single-page requests (which cannot be vetoed) that cross a chunk boundary. Maybe this should be made the common case rather than a rare case.

This has one immediate advantage that the restrictions that a device imposes on a request need only be known to that device. There no longer needs to be a protocol whereby the device can advise the filesystem about the permissible sizes. Filesystems can generate arbitrarily sized requests, and the device (at each level in the stack) can split the request if needed.

This also has the benefit that the `merge_bvec_fn` will no longer be needed, so any kernel stack issues it might cause will disappear.

But there is a reason why this wasn't done long ago. Splitting is hard.

A bio structure consists of two separate parts. The main `struct bio` and a subsidiary array of `struct bio_vec` which lists the individual pages with their offsets and lengths. When splitting a bio, it would be ideal to share the `bio_vec`s between the two new bios. Each bio could identify a start location in the array of `bio_vec`s and a total size to use. This would make splitting a bio quite easy. However sharing the bio_vec is not currently possible.

Some device drivers are written to process an IO request in stages. The device addressing commands for the whole request might be passed to the device, but then the data needs to be passed a little at a time. To keep track of how much data has been passed, the bio (and previously the buffer_head) would be modified to reflect the fact that some of the data had been processed and that the next piece of data to be processed is further along. These modification involve updating the offset and length fields in the `bio_vec` array. If two bios are sharing the same array, they could easily trip over each other.

So to make it possible to split requests effectively, we need to first make the `bio_vec` array immutable. Ideally we want to make substantial parts of the `struct bio` immutable too as this makes it possible to share them among different **struct request** structures at the lowest level of a device stack. Conceptually, it is quite easy to make these fields immutable. In each place where they are changed, there are other data structures available which could equally well record the important

changes. However in practice it is quite a major reshuffle to assumptions made by a lot of the block-device code.

At the time of writing, a 35-patch set is under development and testing which does exactly this. It re-educates much of the block layer so that `struct bio_vec` and several fields of `struct bio` remain unchanged by request processing and leverages this to allow various block device to split requests as and when required. This greatly simplifies the request submission code in several places, and places the control where it is needed.

More review and testing is needed before this will get close to mainline (after all, this is people's data we are risking here).

# 4 ... and beyond

Once the core issues discussed above are dealt with and out of the way there will be plenty of other issues to look at. Apart from obvious goals like reliability, functionality, and efficiency, the most commonly mentioned issue surrounding stacked block devices is uniformity. We often hear "when is someone going to merge **dm** and **md**", though strangely we never seem to hear "I have some great code here to make **dm** and **md** work better together".

Before you get your hopes up, I'm not here to tell you what the answer is, at least not completely. But I thought it might help to explore and expose some facets of the problem. In particular, I am going to steer clear of suggesting why things are the way they are. Peoples motivations are very hard to judge, and change over time. So beyond the obvious statement 'Developers are only human', I'll leave any discussion of causality to sociologists.

Firstly I'll present a brief overview of different stacked block devices, then look at some of the crucial differences, and finally wave my hands about where I think we should look to go forward.

## 4.1 A taxonomy of stacking

The stacking block device drivers that are currently in the mainline kernel include:

**loop** The **loop** device is possibly the simplest. It takes one input and provides one output, and may provide some translation on the way though. The output is, obviously, a block device.

The translation can involve very simple partitioning — adding an offset and limiting to a size — and also encryption or decryption of the data.

The input can be either another block device or a regular file. The possibility of using a regular file is unique to the **loop** driver and it creates some very convenient functionality such as being able to mount a filesystem image that is stored in another filesystem (interesting things can be done with images stored in sparse files). It also creates some implementation difficulties such that **loop** has been substantially rewritten a number of times. In it's current version, a separate kernel thread is used to handle all IO to files, thus avoiding some awkward ordering issues if a filesystem tried to write, though a **loop** device, directly into another filesystem.

**md** The **md** module — which might stand for 'Multiple devices' or 'Meta-Disk' or some other combination of words — allows devices to be joined in the well known RAID levels, 0, 1, 4, 5 and 6 as well as a version of RAID10 which is a combination of RAID0 and RAID1 and simple device catenation (aka 'linear').

It also has limited support for 'multipath' devices where there are two or more paths to the one physical device, and the driver needs to load-balance or fail-over between the paths. This support is not only limited, but is not being developed and should probably be deprecated. **dm** has much better multipath support.

Finally it has a 'faulty' module for simulating device failures for testing.

The RAID1 and RAID4/5/6 modules have quite rich feature sets including write-intent logging to avoid costly resyncs after a crash and online reshaping of the arrays, though there is plenty more that is wanted here.

