The case for struct bio
- or -
A road map for a stackable BIO subsystem in FreeBSD

Poul-Henning Kamp <phk@FreeBSD.org>

The FreeBSD Project

ABSTRACT

Historically, the only translation performed on I/O requests after they they left the file-system layer were logical sub disk implementation done in the device driver. No universal standard for how sub disks are configured and implemented exists, in fact pretty much every single platform and operating system have done it their own way. As FreeBSD migrates to other platforms it needs to understand these local conventions to be able to co-exist with other operating systems on the same disk.

Recently a number of technologies like RAID have expanded the concept of "a disk" a fair bit and while these technologies initially were implemented in separate hardware they increasingly migrate into the operating systems as standard functionality.

Both of these factors indicate the need for a structured approach to systematic "geometry manipulation" facilities in FreeBSD.

This paper contains the road-map for a stackable "BIO" system in FreeBSD, which will support these facilities.

1. The miseducation of `struct buf`.

To fully appreciate the topic, I include a little historic overview of struct buf, it is a most enlightening case of not exactly bit-rot but more appropriately design-rot.

In the beginning, which for this purpose extends until virtual memory is was introduced into UNIX, all disk I/O were done from or to a struct buf. In the 6th edition sources, as printed in Lions Book, struct buf looks like this:

```
struct buf
{
   int     b_flags;        /* see defines below */
   struct  buf *b_forw;    /* headed by devtab of b_dev */
   struct  buf *b_back;    /*  '  */
   struct  buf *av_forw;   /* position on free list, */
   struct  buf *av_back;   /*     if not BUSY*/
   int     b_dev;          /* major+minor device name */
   int     b_wcount;       /* transfer count (usu. words) */
   char    *b_addr;        /* low order core address */
   char    *b_xmem;        /* high order core address */
   char    *b_blkno;       /* block # on device */
   char    b_error;        /* returned after I/O */
   char    *b_resid;       /* words not transferred after
                                             error */
} buf[NBUF];
```

At this point in time, struct buf had only two functions: To act as a cache and to transport I/O operations to device drivers. For the purpose of this document, the cache functionality is uninteresting and will be ignored.

The I/O operations functionality consists of three parts:

- Where in Ram/Core is the data located (b_addr, b_xmem, b_wcount).
- Where on disk is the data located (b_dev, b_blkno)
- Request and result information (b_flags, b_error, b_resid)

In addition to this, the av_forw and av_back elements are used by the disk device drivers to put requests on a linked list. All in all the majority of struct buf is involved with the I/O aspect and only a few fields relate exclusively to the cache aspect.

If we step forward to the BSD 4.4-Lite-2 release, struct buf has grown a bit here or there:

```
struct buf {
        LIST_ENTRY(buf) b_hash;         /* Hash chain. */
        LIST_ENTRY(buf) b_vnbufs;       /* Buffer's associated vnode. */
        TAILQ_ENTRY(buf) b_freelist;    /* Free list position if not active. */
        struct  buf *b_actf, **b_actb;  /* Device driver queue when active. */
        struct  proc *b_proc;           /* Associated proc; NULL if kernel. */
        volatile long   b_flags;        /* B_* flags. */
        int     b_error;                /* Errno value. */
        long    b_bufsize;              /* Allocated buffer size. */
        long    b_bcount;               /* Valid bytes in buffer. */
        long    b_resid;                /* Remaining I/O. */
        dev_t   b_dev;                  /* Device associated with buffer. */
        struct {
                caddr_t b_addr;         /* Memory, superblocks, indirect etc. */
        } b_un;
        void    *b_saveaddr;            /* Original b_addr for physio. */
        daddr_t b_lblkno;               /* Logical block number. */
        daddr_t b_blkno;                /* Underlying physical block number. */
                                        /* Function to call upon completion. */
        void    (*b_iodone) __P((struct buf *));
        struct  vnode *b_vp;            /* Device vnode. */
        long    b_pfcent;               /* Center page when swapping cluster. */
                                        /* XXX pfcent should be int; overld. */
        int     b_dirtyoff;             /* Offset in buffer of dirty region. */
        int     b_dirtyend;             /* Offset of end of dirty region. */
        struct  ucred *b_rcred;         /* Read credentials reference. */
        struct  ucred *b_wcred;         /* Write credentials reference. */
        int     b_validoff;             /* Offset in buffer of valid region. */
        int     b_validend;             /* Offset of end of valid region. */
};
```

