The Vinum Volume Manager

Greg Lehey
Nan Yang Computer Services Ltd.
PO Box 460
Echunga SA 5153
grog@lemis.com
08-8388-8286

ABSTRACT

This paper describes the *Vinum Volume Manager*, a block device driver which implements virtual disk drives. It isolates disk hardware from the block device interface and maps data in ways which result in an increase in flexibility, performance and reliability compared to the traditional slice view of disk storage. *Vinum* implements the RAID-0, RAID-1 and RAID-5 models, both individually and in combination.

Introduction

Despite the rapid evolution of disk hardware, the current UNIX disk abstraction is inadequate for a number of modern applications. In particular, file systems must be stored on a single disk partition (or volume), and there is no kernel support for redundant data storage. In addition, the direct relationship between disk volumes and their location on disk make it generally impossible to enlarge a disk volume once it has been created. Performance can often be limited by the maximum data rate which can be achieved with the disk hardware.

The largest modern disks store only about 30 GB, but large installations routinely have more than a terabyte of disk storage, and it is not uncommon to see disk storage of several hundred gigabytes even on PCs. Storage-intensive applications such as Internet World-Wide Web and FTP servers have accelerated the demand for high-performance, high-volume, reliable storage systems.

The current trend is to realize such systems in *disk array* hardware, which looks like a very large disk to the host system, but which spreads the data over a number of disks, possibly in a redundant fashion such as RAID-1 or RAID-5. Disk arrays have a number of advantages:

- They are portable. Since they have a standard interface, usually SCSI, but sometimes IDE, they can be installed on almost any system without kernel modifications.
- They can offer impressive performance: they offload the calculations (in particular, the parity calculations for RAID-5) to the array, and in the case of replicated data, the aggregate transfer rate to the array is less than it would be to local disks. RAID-0 (striping) and RAID-5 organizations also spread the load more evenly over the physical disks, thus improving performance. Nevertheless, an array is typically connected via

a single SCSI connection, which can be a bottleneck.

- They are reliable. A good disk array offers a large number of features designed to enhance reliability, including enhanced cooling, hot-plugging (the ability to replace a drive while the array is running) and automatic failure recovery.

On the other hand, disk arrays are relatively expensive and not particularly flexible. An alternative is a software-based *volume manager* which performs similar functions in software. A number of these systems exist, notably the VERITAS® volume manager [Veritas], Solaris *DiskSuite* [Solstice], IBM's *Logical Volume Facility* [IBM] and SCO's *Virtual Disk Manager* [SCO]. An implementation of RAID software is also available for Linux [Linux].

Vinum

Vinum is an open source [OpenSource] volume manager implemented under FreeBSD [FreeBSD]. A number of features distinguish it from commercial volume managers:

- *Vinum* implements RAID-0 (striping), RAID-1 (mirroring) and RAID-5 (rotated block-interleaved parity). In RAID-5, a group of disks are protected against the failure of any one disk by an additional disk with block checksums of the other disks.[1]
- Drive layouts can be combined to increase robustness, including striped mirrors (so-called ''RAID-10'').
- *Vinum* implements only those features which appear useful. Some commercial volume managers appear to have been implemented with the goal of maximizing the size of the spec sheet. *Vinum* does not implement ''ballast'' features such as RAID-4, although it would have been trivial to do so.
- Volume managers initially emphasized reliability and performance rather than ease of use. The results are frequently down time due to misconfiguration, with consequent reluctance on the part of operational personnel to attempt to use the more unusual features of the product. *Vinum* attempts to provide an easier-to-use non-GUI interface.

Concepts

As used in this document, a *volume manager* is a software component which isolates file systems from the underlying disk hardware. Instead of building file systems on *disk partitions*, they are built on logical disks, called *volumes*. This has a number of advantages:

- Volumes may span disk drives.
- Volumes may be larger than any drive.

1. The RAID-5 functionality is currently available under license from Cybernet, Inc. [Cybernet]. It will be released as open source at a later date.

- By spreading the disk load over multiple volumes, it is possible to improve performance.
- By replicating data within the volume, it is possible to improve availability.
- By changing the volume configuration, it is possible to reorganize disk storage online.
- It is possible to extend the size of volumes.

To achieve these results, *Vinum* defines a hierarchy of four logical objects:

- A *volume* is a logical disk. From a user viewpoint, it is almost indistinguishable from a disk partition, the conventional representation of a logical disk. A volume contains one or more *plexes*.
- A *plex* is a representation of the data in a volume. Each plex has an address space the same size as the size of the volume, though it is not required that the address space be completely mapped to disk storage. Each plex represents a (possibly incomplete) copy of the data in the volume, thus providing protection against disk failure. This represents an implementation of RAID-1.
- Each plex contains one or more *subdisks*. A subdisk is a contiguous segment of disk storage. Conceptually, it is similar to a disk slice, but the implementation is different. In particular, a disk slice contains metadata such as labels, while a subdisk does not. A plex can be extended in length by adding subdisks to it: since subdisks can be located on any device under *Vinum* control, there is no requirement for contiguous free space in order to expand a plex.
- A *drive* is a hardware component which may contain subdisks. From an implementation viewpoint, it may be a complete device or a disk slice. Vinum does not depend on any particular disk hardware, though it is intended for use primarily on conventional hard disks.

Plex organization

Subdisks may be mapped to plexes in one of three different ways. The following figures illustrate the possible ways of mapping blocks of 4096 (`0x1000`) bytes in a plex with four subdisks.

- A *concatenated* plex maps the subdisks to the plex sequentially, corresponding to RAID-0. The plex maps to the complete address space of each subdisk in turn. In a concatenated plex, subdisks do not need to be of equal size.

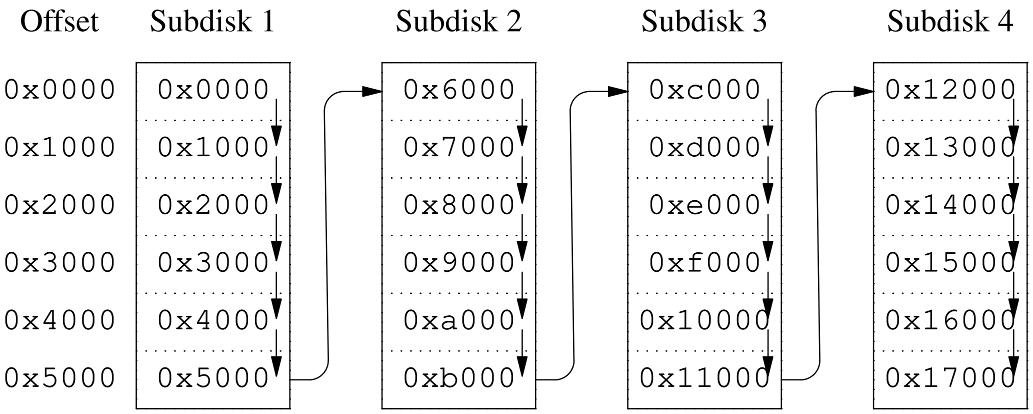


Figure 1: Concatenated plex

The dotted lines in this figure represetn the logical blocks in order to illustrate the differences from the other organizations.

