The vinum Volume Manager

Table of Contents

1. Synopsis . . . 1
2. Access Bottlenecks . . . 2
3. Data Integrity . . . 3
4. vinum Objects . . . 4
5. Some Examples . . . 6
6. Object Naming . . . 12
7. Configuring vinum . . . 14
8. Using vinum for the Root File System . . . 15

1. Synopsis

No matter the type of disks, there are always potential problems. The disks can be too small, too slow, or too unreliable to meet the system's requirements. While disks are getting bigger, so are data storage requirements. Often a file system is needed that is bigger than a disk's capacity. Various solutions to these problems have been proposed and implemented.

One method is through the use of multiple, and sometimes redundant, disks. In addition to supporting various cards and controllers for hardware Redundant Array of Independent Disks RAID systems, the base FreeBSD system includes the vinum volume manager, a block device driver that implements virtual disk drives and addresses these three problems. vinum provides more flexibility, performance, and reliability than traditional disk storage and implements RAID-0, RAID-1, and RAID-5 models, both individually and in combination.

This chapter provides an overview of potential problems with traditional disk storage, and an introduction to the vinum volume manager.

vinum is deprecated and is not present in FreeBSD 15.0 and later. Users are advised to migrate to gconcat(8), gmirror(8), gstripe(8), graid(8), or zfs(8).

Starting with FreeBSD 5, vinum has been rewritten to fit into the GEOM architecture, while retaining the original ideas, terminology, and on-disk metadata. This rewrite is called *gvinum* (for *GEOM vinum*). While this chapter uses the term vinum, any command invocations should be performed with `gvinum`. The name of the kernel module has changed from the original vinum.ko to geom_vinum.ko, and all device nodes reside under /dev/gvinum instead of /dev/vinum. As of FreeBSD 6, the original vinum implementation is no longer available in the code base.

2. Access Bottlenecks

Modern systems frequently need to access data in a highly concurrent manner. For example, large FTP or HTTP servers can maintain thousands of concurrent sessions and have multiple 100 Mbit/s connections to the outside world, well beyond the sustained transfer rate of most disks.

Current disk drives can transfer data sequentially at up to 70 MB/s, but this value is of little importance in an environment where many independent processes access a drive, and where they may achieve only a fraction of these values. In such cases, it is more interesting to view the problem from the viewpoint of the disk subsystem. The important parameter is the load that a transfer places on the subsystem, or the time for which a transfer occupies the drives involved in the transfer.

In any disk transfer, the drive must first position the heads, wait for the first sector to pass under the read head, and then perform the transfer. These actions can be considered to be atomic as it does not make any sense to interrupt them.

Consider a typical transfer of about 10 kB: the current generation of high-performance disks can position the heads in an average of 3.5 ms. The fastest drives spin at 15,000 rpm, so the average rotational latency (half a revolution) is 2 ms. At 70 MB/s, the transfer itself takes about 150 µs, almost nothing compared to the positioning time. In such a case, the effective transfer rate drops to a little over 1 MB/s and is clearly highly dependent on the transfer size.

The traditional and obvious solution to this bottleneck is "more spindles": rather than using one large disk, use several smaller disks with the same aggregate storage space. Each disk is capable of positioning and transferring independently, so the effective throughput increases by a factor close to the number of disks used.

The actual throughput improvement is smaller than the number of disks involved. Although each drive is capable of transferring in parallel, there is no way to ensure that the requests are evenly distributed across the drives. Inevitably the load on one drive will be higher than on another.

The evenness of the load on the disks is strongly dependent on the way the data is shared across the drives. In the following discussion, it is convenient to think of the disk storage as a large number of data sectors which are addressable by number, rather like the pages in a book. The most obvious method is to divide the virtual disk into groups of consecutive sectors the size of the individual physical disks and store them in this manner, rather like taking a large book and tearing it into smaller sections. This method is called *concatenation* and has the advantage that the disks are not required to have any specific size relationships. It works well when the access to the virtual disk is spread evenly about its address space. When access is concentrated on a smaller area, the improvement is less marked. Concatenated Organization illustrates the sequence in which storage units are allocated in a concatenated organization.