The main piece of action is the addition of vnodes, a VM system and a prototype LFS filesystem, all of which needed some handles on struct buf. Comparison will show that the I/O aspect of struct buf is in essence unchanged, the length field is now in bytes instead of words, the linked list the drivers can use has been renamed (b_actf, b_actb) and a b_iodone pointer for callback notification has been added but otherwise there is no change to the fields which represent the I/O aspect. All the new fields relate to the cache aspect, link buffers to the VM system, provide hacks for file-systems (b_lblkno) etc etc.

By the time we get to FreeBSD 3.0 more stuff has grown on struct buf:

```
struct buf {
        LIST_ENTRY(buf) b_hash;          /* Hash chain. */
        LIST_ENTRY(buf) b_vnbufs;        /* Buffer's associated vnode. */
        TAILQ_ENTRY(buf) b_freelist;     /* Free list position if not active. */
        TAILQ_ENTRY(buf) b_act;          /* Device driver queue when active. *new* */
        struct  proc *b_proc;            /* Associated proc; NULL if kernel. */
        long    b_flags;                 /* B_* flags. */
        unsigned short b_qindex;         /* buffer queue index */
        unsigned char b_usecount;        /* buffer use count */
        int     b_error;                 /* Errno value. */
        long    b_bufsize;               /* Allocated buffer size. */
        long    b_bcount;                /* Valid bytes in buffer. */
        long    b_resid;                 /* Remaining I/O. */
        dev_t   b_dev;                   /* Device associated with buffer. */
        caddr_t b_data;                  /* Memory, superblocks, indirect etc. */
        caddr_t b_kvabase;               /* base kva for buffer */
        int     b_kvasize;               /* size of kva for buffer */
        daddr_t b_lblkno;                /* Logical block number. */
        daddr_t b_blkno;                 /* Underlying physical block number. */
        off_t   b_offset;                /* Offset into file */
                                         /* Function to call upon completion. */
        void    (*b_iodone) __P((struct buf *));
                                         /* For nested b_iodone's. */
        struct  iodone_chain *b_iodone_chain;
        struct  vnode *b_vp;             /* Device vnode. */
        int     b_dirtyoff;              /* Offset in buffer of dirty region. */
        int     b_dirtyend;              /* Offset of end of dirty region. */
        struct  ucred *b_rcred;          /* Read credentials reference. */
        struct  ucred *b_wcred;          /* Write credentials reference. */
        int     b_validoff;              /* Offset in buffer of valid region. */
        int     b_validend;              /* Offset of end of valid region. */
        daddr_t b_pblkno;                /* physical block number */
        void    *b_saveaddr;             /* Original b_addr for physio. */
        caddr_t b_savekva;               /* saved kva for transfer while bouncing */
        void    *b_driver1;              /* for private use by the driver */
        void    *b_driver2;              /* for private use by the driver */
        void    *b_spc;
        union   cluster_info {
                TAILQ_HEAD(cluster_list_head, buf) cluster_head;
                TAILQ_ENTRY(buf) cluster_entry;
        } b_cluster;
        struct  vm_page *b_pages[btoc(MAXPHYS)];
        int             b_npages;
        struct  workhead b_dep;          /* List of filesystem dependencies. */
};
```

Still we find that the I/O aspect of struct buf is in essence unchanged. A couple of fields have been added which allows the driver to hang local data off the buf while working on it have been added (b_driver1, b_driver2) and a "physical block number" (b_pblkno) have been added.

This p_blkno is relevant, it has been added because the disklabel/slice code have been abstracted out of the device drivers, the filesystem ask for b_blkno, the slice/label code translates this into b_pblkno which the device driver operates on.