- A *striped plex* maps the plex address space to equal-sized blocks of each subdisk in turn. As a result, the subdisks in a plex must all have the same size:

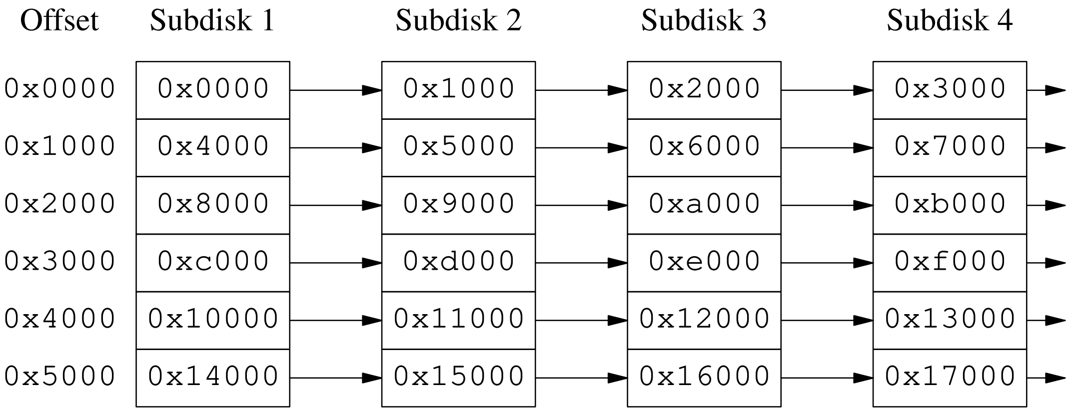


Figure 2: Striped plex

- As implemented by *Vinum*, a *RAID-5* plex is similar to a striped plex, except that it implements RAID-5 by including a parity block in each stripe. As required by RAID-5, the location of this parity block changes from one stripe to the next:

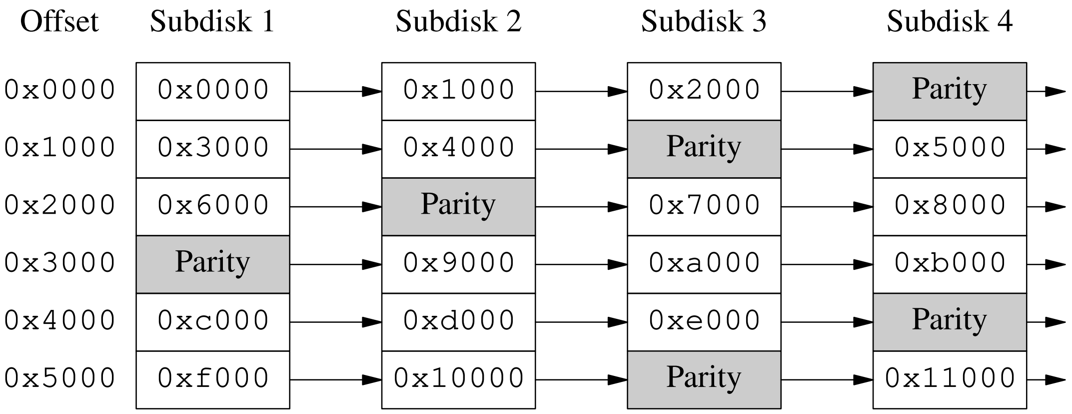


Figure 3: RAID-5 plex

Which plex organization?

Vinum implements only that subset of RAID organizations which make sense in the framework of the implementation. It would have been possible to implement all RAID levels, but there was no reason to do so. Each of the chosen organizations has unique advantages:

- Concatenated plexes are the most flexible: they can contain any number of subdisks, and the subdisks may be of different length. The plex may be extended by adding additional subdisks. They require less CPU time than striped or RAID-5 plexes, though the difference in CPU overhead from striped plexes is not significant. On the other hand, they are most susceptible to hot spots, where one disk is very active and others are idle.
- The greatest advantage of striped (RAID-0) plexes is that they reduce hot spots: by choosing an optimum sized stripe (empirically determined to be in the order of 256 kB), the load on the component drives can be made more even. The disadvantages of this approach are (fractionally) more complex code and restrictions on subdisks: they must be all the same size, and extending a plex by adding new subdisks is so complicated that *Vinum* does not implement it. An additional, trivial restriction is that a striped plex must have at least two subdisks, since otherwise it is indistinguishable from a concatenated plex.
- The implementation of RAID-5 plexes stretches the concept of the volume manager somewhat. While RAID-1 is implemented at the volume level, RAID-5 is implemented at the plex level. As implemented, RAID-5 plexes are effectively an extension of striped plexes. Compared to striped plexes, they offer the advantage of fault tolerance, but the disadvantages of higher storage cost and significantly higher CPU overhead, particularly for writes. The code is an order of magnitude more complex than for concatenated and striped plexes. Like striped plexes, RAID-5 plexes must have equal-sized subdisks and cannot be extended. *Vinum* enforces a minimum of three subdisks for a RAID-5 plex, since any smaller number would not make any sense.

The following table gives an overview of the advantages and disadvantages of each plex organization.

Plex type	Minimum subdisks	can add subdisks	must be equal size	application
concatenated	1	yes	no	Large, non-redundant data storage
striped	2	no	yes	Highly concurent access
RAID-5	3	no	yes	Highly reliable storage, primarily read access

Figure 4: Vinum plex organizations

These are not the only possible organizations. In addition, the following could have been implemented:

- RAID-4, which differs from RAID-5 only by the fact that all parity data is stored on a specific disk. This simplifies the algorithms somewhat at the expense of drive utilization: the activity on the parity disk is a direct function of the read to write ratio. Since *Vinum* implements RAID-5, RAID-4's only advantage is nullified.
- RAID-3, effectively an implementation of RAID-4 with a stripe size of one byte. Each transfer requires reading each disk (with the exception of the parity disk for reads). Without spindle synchronization (where the corresponding sectors pass the heads of each drive at the same time), RAID-3 would be very inefficient. In a multiple-access system, it also causes high latency.
- RAID-2, which uses two subdisks to store a Hamming code, and which otherwise resembles RAID-3. Compared to RAID-3, it offers a lower data density, higher CPU usage and no compensating advantages.

In addition, RAID-5 can be interpreted in two different ways: the data can be striped, as in the *Vinum* implementation, or it can be written serially, exhausting the address space of one subdisk before starting on the other, effectively a modified concatenated organization. There is no recognizable advantage to this approach, since it does not provide any of the other advantages of concatenation.

Configuring *Vinum*

Vinum maintains a *configuration database* which describes the objects known to an individual system. Initially, the user creates the configuration database from one or more configuration files with the aid of the *vinum(8)* utility program. *Vinum* stores a copy of its configuration database on each disk slice (which *Vinum* calls a *device*) under its control. This database is updated on each state change, so that a restart accurately restores the state of each *Vinum* object.

The configuration file

The configuration file describes individual *Vinum* objects. The definition of a simple volume might be:

```
drive a device /dev/sd0h
drive b device /dev/sd1h
drive c device /dev/sd2h
drive d device /dev/sd3h
volume myvol
  plex org concat
    sd length 512m drive a
    sd length 512m drive b
  plex org concat
    sd length 512m drive c
    sd length 512m drive d
```

This file describes a total of 11 *Vinum* objects:

- The `drive` line describe four disk partitions (*drives*) and their location relative to the underlying hardware. They are given the symbolic names *a*, *b*, *c* and *d*.
- The `volume` line describes a volume. The only required attribute is the name, in this case `myvol`.
- The `plex` lines define plexes. The only required parameter is the organization, in this case `concat`. No name is necessary: the system automatically generates a name from the volume name by adding the suffix `.p`***x***, where ***x*** is the number of the plex in the volume. Thus the first plex will be called *myvol.p0* and the second will be called *myvol.p1*.
- The `sd` lines describe subdisks. The minimum specifications are the name of a driver on which to store it, and the length of the subdisk. As with plexes, no name is necessary: the system automatically assigns names derived from the plex name by adding the suffix `.s`***x***, where ***x*** is the number of the subdisk in the plex. Thus *Vinum* gives these four subdisks the names *myvol.p0.s0*, *myvol.p0.s1*, *myvol.p1.s0* and *myvol.p1.s1* respectively.