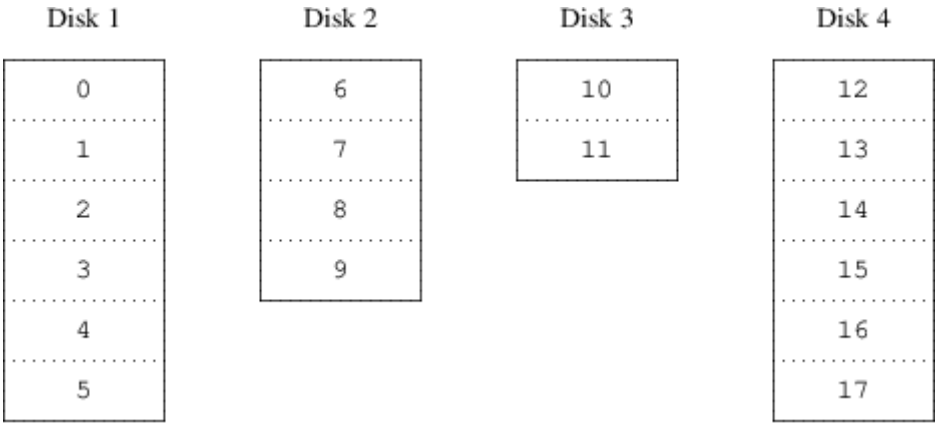

Figure 1. Concatenated Organization

An alternative mapping is to divide the address space into smaller, equal-sized components and store them sequentially on different devices. For example, the first 256 sectors may be stored on the first disk, the next 256 sectors on the next disk and so on. After filling the last disk, the process repeats until the disks are full. This mapping is called *striping* or RAID-0.

RAID offers various forms of fault tolerance, though RAID-0 is somewhat misleading as it provides no redundancy. Striping requires somewhat more effort to locate the data, and it can cause additional I/O load where a transfer is spread over multiple disks, but it can also provide a more constant load across the disks. Striped Organization illustrates the sequence in which storage units are allocated in a striped organization.

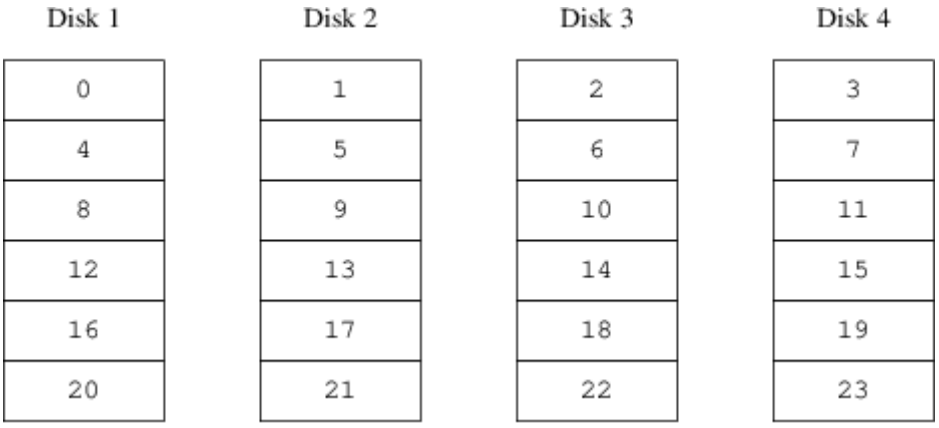

Figure 2. Striped Organization

3. Data Integrity

The final problem with disks is that they are unreliable. Although reliability has increased tremendously over the last few years, disk drives are still the most likely core component of a server to fail. When they do, the results can be catastrophic and replacing a failed disk drive and restoring data can result in server downtime.