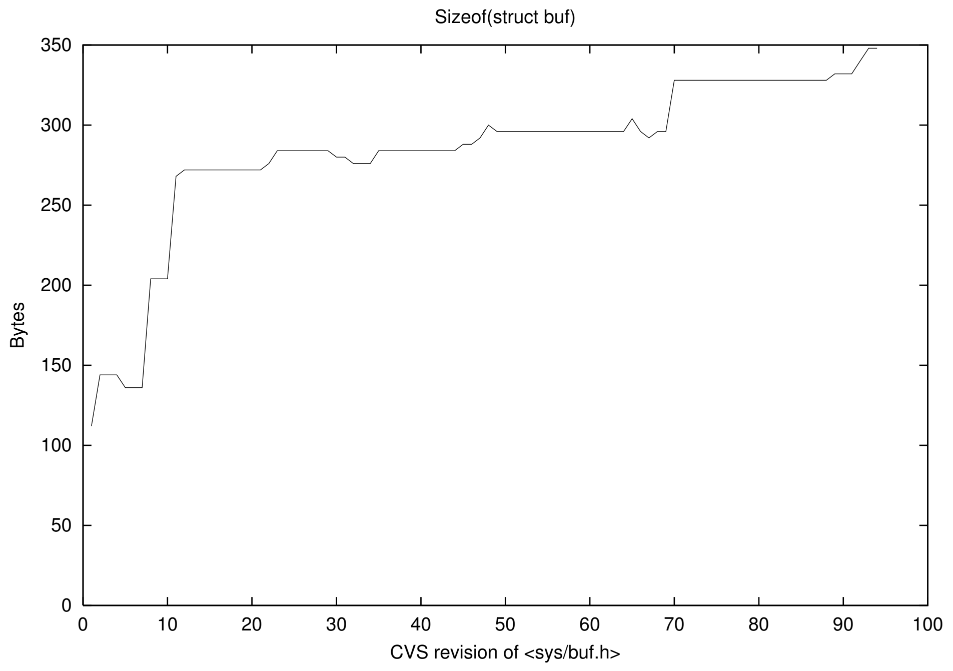
After this point some minor cleanups have happened, some unused fields have been removed etc but the I/O aspect of struct buf is still only a fraction of the entire structure: less than a quarter of the bytes in a struct buf are used for the I/O aspect and struct buf seems to continue to grow and grow.

Since version 6 as documented in Lions book, a three significant pieces of code have emerged which need to do non-trivial translations of the I/O request before it reaches the device drivers: CCD, slice/label and Vinum. They all basically do the same: they map I/O requests from a logical space to a physical space, and the mappings they perform can be 1:1 or 1:N. [1]

The 1:1 mapping of the slice/label code is rather trivial, and the addition of the b_pblkno field catered for the majority of the issues this resulted in, leaving but one: Reads or writes to the magic "disklabel" or equally magic "MBR" sectors on a disk must be caught, examined and in some cases modified before being passed on to the device driver. This need resulted in the addition of the b_iodone_chain field which adds a limited ability to stack I/O operations;

The 1:N mapping of CCD and Vinum are far more interesting. These two subsystems look like a device driver, but rather than drive some piece of hardware, they allocate new struct buf data structures populates these and pass them on to other device drivers.

Apart from it being inefficient to lug about a 348 bytes data structure when 80 bytes would have done, it also leads to significant code rot when programmers don't know what to do about the remaining fields or even worse: "borrow" a field or two for their own uses.



Conclusions:

- Struct buf is victim of chronic bloat.
- The I/O aspect of struct buf is practically constant and only about ¼ of the total bytes.
- Struct buf currently have several users, vinum, ccd and to limited extent diskslice/label, which need only the I/O aspect, not the vnode, caching or VM linkage.

The I/O aspect of struct buf should be put in a separate `struct bio`.

[1] It is interesting to note that Lions in his comments to the `rkaddr` routine (p. 16-2) writes *The code in this procedure incorporates a special feature for files which extend over more than one disk drive. This feature is described in the UPM Section "RK(IV)". Its usefulness seems to be restricted.* This more than hints at the presence already then of various hacks to stripe/span multiple devices.

2. Implications for future struct buf improvements

Concerns have been raised about the implications this separation will have for future work on struct buf, I will try to address these concerns here.

As the existence and popularity of vinum and ccd proves, there is a legitimate and valid requirement to be able to do I/O operations which are not initiated by a vnode or filesystem operation. In other words, an I/O request is a fully valid entity in its own right and should be treated like that.