After processing this file, *vinum(8)* produces the following output:

```
vinum -> create config1
Configuration summary

Drives:         4 (4 configured)
Volumes:        1 (4 configured)
Plexes:         2 (8 configured)
Subdisks:       4 (16 configured)

D a                     State: up       Device /dev/sd0h
D b                     State: up       Device /dev/sd1h
D c                     State: up       Device /dev/sd2h
D d                     State: up       Device /dev/sd3h

V myvol                 State: up       Plexes:       2 Size:       1024 MB

P myvol.p0            C State: up       Subdisks:     2 Size:       1024 MB
P myvol.p1            C State: up       Subdisks:     2 Size:       1024 MB

S myvol.p0.s0           State: up       PO:        0  B Size:        512 MB
```

```
S myvol.p0.s1           State: up       PO:      512 MB Size:         512 MB
S myvol.p1.s0           State: up       PO:        0  B Size:         512 MB
S myvol.p1.s1           State: up       PO:      512 MB Size:         512 MB
```

This output shows the brief listing format of *vinum(8)*.

The following figure demonstrates the layout in graphic form:

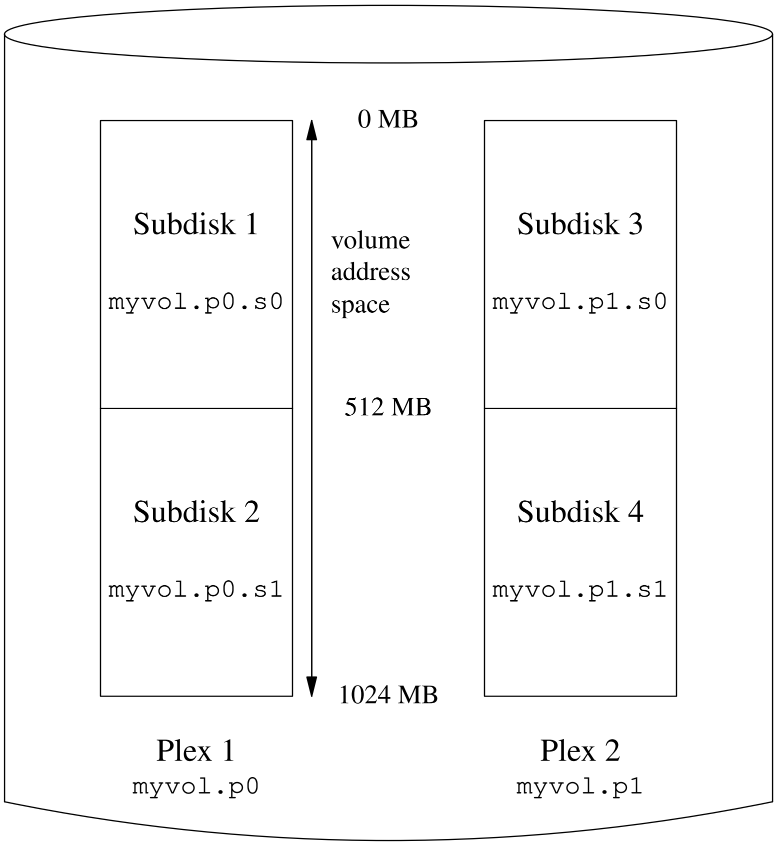


Figure 5: A Vinum volume

In this representation, each plex covers the complete address space of the volume. Subdisks *myvol.p0.s0* and *myvol.p1.s0* cover the first half of the address space, and subdisks *myvol.p0.s1* and *myvol.p1.s1* cover the second half of the address space.

Since *Vinum* stores these definitions in the configuration database, there is never any need to define them again in another configuration file. At a later date the user might want to create another volume and add a plex to the existing volume *vol*. In order to do so, he might need to add another drive to the *Vinum* configuration. He could perform these tasks with the following configuration file:

```
plex volume myvol org striped 256k
  sd size 256m drive a
  sd size 256m drive b
  sd size 256m drive c
  sd size 256m drive d
```

```
drive e device /dev/sd4h
volume bigraid
  plex org raid5 256k
    sd size 2g drive a
    sd size 2g drive b
    sd size 2g drive c
    sd size 2g drive d
    sd size 2g drive e
```

In this example, the user first adds another plex to volume *myvol*. The system automatically assigns it the name *myvol.p2*. The plex contains four subdisks, and the data is striped across the subdisks in stripes of 256 kB.

The new volume is called *bigraid* and contains one plex with a RAID-5 organization and five subdisks. Again the stripe size is 256 kB. In order to accomodate the fifth subdisk, the user defines a fifth drive, drive *e*, which is on the disk partition */dev/sd4h*. He doesn't need to define the others, since they are already known to the system.

After running *vinum(8)*, the configuration looks like:

```
vinum -> create config2
Configuration summary

Drives:         5 (8 configured)
Volumes:        2 (4 configured)
Plexes:         4 (8 configured)
Subdisks:       13 (16 configured)

D a                     State: up       Device /dev/sd0h
D b                     State: up       Device /dev/sd1h
D c                     State: up       Device /dev/sd2h
D d                     State: up       Device /dev/sd3h
D e                     State: up       Device /dev/sd4h

V myvol                 State: up       Plexes:       3 Size:       1024 MB
V bigraid               State: up       Plexes:       1 Size:          8 GB

P myvol.p0            C State: up       Subdisks:     2 Size:       1024 MB
P myvol.p1            C State: up       Subdisks:     2 Size:       1024 MB
P myvol.p2            S State: up       Subdisks:     4 Size:       1024 MB
P bigraid.p0         R5 State: init     Subdisks:     5 Size:          8 GB

S myvol.p0.s0           State: up       PO:        0  B Size:        512 MB
S myvol.p0.s1           State: up       PO:      512 MB Size:        512 MB
S myvol.p1.s0           State: up       PO:        0  B Size:        512 MB
S myvol.p1.s1           State: up       PO:      512 MB Size:        512 MB
S myvol.p2.s0           State: up       PO:        0  B Size:        256 MB
S myvol.p2.s1           State: up       PO:      256 MB Size:        256 MB
S myvol.p2.s2           State: up       PO:      512 MB Size:        256 MB
S myvol.p2.s3           State: up       PO:      768 MB Size:        256 MB
S bigraid.p0.s0         State: init     PO:        0  B Size:       2048 MB
S bigraid.p0.s1         State: init     PO:     2048 MB Size:       2048 MB
S bigraid.p0.s2         State: init     PO:     4096 MB Size:       2048 MB
S bigraid.p0.s3         State: init     PO:     6144 MB Size:       2048 MB
S bigraid.p0.s4         State: init     PO:     8192 MB Size:       2048 MB
```

Startup

The configuration information is stored on the disk slices in essentially the same form as in the configuration files, which enables the same routines to be used for initialization. When reading from the configuration database, *vinum(8)* recognizes a number of keywords which are not allowed in the configuration files. At a point where the user has commenced initialization of plex *bigraid.p0*, the configuration database might contain:

```
drive a state up device /dev/sd0h
drive b state up device /dev/sd1h
drive c state up device /dev/sd2h
drive d state up device /dev/sd3h
drive e state up device /dev/sd4h
volume myvol state up
volume bigraid state down
plex name myvol.p0 state up org concat vol myvol
plex name myvol.p1 state up org concat vol myvol
plex name myvol.p2 state init org striped 512b vol myvol
plex name bigraid.p0 state initializing org raid5 512b vol bigraid
sd name myvol.p0.s0 drive a plex myvol.p0 state up len 1048576b driveoffset 265
b plexoffset 0b
sd name myvol.p0.s1 drive b plex myvol.p0 state up len 1048576b driveoffset 265
b plexoffset 1048576b
sd name myvol.p1.s0 drive c plex myvol.p1 state up len 1048576b driveoffset 265
b plexoffset 0b
sd name myvol.p1.s1 drive d plex myvol.p1 state up len 1048576b driveoffset 265
b plexoffset 1048576b
sd name myvol.p2.s0 drive a plex myvol.p2 state init len 524288b driveoffset 10
48841b plexoffset 0b
sd name myvol.p2.s1 drive b plex myvol.p2 state init len 524288b driveoffset 10
48841b plexoffset 524288b
sd name myvol.p2.s2 drive c plex myvol.p2 state init len 524288b driveoffset 10
48841b plexoffset 1048576b
sd name myvol.p2.s3 drive d plex myvol.p2 state init len 524288b driveoffset 10
48841b plexoffset 1572864b
sd name bigraid.p0.s0 drive a plex bigraid.p0 state initializing len 4194304b d
riveoffset 1573129b plexoffset 0b
sd name bigraid.p0.s1 drive b plex bigraid.p0 state initializing len 4194304b d
riveoffset 1573129b plexoffset 4194304b
sd name bigraid.p0.s2 drive c plex bigraid.p0 state initializing len 4194304b d
riveoffset 1573129b plexoffset 8388608b
sd name bigraid.p0.s3 drive d plex bigraid.p0 state initializing len 4194304b d
riveoffset 1573129b plexoffset 12582912b
sd name bigraid.p0.s4 drive e plex bigraid.p0 state initializing len 4194304b d
riveoffset 1573129b plexoffset 16777216b
```