**dm** The 'Device Mapper' is the successor of the LVM module that first introduced Logical Volume Management. A major redesign of LVM produced LVM2 which moved a lot of code into user-space, and left **dm** as the kernel side of the system. The Device Mapper supports linear and

striping modes similar to linear and raid0 in **md**. It supports a cryptographic module similar to that in the **loop** driver (though different in detail). It also has quite mature multipath support which is much more advanced than that which is in **md**. It provides for taking a read-only snapshot of an active device and for migrating data from one underlying device to another.

And on top of all of these it provides a partitioning service whereby a logical volume can be created from multiple sections of various lower level devices.

Thus it provides a very rich collection of services to present available physical storage as virtual storage in various ways. It duplicates some parts of **loop**, and some parts of **md**, and with the **dm-loop** and **dm-raid5** modules that have been mentioned as being under development, it may eventually duplicate all the functionality of those modules.

**pktcdvd** This is a very different sort of stacking device. It is designed specifically to stack on top of CD and DVD drives. These drive have special requirements for writing in that writes must be sized and aligned to 'packets' which are larger than regular blocks, and there are extra flushing requirements between a write and a subsequent read.

The **pktcdvd** driver takes a single device with these special requirements and presents it as a regular device to which reads and write (with a basic block size of 2K) can be sent freely.

To this we will probably soon be able to add DRDB which is being proposed for submission. It has similar functionality to RAID1, but is designed to work in a cluster with both devices and clients being potentially on other nodes in a cluster.

## 4.2 Differences

Probably the biggest difference between the various stacking drivers — particularly between those that have similar functionality — is the way they communicate with user-space applications. How the devices are created, modified, and queried.

General configuration and querying happens through various 'ioctl' commands. Such interfaces are notoriously 'ad hoc' and these drivers are no exception. So while they often do similar things, they require them to be done in different ways. This makes it awkward to have a single tool for working with all the different sorts of stacked devices.

Creation of a device is very different as well. For **loop** and **md**, the act of opening a device special file with the right major/minor numbers creates the device, though with no configuration. Ioctl commands are then sent to set it up. For **pktcdvd** and **dm**, there is a special purpose 'character' device which can be sent ioctl commands to create a new block device having first configured it. This too leads to some distinct differences in the general approach to using these devices.

Finally there is a need, with **dm** and **md** at least, to notify user-space of events, such as reporting that a device has failed. **dm** uses a netlink connection to send this information. **md** uses files in sysfs, and supports 'poll' or 'select' notifications on these files.

As can be seen, the differences are more incidental than fundamental. It is quite possible for **dm**, **md**, and **loop** devices to stack on top of each other and for a single user-space framework to be able to work with them all. However the incidental differences cause enough friction that this isn't always as enjoyable an experience as it could be.

## 4.3 Hand Waving

If we wanted to make working with stacking devices smoother, my personal belief is that we should make them — as much as possible — disappear. **md** and **dm**, as entities, need to fade away. The core functionality should be added to the common block layer, and we should then be able to see and work with individual drives for such things as 'file as device', 'crypto', 'raid1', 'raid5', 'striped', 'concat', 'multipath' etc.

Each different module may well share code in a common library, but they should be independent modules. They should do one job and do it well. So for example, a **loop** device that both mapped a file to a block device and performed encryption/decryption would be replaced by two separate modules, one for **loop**, one for **crypto**. There would need to be some sort of legacy support for the old interface for a while, but that should utilise the new simple devices.

Similarly **dm** would drop its internal 'dm_target' and instead of having a table of targets, it would have a table of block devices which may themselves any sort of stacked device.

Some experimentation would be needed to determine what common support would be needed in the block layer, and what new interfaces might be needed between block devices.

The one item of block-layer support that stands out for me is the ability to suspend and retarget a block device while it is open. This means that new IO requests would have to block, and a count of pending requests would need to be tracked so that it would be possible to wait until all requests were completed and the device was quiescent. Then, the device interface could be attached to a different module and requests could start flowing again. This would allow a lot of flexibility configuring and reconfiguring storage.

The one item of inter-device communication that stands out is a protocol for notifying and accepting changes to the size of a device. Both **dm** and **md** allow devices to change size, and need to take action when that happens. A clear protocol is needed so that the right actions happen in the right order. If this protocol were available to filesystems too (which it should be) they could automatically grow or shrink, just by changing the size of the device that contains them.

# 5 Conclusion

Stacking devices have not always enjoyed a first-class status in the ecosystem of devices in Linux, but it need not remain that way. With a little care, some forward planning, and a lot of work, we can have a play ground where blocks in stacks are as safe as those on the ground.