Without doubt, the I/O request has to be tuned to fit the needs of struct buf users in the best possible way, and consequently any future changes in struct buf are likely to affect the I/O request semantics.

One particular change which has been proposed is to drop the present requirement that a struct buf be mapped contiguously into kernel address space. The argument goes that since many modern drivers use physical address DMA to transfer the data maintaining such a mapping is needless overhead.

Of course some drivers will still need to be able to access the buffer in kernel address space and some kind of compatibility must be provided there.

The question is, if such a change is made impossible by the separation of the I/O aspect into its own data structure?

The answer to this is ‘‘no’’. Anything that could be added to or done with the I/O aspect of struct buf can also be added to or done with the I/O aspect if it lives in a new "struct bio".

3. Implementing a `struct bio`

The first decision to be made was who got to use the name "struct buf", and considering the fact that it is the I/O aspect which gets separated out and that it only covers about ¼ of the bytes in struct buf, obviously the new structure for the I/O aspect gets a new name. Examining the naming in the kernel, the "bio" prefix seemed a given, for instance, the function to signal completion of an I/O request is already named "biodone()".

Making the transition smooth is obviously also a priority and after some prototyping [2] it was found that a totally transparent transition could be made by embedding a copy of the new "struct bio" as the first element of "struct buf" and by using cpp(1) macros to alias the fields to the legacy struct buf names.

3.1. The b_flags problem.

Struct bio was defined by examining all code existing in the driver tree and finding all the struct buf fields which were legitimately used (as opposed to "hi-jacked" fields). One field was found to have "dual-use": the b_flags field. This required special attention. Examination showed that b_flags were used for three things:

- Communication of the I/O command (READ, WRITE, FORMAT, DELETE)
- Communication of ordering and error status
- General status for non I/O aspect consumers of struct buf.

For historic reasons B_WRITE was defined to be zero, which lead to confusion and bugs, this pushed the decision to have a separate "b_iocmd" field in struct buf and struct bio for communicating only the action to be performed.

The ordering and error status bits were put in a new flag field "b_ioflag". This has left sufficiently many now unused bits in b_flags that the b_xflags element can now be merged back into b_flags.

3.2. Definition of struct bio

With the cleanup of b_flags in place, the definition of struct bio looks like this:

[2] The software development technique previously known as "Trial & Error".

```
struct bio {
        u_int   bio_cmd;                /* I/O operation. */
        dev_t   bio_dev;                /* Device to do I/O on. */
        daddr_t bio_blkno;              /* Underlying physical block number. */
        off_t   bio_offset;             /* Offset into file. */
        long    bio_bcount;             /* Valid bytes in buffer. */
        caddr_t bio_data;               /* Memory, superblocks, indirect etc. */
        u_int   bio_flags;              /* BIO_ flags. */
        struct buf      *_bio_buf;      /* Parent buffer. */
        int     bio_error;              /* Errno for BIO_ERROR. */
        long    bio_resid;              /* Remaining I/O in bytes. */
        void    (*bio_done) __P((struct buf *));
        void    *bio_driver1;           /* Private use by the callee. */
        void    *bio_driver2;           /* Private use by the callee. */
        void    *bio_caller1;           /* Private use by the caller. */
        void    *bio_caller2;           /* Private use by the caller. */
        TAILQ_ENTRY(bio) bio_queue;     /* Disksort queue. */
        daddr_t bio_pblkno;               /* physical block number */
        struct  iodone_chain *bio_done_chain;
};
```