The obvious differences here are the presence of explicit location information and naming (both of which are also allowed, but discouraged, for use by the user) and the information on the states (which are not available to the user).

At system startup, *Vinum* reads the configuration database from one of the *Vinum* drives. Under normal circumstances, each drive contains an identical copy of the configuration database, so it does not matter which drive is read. After a crash, however, *Vinum* must determine which drive was updated most recently and read the configuration from this drive.

Object naming

As described above, *Vinum* assigns default names to plexes and subdisks, although they may be overridden. Overriding the default names is not recommended: experience with the VERITAS volume manager, which allows arbitary naming of objects, has shown that this flexibility does not bring a significant advantage, and it can cause confusion.

Names may contain any non-blank character, but it is recommended to restrict them to letters, digits and the underscore characters. The names of volumes, plexes and subdisks may be up to 64 characters long, and the names of drives may up to 32 characters long.

After reading the configuration database, *vinum(8)* creates a directory */dev/vinum*, in which it makes device entries for each volume it finds. It also creates subdirectories */dev/vinum/vol*, */dev/vinum/plex*, */dev/vinum/sd* and */dev/vinum/drive*, in which it stores device entries for the corresponding objects. The directories */dev/vinum/vol* and */dev/vinum/plex* contain subdirectories representing the hierarchy of the plexes and subdisks attached to them.

After processing the first of the configuration files described above, *Vinum* creates the following devices:

```
/dev/vinum:
total 5
brwx------  1 root  wheel   25, 0x40000000 Jul 28 10:57 control
drwxr-xr-x  2 root  wheel       512 Jul 28 10:57 drive
drwxr-xr-x  2 root  wheel       512 Jul 28 10:57 plex
drwxr-xr-x  2 root  wheel       512 Jul 28 10:57 sd
drwxr-xr-x  3 root  wheel       512 Jul 28 10:57 vol

/dev/vinum/drive:
total 0
brw-r-----  1 root  operator    4,  39 May 25 12:32 a
brw-r-----  1 root  operator    4,  15 May 24 14:05 b
brw-r-----  1 root  operator    4,  23 May 24 14:05 c
brw-r-----  1 root  operator    4,  31 May 24 14:05 d

/dev/vinum/plex:
total 0
brwxr-xr--  1 root  wheel   25, 0x10000000 Jul 28 10:57 myvol.p0
brwxr-xr--  1 root  wheel   25, 0x10010000 Jul 28 10:57 myvol.p1

/dev/vinum/sd:
total 0
brwxr-xr--  1 root  wheel   25, 0x20000000 Jul 28 10:57 myvol.p0.s0
brwxr-xr--  1 root  wheel   25, 0x20100000 Jul 28 10:57 myvol.p0.s1
brwxr-xr--  1 root  wheel   25, 0x20010000 Jul 28 10:57 myvol.p1.s0
brwxr-xr--  1 root  wheel   25, 0x20110000 Jul 28 10:57 myvol.p1.s1

/dev/vinum/vol:
total 1
brwxr-xr--  1 root  wheel   25,   0 Jul 28 10:57 myvol
drwxr-xr-x  4 root  wheel       512 Jul 28 10:57 myvol.plex

/dev/vinum/vol/myvol.plex:
total 2
brwxr-xr--  1 root  wheel   25, 0x10000000 Jul 28 10:57 myvol.p0
drwxr-xr-x  2 root  wheel       512 Jul 28 10:57 myvol.p0.sd
brwxr-xr--  1 root  wheel   25, 0x10010000 Jul 28 10:57 myvol.p1
drwxr-xr-x  2 root  wheel       512 Jul 28 10:57 myvol.p1.sd

/dev/vinum/vol/myvol.plex/myvol.p0.sd:
total 0
brwxr-xr--  1 root  wheel   25, 0x20000000 Jul 28 10:57 myvol.p0.s0
brwxr-xr--  1 root  wheel   25, 0x20100000 Jul 28 10:57 myvol.p0.s1

/dev/vinum/vol/myvol.plex/myvol.p1.sd:
total 0
brwxr-xr--  1 root  wheel   25, 0x20010000 Jul 28 10:57 myvol.p1.s0
brwxr-xr--  1 root  wheel   25, 0x20110000 Jul 28 10:57 myvol.p1.s1
```

Unlike UNIX drives, *Vinum* volumes are not partitioned, and thus do not contain a partition table. This has required modification to some disk utilities, notably *newfs*, which previously tried to interpret the last letter of a *Vinum* volume name as a slice identifier.

Although it is recommended that plexes and subdisks should not be allocated specific names, *Vinum* drives must be named. This makes it possible to move a drive to a different location and still recognize it automatically. Drive names may be up to 32 characters long.

The implementation

At the time of writing, *Vinum* is still in a late implementation stage. Many aspects of the implementation are subject to change. This section examines some of the more interesting tradeoffs in the implementation.

Where the driver fits

To the operating system, *Vinum* looks like a block device, so it is normally be accessed as a block device. Instead of operating directly on the device, it creates new requests and passes them to the real device drivers. Conceptually it could pass them to other *Vinum* devices, though this usage makes no sense and would probably cause problems. The following figure, borrowed from [McKusick],[2] shows the standard 4.4BSD I/O structure:

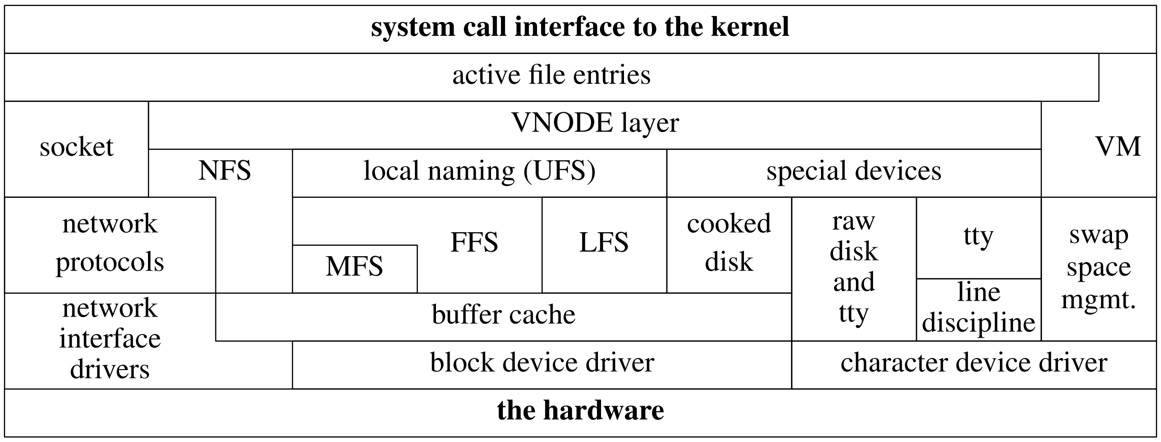


Figure 6: Kernel I/O structure, after McKusick et. al.