3.3. Definition of struct buf

After adding a struct bio to struct buf and the fields aliased into it struct buf looks like this:

```
struct buf {
        /* XXX: b_io must be the first element of struct buf for now /phk */
        struct bio b_io;                /* "Builtin" I/O request. */
#define b_bcount        b_io.bio_bcount
#define b_blkno         b_io.bio_blkno
#define b_caller1       b_io.bio_caller1
#define b_caller2       b_io.bio_caller2
#define b_data          b_io.bio_data
#define b_dev           b_io.bio_dev
#define b_driver1       b_io.bio_driver1
#define b_driver2       b_io.bio_driver2
#define b_error         b_io.bio_error
#define b_iocmd         b_io.bio_cmd
#define b_iodone        b_io.bio_done
#define b_iodone_chain  b_io.bio_done_chain
#define b_ioflags       b_io.bio_flags
#define b_offset        b_io.bio_offset
#define b_pblkno        b_io.bio_pblkno
#define b_resid         b_io.bio_resid
        LIST_ENTRY(buf) b_hash;         /* Hash chain. */
        TAILQ_ENTRY(buf) b_vnbufs;      /* Buffer's associated vnode. */
        TAILQ_ENTRY(buf) b_freelist;    /* Free list position if not active. */
        TAILQ_ENTRY(buf) b_act;         /* Device driver queue when active. *new* */
        long    b_flags;                /* B_* flags. */
        unsigned short b_qindex;        /* buffer queue index */
        unsigned char b_xflags;         /* extra flags */
[...]
```

Putting the struct bio as the first element in struct buf during a transition period allows a pointer to either to be cast to a pointer of the other, which means that certain pieces of code can be left un-converted with the use of a couple of casts while the remaining pieces of code are tested. The ccd and vinum modules have been left un-converted like this for now.

This is basically where FreeBSD-current stands today.

The next step is to substitute struct bio for struct buf in all the code which only care about the I/O aspect: device drivers, diskslice/label. The patch to do this is up for review. [3] and consists mainly of systematic substitutions like these

```
s/struct buf/struct bio/
s/b_flags/bio_flags/
s/b_bcount/bio_bcount/
&c &c
```

3.4. Future work

It can be successfully argued that the cpp(1) macros used for aliasing above are ugly and should be expanded in place. It would certainly be trivial to do so, but not by definition worthwhile.

Retaining the aliasing for the b_* and bio_* name-spaces this way leaves us with considerable flexibility in modifying the future interaction between the two. The DEV_STRATEGY() macro is the single point where a struct buf is turned into a struct bio and launched into the drivers to full-fill the I/O request and this provides us with a single isolated location for performing non-trivial translations.

As an example of this flexibility: It has been proposed to essentially drop the b_blkno field and use the b_offset field to communicate the on-disk location of the data. b_blkno is a 32bit offset of B_DEVSIZE (512) bytes sectors which allows us to address two terabytes worth of data. Using b_offset as a 64 bit byte-address would not only allow us to address 8 million times larger disks, it would also make it possible to accommodate disks which use non-power-of-two sector-size, Audio CD-ROMs for instance.

The above mentioned flexibility makes an implementation almost trivial:

- Add code to DEV_STRATEGY() to populate b_offset from b_blkno in the cases where it is not valid. Today it is only valid for a struct buf marked B_PHYS.
- Change diskslice/label, ccd, vinum and device drivers to use b_offset instead of b_blkno.
- Remove the bio_blkno field from struct bio, add it to struct buf as b_blkno and remove the cpp(1) macro which aliased it into struct bio.

Another possible transition could be to not have a "built-in" struct bio in struct buf. If for some reason struct bio grows fields of no relevance to struct buf it might be cheaper to remove struct bio from struct buf, un-alias the fields and have DEV_STRATEGY() allocate a struct bio and populate the relevant fields from struct buf. This would also be entirely transparent to both users of struct buf and struct bio as long as we retain the aliasing mechanism and DEV_STRATEGY().

[3] And can be found at http://phk.freebsd.dk/misc

4. Towards a stackable BIO subsystem.

Considering that we now have three distinct pieces of code living in the nowhere between DEV_STRATEGY() and the device drivers: diskslice/label, ccd and vinum, it is not unreasonable to start to look for a more structured and powerful API for these pieces of code.

In traditional UNIX semantics a "disk" is a one-dimensional array of 512 byte sectors which can be read or written. Support for sectors of multiple of 512 bytes were implemented with a sort of "don't ask-don't tell" policy where system administrator would specify a larger minimum sector-size to the filesystem, and things would "just work", but no formal communication about the size of the smallest transfer possible were exchanged between the disk driver and the filesystem.