The following figure shows the I/O structure in FreeBSD after installing *Vinum*. Apart from the effect of *Vinum*, it shows the gradual lack of distinction between block and character devices that has occurred since the release of 4.4BSD: FreeBSD implements disk character devices in the corresponding block driver.

2. This figure is © 1996 Addison-Wesley, and is reproduced with permission.

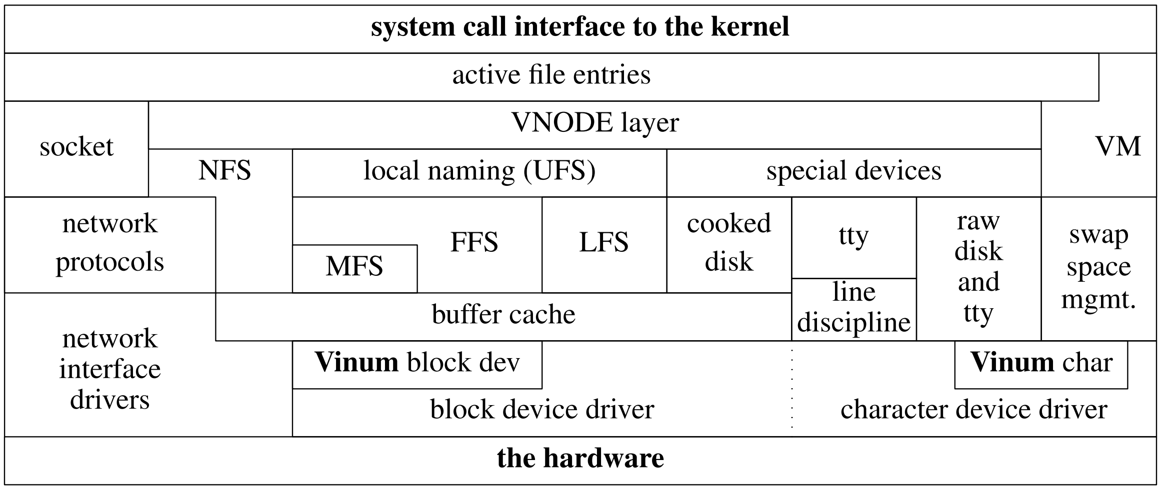


Figure 7: Kernel I/O structure with *Vinum*

Design limitations

Vinum was intended to have as few arbitrary limits as possible consistent with an efficient implementation. Nevertheless, a number of limits were imposed in the interests of efficiency, mainly in connection with the device minor number format. These limitations will no longer be relevant after the introduction of a device file system.

Restriction	Reasoning
Fixed maximum number of volumes per system.	In order to maintain compatibility with other versions of UNIX, it was considered desirable to keep the device numbers of volume in the traditional 8+8 format (8 bits major number, 8 bits minor number). This restricts the number of volumes to 256. In view of the fact that *Vinum* provides for larger volumes than disks, and current machines are seldom able to control more than 64 disk drives, this restriction seems unlikely to become severe for some years to come.
Fixed number of plexes per volume	Plexes supply redundancy according to RAID-1. For this purpose, two plexes are sufficient under normal circumstances. For rebuilding and archival purposes, additional plexes can be useful, but it is difficult to find a situation where more than four plexes are necessary or useful. On the other hand, additional plexes beyond four bring little advantage for reading and a significant disadvantage for writing. I believe that eight plexes are ample.

Fixed maximum number of subdisks per plex.	For similar reasons, the number of subdisks was limited to 256. It seldom makes sense to have more than about 10 subdisks per plex, so this restriction does not currently appear severe. There is no specific overall limitation on the number of subdisks.
Minimum device size	A device must contain at least 1 MB of storage. This assumption makes it possible to dispense with some boundary condition checks. *Vinum* requires 133 kB of disk space to store the header and configuration information, so this restriction does not appear serious.

Memory allocation

In order to perform its functionality, *Vinum* allocates a large number of dynamic data structures. Currently these structures are allocated by calling kernel `malloc`. This is a potential problem, since `malloc` interacts with the virtual memory system and may trigger a page fault. The potential for a deadlock exists if the page fault requires a transfer to a *Vinum* volume. It is probably that *Vinum* will modify its allocation strategy by reserving a small number of buffers when it starts and using these if a `malloc` request fails.

To cache or not to cache

Traditionally, UNIX block devices are accessed from the file system via caching routines such as *bread* and *bwrite*. It is also possible to access them directly, but this facility is seldom used. The use of caching enables significant improvements in performance.

Vinum does not cache the data it passes to the lower-level drivers. It would also seem counterproductive to do so: the data is available in cache already, and the only effect of caching it a second time would be to use more memory, thus causing more frequent cache misses.

RAID-5 plexes pose a problem to this reasoning. A RAID-5 write normally first reads the parity block, so there might be some advantage in caching at least the parity blocks. This issue has been deferred for further study.

Access optimization

The algorithms for RAID-5 access are surprisingly complicated and require a significant amount of temporary data storage. To achieve reasonable performance, they must take error recovery strategies into account at a low level. A RAID 5 access can require one or more of the following actions:

- *Normal read*. All participating subdisks are up, and the transfer can be made directly to the user buffer.

- *Recovery read*. One participating subdisk is down. To recover data, all the other subdisks, including the parity subdisk, must be read. The data is recovered by exclusive-oring all the other blocks.
- *Normal write*. All the participating subdisks are up. This write proceeds in four phases:
  1. Read the old contents of each block and the parity block.
  2. ‘‘Remove’’ the old contents from the parity block with exclusive or.
  3. ‘‘Insert’’ the new contents of the block in the parity block, again with exclusive or.
  4. Write the new contents of the data blocks and the parity block. The data block transfers can be made directly from the user buffer.
- *Degraded write* where the data block is not available. This requires the following steps:
  1. Read in all the other data blocks, excluding the parity block.
  2. Recreate the parity block from the other data blocks and the data to be written.
  3. Write the parity block.
- *Parityless write*, a write where the parity block is not available. This is in fact the simplest: just write the data blocks. This can proceed directly from the user buffer.

Combining access strategies

In practice, a transfer request may combine the actions above. In particular:

- A read request may request reading both available data (normal read) and non-available data (recovery read). This can be a problem if the address ranges of the two reads do not coincide: the normal read must be extended to cover the address range of the recovery read, and must thus be performed out of malloced memory.
- Combination of degraded data block write and normal write. The address ranges of the reads may also need to be extended to cover all participating blocks.

An exception exists when the transfer is shorter than the width of the stripe and is spread over two subdisks. In this case, the subdisk addresses do not overlap, so they are effectively two separate requests.