A truly generalised concept of a disk needs to be more flexible and more expressive. For instance, a user of a disk will want to know:

- What is the sector size. Sector-size these days may not be a power of two, for instance Audio CDs have 2352 byte "sectors".
- How many sectors are there.
- Is writing of sectors supported.
- Is freeing of sectors supported. This is important for flash based devices where a wear-distribution software or hardware function uses the information about which sectors are actually in use to optimise the usage of the slow erase function to a minimum.
- Is opening this device in a specific mode, (read-only or read-write) allowed. The VM system and the file-systems generally assume that nobody writes to "their storage" under their feet, and therefore opens which would make that possible should be rejected.
- What is the "native" geometry of this device (Sectors/Heads/Cylinders). This is useful for staying compatible with badly designed on-disk formats from other operating systems.

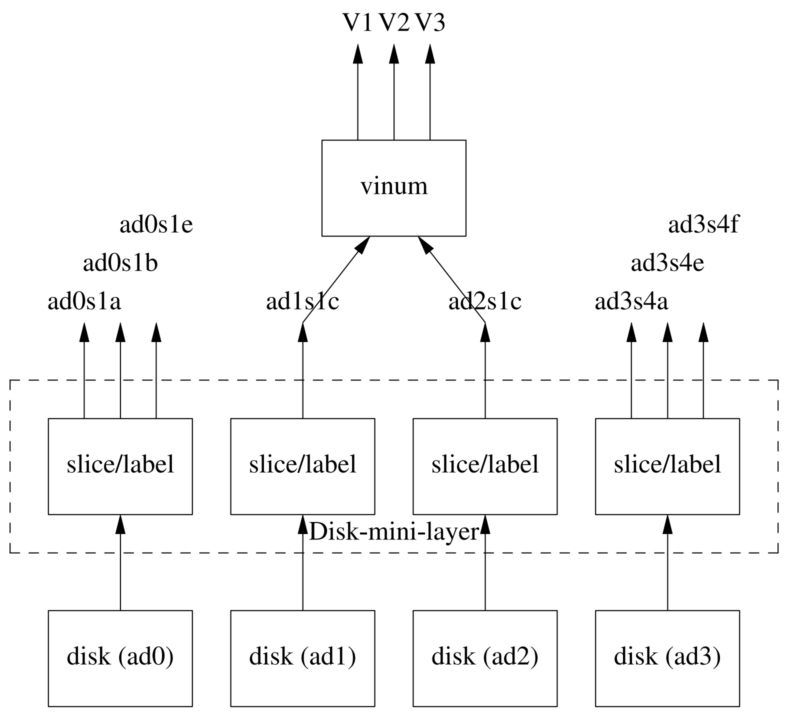
Obviously, all of these properties are dynamic in the sense that in these days disks are removable devices, and they may therefore change at any time. While some devices like CD-ROMs can lock the media in place with a special command, this cannot be done for all devices, in particular it cannot be done with normal floppy disk drives.

If we adopt such a model for disk, retain the existing "strategy/biodone" model of I/O scheduling and decide to use a modular or stackable approach to geometry translations we find that nearly endless flexibility emerges: Mirroring, RAID, striping, interleaving, disk-labels and sub-disks, all of these techniques would get a common framework to operate in.

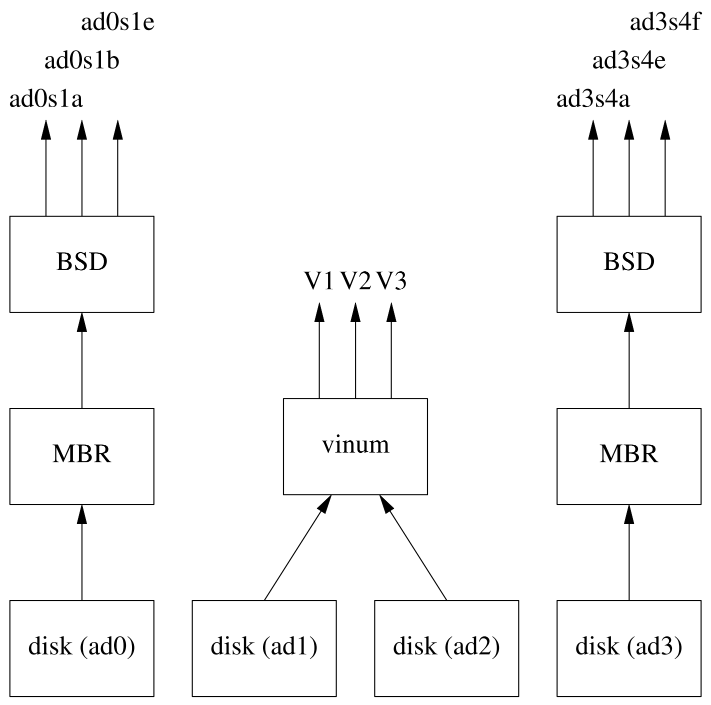
In practice of course, such a scheme must not complicate the use of or installation of FreeBSD. The code will have to act and react exactly like the current code but fortunately the current behaviour is not at all hard to emulate so implementation-wise this is a non-issue.

But lets look at some drawings to see what this means in practice.

Today the plumbing might look like this on a machine:

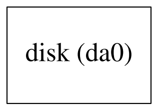


And while this drawing looks nice and clean, the code underneat isn't. With a stackable BIO implementation, the picture would look like this:

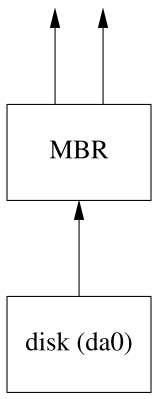


The first thing we notice is that the disk mini-layer is gone, instead separate modules for the Microsoft style MBR and the BSD style disklabel are now stacked over the disk. We can also see that Vinum no longer needs to go though the BSD/MBR layers if it wants access to the entire physical disk, it can be stacked right over the disk.

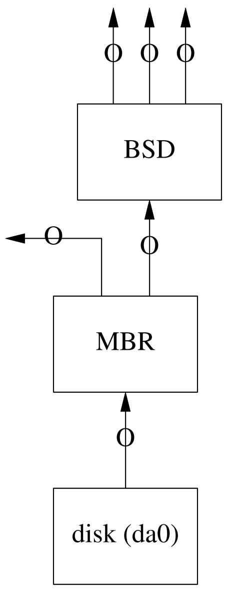
Now, imagine that a ZIP drive is connected to the machine, and the user loads a ZIP disk in it. First the device driver notices the new disk and instantiates a new disk:



A number of the geometry modules have registered as "auto-discovering" and will be polled sequentially to see if any of them recognise what is on this disk. The MBR module finds a MBR in sector 0 and attach an instance of itself to the disk:



It finds two "slices" in the MBR and creates two new "disks" one for each of these. The polling of modules is repeated and this time the BSD label module recognises a FreeBSD label on one of the slices and attach itself:



The BSD module finds three partitions, creates them as disks and the polling is repeated for each of these. No modules recognise these and the process ends. In theory one could have a module recognise the

UFS superblock and extract from there the path to mount the disk on, but this is probably better implemented in a general "device-daemon" in user-land.

On this last drawing I have marked with "O" the "disks" which can be accessed from user-land or kernel. The VM and file-systems generally prefer to have exclusive write access to the disk sectors they use, so we need to enforce this policy. Since we cannot know what transformation a particular module implements, we need to ask the modules if the open is OK, and they may need to ask their neighbours before they can answer.

We decide to mount a filesystem on one of the BSD partitions at the very top. The open request is passed to the BSD module, which finds that none of the other open partitions (there are none) overlap this one, so far no objections. It then passes the open to the MBR module, which goes through basically the same procedure finds no objections and pass the request to the disk driver, which since it was not previously open approves of the open.

Next we mount a filesystem on the next BSD partition. The BSD module again checks for overlapping open partitions and find none. This time however, it finds that it has already opened the "downstream" in R/W mode so it does not need to ask for permission for that again so the open is OK.