Examples

The following examples illustrate these concepts:

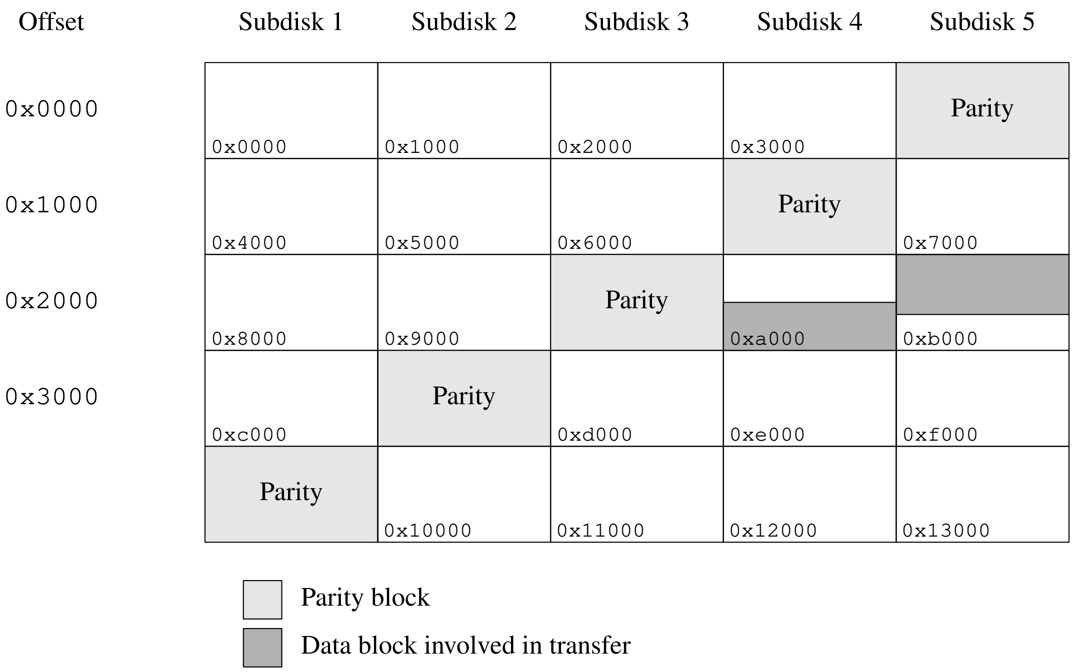


Figure 8: A sample RAID-5 transfer

This diagram illustrates a number of typical points about RAID-5 transfers. It shows the beginning of a plex with five subdisks and a stripe size of 4 kB. The shaded area shows the area involved in a transfer of 4.5 kB (9 sectors), starting at offset `0xa800` in the plex. A read of this area generates two requests to the lower-level driver: 4 sectors from subdisk 4, starting at offset `0x2800`, and 5 sectors from subdisk 5, starting at offset `0x2000`.

Writing this area is significantly more complicated. From a programming standpoint, the simplest approach is to consider the transfers individually. This would create the following requests:

- Read the old contents of 4 sectors from subdisk 4, starting at offset `0x2800`.
- Read the old contents of 4 sectors from subdisk 3 (the parity disk), starting at offset `0x2800`.
- Perform an exclusive OR of the data read from subdisk 4 with the data read from subdisk 3, storing the result in subdisk 3's data buffer. This effectively ‘‘removes’’ the old data from the parity block.
- Perform an exclusive OR of the data to be written to subdisk 4 with the data read from subdisk 3, storing the result in subdisk 3's data buffer. This effectively ‘‘adds’’ the new data to the parity block.
- Write the new data to 4 sectors of subdisk 4, starting at offset `0x2800`.
- Write 4 sectors of new parity data to subdisk 3 (the parity disk), starting at offset `0x2800`.

- Read the old contents of 5 sectors from subdisk 5, starting at offset `0x2000`.
- Read the old contents of 5 sectors from subdisk 3 (the parity disk), starting at offset `0x2000`.
- Perform an exclusive OR of the data read from subdisk 5 with the data read from subdisk 3, storing the result in subdisk 3's data buffer. This effectively ''removes'' the old data from the parity block.
- Perform an exclusive OR of the data to be written to subdisk 5 with the data read from subdisk 3, storing the result in subdisk 3's data buffer. This effectively ''adds'' the new data to the parity block.
- Write the new data to 5 sectors of subdisk 5, starting at offset `0x2000`.
- Write 5 sectors of new parity data to subdisk 3 (the parity disk), starting at offset `0x2000`.

This approach is clearly suboptimal. The operation involves a total of 8 I/O operations and transfers 36 sectors of data. In addition, the two halves of the operation block each other, since each must access the same data on the parity subdisk. *Vinum* optimizes this access in the following manner:

- Read the old contents of 4 sectors from subdisk 4, starting at offset `0x2800`.
- Read the old contents of 5 sectors from subdisk 5, starting at offset `0x2000`.
- Read the old contents of 8 sectors from subdisk 3 (the parity disk), starting at offset `0x2000`. This represents the complete parity block for the stripe.
- Perform an exclusive OR of the data read from subdisk 4 with the data read from subdisk 3, starting at offset `0x800` into the buffer, and storing the result in the same place in subdisk 3's data buffer.
- Perform an exclusive OR of the data read from subdisk 5 with the data read from subdisk 3, starting at the beginning of the buffer, and storing the result in the same place in subdisk 3's data buffer offset.
- Perform an exclusive OR of the data to be written to subdisk 4 with the modified parity block, starting at offset `0x800` into the buffer, and storing the result in the same place in subdisk 3's data buffer.
- Perform an exclusive OR of the data to be written to subdisk 5 with the modified parity block, starting at the beginning of the buffer, and storing the result in the same place in subdisk 3's data buffer offset.
- Write the new data to 4 sectors of subdisk 4, starting at offset `0x2800`.
- Write the new data to 5 sectors of subdisk 5, starting at offset `0x2000`.
- Write the 8 sectors of new parity data to subdisk 3 (the parity disk), starting at offset `0x2000`.

This is still a lot of work, but by comparison with the non-optimized version, the number of I/O operations has been reduced to 6, and the number of sectors transferred is reduced by 2. The larger the overlap, the greater the saving. If the request had been for a total of 17 sectors, starting at offset `0x9800`, the unoptimized version would have performed 12 I/O operations and moved a total of 68 sectors, while the optimized version would perform 8 I/O operations and move a total of 50 sectors.

Degraded read

The following figure illustrates the situation where a data subdisk fails, in this case

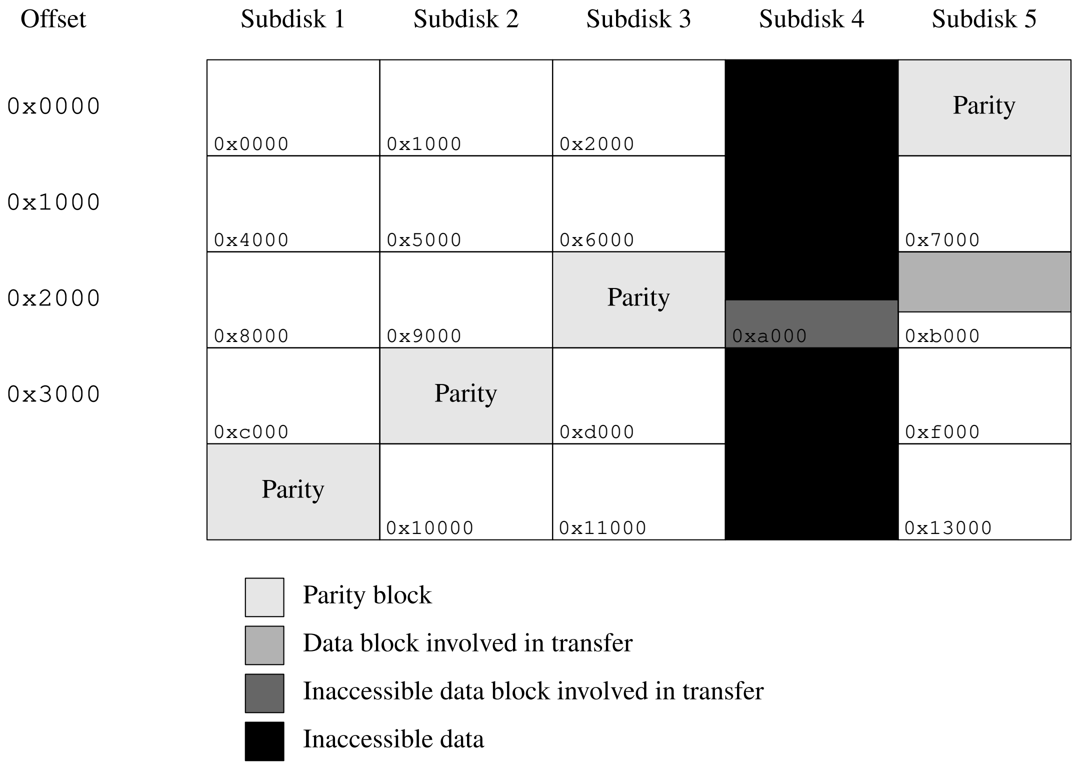


Figure 9: RAID-5 transfer with inaccessible data block

subdisk 4.

In this case, reading the data from subdisk 5 is trivial. Recreating the data from subdisk 4, however, requires reading all the remaining subdisks. Specifically,

- Read 4 sectors each from subdisks 1, 2 and 3, starting at offset `0x2800` in each case.
- Read 8 sectors from subdisk 5, starting at offset `0x2800`.
- Clear the user buffer area for the data corresponding to subdisk 4.
- Perform an "exclusive or" operation on this data buffer with data from subdisks 1, 2, 3, and the last four sectors of the data from subdisk 5.
- Transfer the first 5 sectors of data from the data buffer for subdisk 5 to the corresponding place in the user data buffer.

Degraded write

There are two different scenarios to be considered in a degraded write. Referring to the previous example, the operations required are a mixture of normal write (for subdisk 5) and degraded write (for subdisk 4). In detail, the operations are:

- Read 4 sectors each from subdisks 1 and 2, starting at offset `0x2800`, into temporary storage.
- Read 5 sectors from subdisk 3 (parity block), starting at offset `0x2000`, into the beginning of an 8 sector temporary storage buffer.
- Clear the last 3 sectors of the parity block.
- Read 8 sectors from subdisk 5, starting at offset `0x2000`, into temporary storage.
- "Remove" the first 5 sectors of subdisk 5 data from the parity block with exclusive or.
- Rebuild the last 3 sectors of the parity block by exclusive or of the corresponding data from subdisks 1, 2, 5 and the data to be written for subdisk 4.
- Write the parity block back to subdisk 3 (8 sectors).
- Write 5 sectors user data to subdisk 5.

Parityless write

Another situation arises when the subdisk containing the parity block fails:

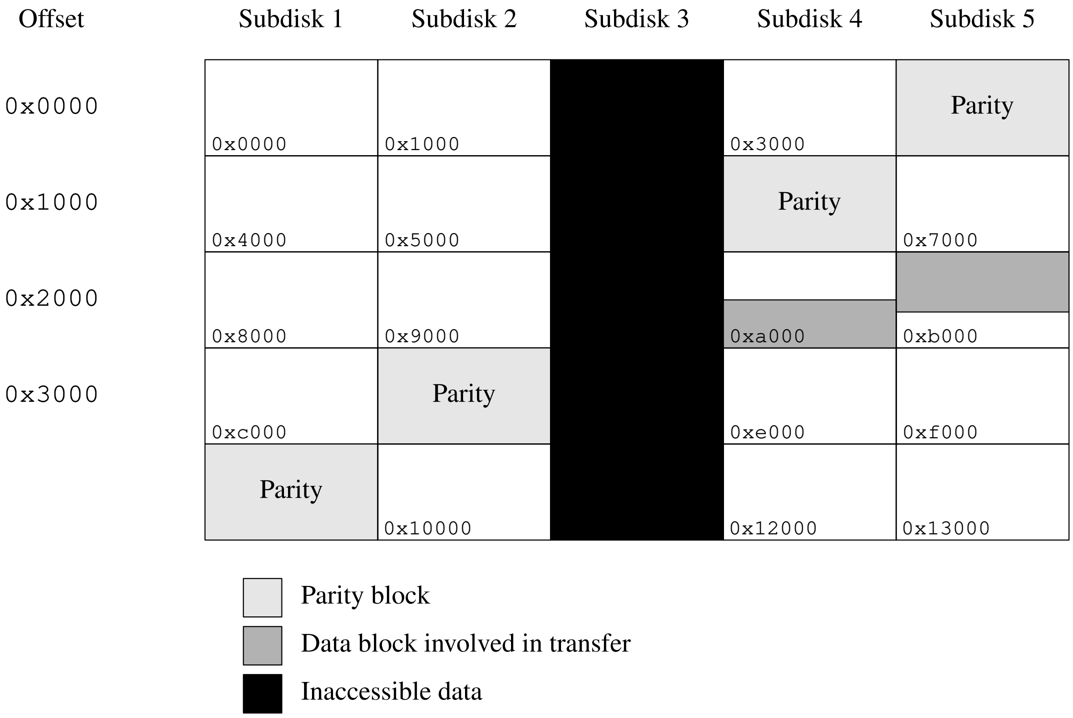


Figure 10: RAID-5 transfer with inaccessible parity block

This configuration poses no problems on reading, since all the data is accessible. On writing, however, it is not possible to write the parity block. It is not possible to recover from this problem at the time of the write, so the write operation simplifies to writing only the data blocks. The parity block will be recreated when the subdisk is brought up again.

Driver structure

One important detail of the nature of the operations which must be performed for RAID-5 access is that they frequently must be performed in two steps. This does not match well with the design of UNIX device drivers: typically, the "top half"[3] of a UNIX device driver issues I/O commands and returns to the caller. The caller may choose to wait for completion, but one of the most frequent uses of a block device is where the virtual memory subsystem issues writes and does not wait for completion.

This poses a problem: who issues the second set of requests? The following possibilities, listed in order of increasing desirability, exist:

1. The top half can wait for completion of the first set of requests and then launch the second set before returning to the caller. This approach can seriously impact system performance and possibly cause deadlocks.
2. In a threaded kernel, the strategy routine can create a thread which waits for completion of the first set of requests and starts the second set without impacting the main thread of the process. At the moment this approach is not possible, since FreeBSD currently does not provide kernel thread support. It also appears likely that it could cause a number of problems in the areas of thread synchronization and performance.
3. Ownership of the requests can be "given" to another process, which will be awakened when they complete. This process can then issue the second set of requests. This approach is feasible, and it is used by some subsystems, notably NFS. It does not pose the same severe performance penalty of the previous possibility, but it does require that another process be scheduled twice for every I/O.
4. The second set of requests can be launched from the "bottom half" of the driver. This is potentially dangerous: the interrupt routine must call the `start` routine. While this is not expressly prohibited, the `start` routine is normally used by the top half of a driver, and may call functions which are prohibited in the bottom half.

Currently, *Vinum* uses the fourth solution. This works for most drivers, but not for the Adaptec 154x driver on a system with more than 16 MB memory: since the 154x is an ISA bus master device, the driver must allocate bounce buffers on machines with more than 16 MB memory. The driver allocates these buffers by calling `malloc`, which calls `tsleep` if memory is not available. As a result, *Vinum* cannot be used on a system with

3. UNIX device drivers run in two separate environments. The "top half" runs in the process context, while the "bottom half" runs in the interrupt context. There are severe restrictions on the functions that the bottom half of the driver can perform.

an Adaptec 154x and more than 16 MB of memory.

It is possible that this deficiency, possibly with others like it, will lead to a change in the driver structure; given the current alternatives, this would mean a dæmon process to handle the I/O.

Performance issues

At present no detailled performance measurements have been made, but indications are that the performance is very close to what could be expected from the underlying disk driver performing the same operations as *Vinum* performs: in other words, the overhead of *Vinum* itself is negligible. This does not mean that *Vinum* has perfect performance: the choice of requests has a strong impact on the overall subsystem performance, and there are some known areas which could be improved upon. In addition, the user can influence performance by the design of the volumes.

The following sections examine some factors which influence performance.