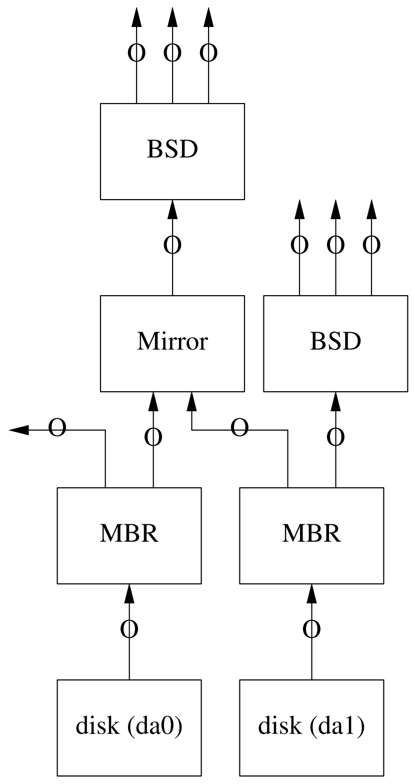
Next we mount a msdos filesystem on the other MBR slice. This is the same case, the MBR finds no overlapping open slices and has already opened "downstream" so the open is OK.

If we now try to open the other slice for writing, the one which has the BSD module attached already. The open is passed to the MBR module which notes that the device is already opened for writing by a module (the BSD module) and consequently the open is refused.

While this sounds complicated it actually took less than 200 lines of code to implement in a prototype implementation.

Now, the user ejects the ZIP disk. If the hardware can give a notification of intent to eject, a call-up from the driver can try to get devices synchronised and closed, this is pretty trivial. If the hardware just disappears like a unplugged parallel zip drive, a floppy disk or a PC-card, we have no choice but to dismantle the setup. The device driver sends a "gone" notification to the MBR module, which replicates this upwards to the mounted msdosfs and the BSD module. The msdosfs unmounts forcefully, invalidates any blocks in the buf/vm system and returns. The BSD module replicates the "gone" to the two mounted file-systems which in turn unmounts forcefully, invalidates blocks and return, after which the BSD module releases any resources held and returns, the MBR module releases any resources held and returns and all traces of the device have been removed.

Now, let us get a bit more complicated. We add another disk and mirror two of the MBR slices:



Now assuming that we lose disk da0, the notification goes up like before but the mirror module still has a valid mirror from disk da1, so it doesn't propagate the "gone" notification further up and the three file-systems mounted are not affected.

It is possible to modify the graph while in action, as long as the modules know that they will not affect any I/O in progress. This is very handy for moving things around. At any of the arrows we can insert a mirroring module, since it has a 1:1 mapping from input to output. Next we can add another copy to the mirror, give the mirror time to sync the two copies. Detach the first mirror copy and remove the mirror module. We have now in essence moved a partition from one disk to another transparently.

5. Getting stackable BIO layers from where we are today.

Most of the infrastructure is in place now to implement stackable BIO layers:

• The dev_t change gave us a public structure where information about devices can be put. This enabled us to get rid of all the NFOO limits on the number of instances of a particular driver/device, and significantly cleaned up the vnode aliasing for device vnodes.

• The disk-mini-layer has taken the knowledge about diskslice/labels out of the majority of the disk-drivers, saving on average 100 lines of code per driver.

• The struct bio/buf divorce is giving us an IO request of manageable size which can be modified without affecting all the filesystem and VM system users of struct buf.

The missing bits are:

• changes to struct bio to make it more stackable. This mostly relates to the handling of the biodone() event, something which will be transparent to all current users of struct buf/bio.

• code to stich modules together and to pass events and notifications between them.

6. An Implementation plan for stackable BIO layers

My plan for implementation stackable BIO layers is to first complete the struct bio/buf divorce with the already mentioned patch.

The next step is to re-implement the monolithic disk-mini-layer so that it becomes the stackable BIO system. Vinum and CCD and all other consumers should not be unable to tell the difference between the current and the new disk-mini-layer. The new implementation will initially use a static stacking to remain compatible with the current behaviour. This will be the next logical checkpoint commit.

The next step is to make the stackable layers configurable, to provide the means to initialise the stacking and to subsequently change it. This will be the next logical checkpoint commit.

At this point new functionality can be added inside the stackable BIO system: CCD can be re-implemented as a mirror module and a stripe module. Vinum can be integrated either as one "macro-module" or as separate functions in separate modules. Also modules for other purposes can be added, sub-disk handling for Solaris, MacOS, etc etc. These modules can be committed one at a time.