The influence of stripe size

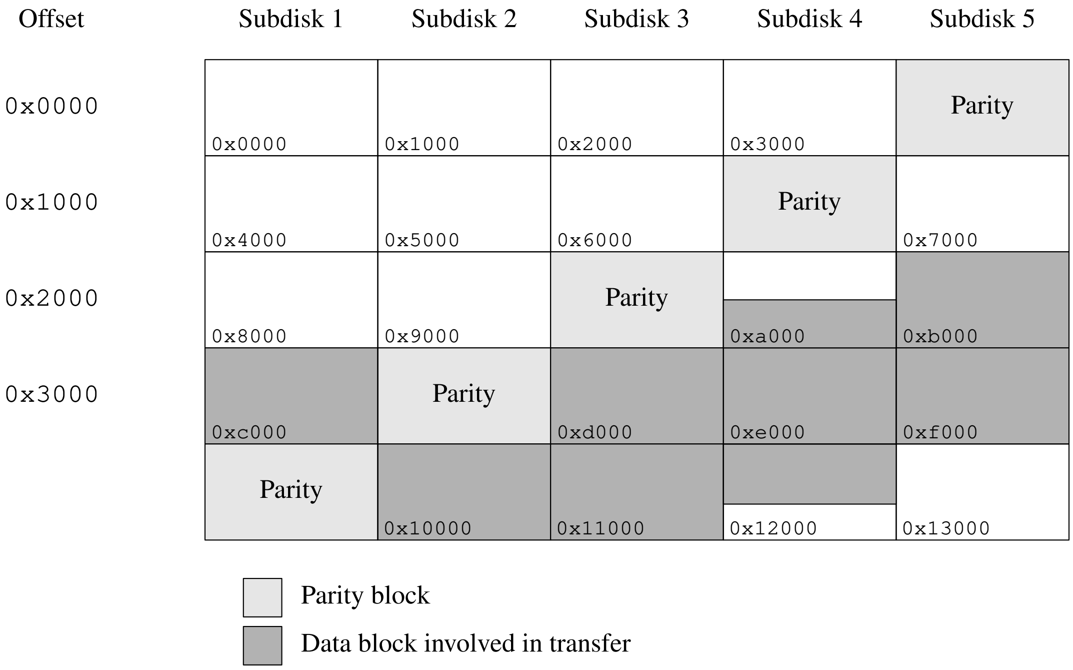
In striped and RAID-5 plexes, the stripe size has a significant influence on performance. In all plex structures except a single-subdisk plex (which by definition is concatenated), the possibility exists that a single transfer to or from a volume will be remapped into more than one physical I/O request. This is never desirable in a system without spindle synchronization, since the average latency for multiple transfers is always larger than the average latency for single transfers to the same kind of disk hardware. Within the bounds of the current BSD I/O architecture (maximum transfer size 128 kB), this increase in latency can easily offset any speed increase in the transfer. This is the main reason why *Vinum* does not implement RAID-2 and RAID-3, which always transfer to all drives.

In the case of a concatenated plex, this remapping occurs only when a request overlaps a subdisk boundary. In a striped or RAID-5 plex, however, the probability is an inverse function of the stripe size. For this reason, a stripe size of 256 kB appears to be optimum: it is small enough to create a relatively random mapping of file system hot spots to individual disks, and large enough to ensure than 95% of all transfers involve only a single data subdisk. Preliminary testing has confirmed this recommendation.

The influence of request structure

For concatenated and striped plexes, *Vinum* creates request structures which map directly to the user-level request buffers. The only additional overhead is the allocation of the request structure, and the possibility of improvement is correspondingly small.

With RAID-5 plexes, the picture is very different. The strategic choices described above work well when the total request size is less than the stripe width. By contrast, consider the following transfer of 32.5 kB, starting from the same offset as the previous examples:



An optimum approach to reading this data performs a total of 5 I/O operations, one on each subdisk. By contrast, *Vinum* treats this transfer as three separate transfers, one per stripe, and thus performs a total of 9 I/O transfers.

In practice, this inefficiency should not cause any problems: as discussed above, the optimum stripe size is larger than the maximum transfer size, so this situation does not arise when an appropriate stripe size is chosen.

Availability

Vinum is currently under development. An alpha version of the base version (without RAID-5 functionality), running on the FreeBSD operating system, is available under a Berkeley-style copyright at [vinum]. The RAID-5 functionality is available under licence from Cybernet, Inc. [Cybernet], and is included in their *NetMAX* Internet connection package.

Future directions

The current version of *Vinum* implements the core functionality. A number of additional features are under consideration:

- *Hot spare* capability: on the failure of a disk drive, the volume manager automatically recovers the data to another drive.
- *Logging* changes to a degraded volume. Rebuilding a plex usually requires copying the entire volume. In a volume with a high read to write, if a disk goes down temporarily and then becomes accessible again (for example, as the result of controller

failure), most of the data is already present and does not need to be copied. Logging pinpoints which blocks require copying in order to bring the stale plex up to date.

- *Snapshots* of a volume. It is often useful to freeze the state of a volume, for example for backup purposes. A backup of a large volume can take several hours. It can be inconvenient or impossible to prohibit updates during this time. A snapshot solves this problem by maintaining *before images*, a copy of the old contents of the modified data blocks. Access to the plex reads the blocks from the snapshot plex if it contains the data, and from another plex if it does not.

  Implementing snapshots in *Vinum* alone would solve only part of the problem: there must also be a way to ensure that the data on the file system is consistent from a user standpoint when the snapshot is taken. This task involves such components as file systems and databases and is thus outside the scope of *Vinum*.

- A *SNMP interface* for central management of *Vinum* systems.
- A *GUI* interface is currently *not* planned, though it is relatively simple to program, since no kernel code is needed. As the number of failures testify, a good GUI interface is apparently very difficult to write, and it tends to gloss over important administrative aspects, so it's not clear that the advantages justify the effort. On the other hand, a graphical output of the configuration could be of advantage.
- An *extensible UFS*. It is possible to extend the size of some modern file systems after they have been created. Although UFS (the *UNIX File System*, previously called the Berkeley *Fast File System*) was not designed for such extension, it is trivial to implement extensibility. This feature would allow a user to add space to a file system which is approaching capacity by first adding subdisks to the plexes and then extending the file system.
- *Remote data replication* is of interest either for backup purposes or for read-only access at a remote site. From a conceptual viewpoint, it could be achieved by interfacing to a network driver instead of a local disk driver.
- *Extending striped and RAID-5 plexes* is a slow complicated operation, but it is feasible.

References

[CMD] CMD Technology, Inc June 1993, *The Need For RAID, An Introduction*. *http://www.fdma.com/info/raidinto.html*

[Cybernet] *The NetMAX Station*, *http://www.cybernet.com/netmax/index.html*. The first product using the *Vinum* Volume Manager.

[FreeBSD] FreeBSD home page, *http://www.FreeBSD.org/*

[IBM] *AIX Version 4.3 System Management Guide: Operating System and Devices, Logical Volume Storage Overview*
http://www.austin.ibm.com/doc_link/en_US/a_doc_lib/aixbman/baseadmn/lvm_overview.htm

[Linux] *RAID Solutions for Linux*, *http://linas.org/linux/raid.html*

[McKusick] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, 1996.

[OpenSource] *The Open Source Page*, *http://www.opensource.org/*

[SCO] *SCO Virtual Disk Manager*, *http://www.sco.com/products/layered/ras/virtual.html*.

[Solstice] *http://www.sun.com/solstice/em-products/system/disksuite.html*

[vinum] Greg Lehey, *The Vinum Volume Manager*, *http://www.lemis.com/vinum.html*

[Wong] Brian Wong, *RAID: What does it mean to me?*, SunWorld Online, September 1995. *http://www.sunworld.com/sunworldonline/swol-09-1995/swol-09-raid5.html*