One approach to this problem is *mirroring*, or RAID-1, which keeps two copies of the data on different physical hardware. Any write to the volume writes to both disks; a read can be satisfied from either, so if one drive fails, the data is still available on the other drive.

Mirroring has two problems:

- It requires twice as much disk storage as a non-redundant solution.
- Writes must be performed to both drives, so they take up twice the bandwidth of a non-mirrored volume. Reads do not suffer from a performance penalty and can even be faster.

An alternative solution is *parity*, implemented in RAID levels 2, 3, 4 and 5. Of these, RAID-5 is the most interesting. As implemented in vinum, it is a variant on a striped organization which dedicates one block of each stripe to parity one of the other blocks. 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. The numbers in the data blocks indicate the relative block numbers.

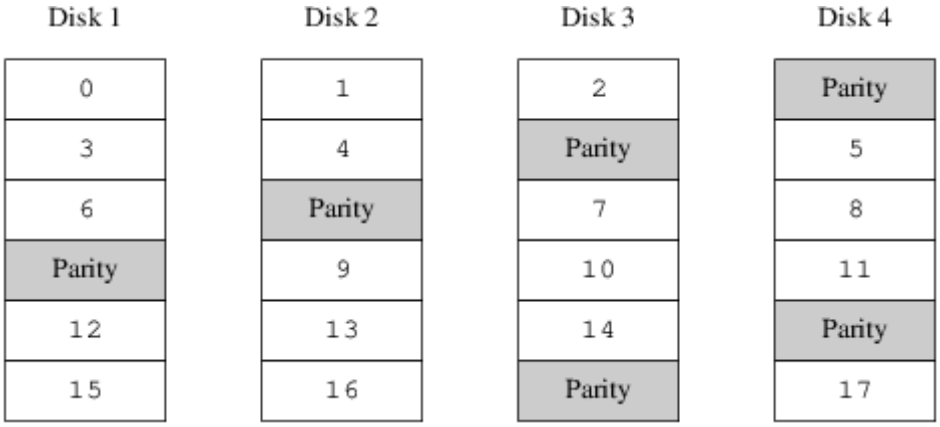

Figure 3. RAID-5 Organization

Compared to mirroring, RAID-5 has the advantage of requiring significantly less storage space. Read access is similar to that of striped organizations, but write access is significantly slower, approximately 25% of the read performance. If one drive fails, the array can continue to operate in degraded mode where a read from one of the remaining accessible drives continues normally, but a read from the failed drive is recalculated from the corresponding block from all the remaining drives.

4. vinum Objects

To address these problems, vinum implements a four-level hierarchy of objects:

- The most visible object is the virtual disk, called a *volume*. Volumes have essentially the same properties as a UNIX® disk drive, though there are some minor differences. For one, they have no size limitations.
- Volumes are composed of *plexes*, each of which represent the total address space of a volume. This level in the hierarchy provides redundancy. Think of plexes as individual disks in a mirrored array, each containing the same data.
- Since vinum exists within the UNIX® disk storage framework, it would be possible to use UNIX® partitions as the building block for multi-disk plexes. In fact, this turns out to be too inflexible as UNIX® disks can have only a limited number of partitions. Instead, vinum subdivides a single UNIX® partition, the *drive*, into contiguous areas called *subdisks*, which are used as building blocks for plexes.
- Subdisks reside on vinum*drives*, currently UNIX® partitions. vinum drives can contain any

number of subdisks. With the exception of a small area at the beginning of the drive, which is used for storing configuration and state information, the entire drive is available for data storage.

The following sections describe the way these objects provide the functionality required of vinum.

4.1. Volume Size Considerations

Plexes can include multiple subdisks spread over all drives in the vinum configuration. As a result, the size of an individual drive does not limit the size of a plex or a volume.

4.2. Redundant Data Storage

vinum implements mirroring by attaching multiple plexes to a volume. Each plex is a representation of the data in a volume. A volume may contain between one and eight plexes.

Although a plex represents the complete data of a volume, it is possible for parts of the representation to be physically missing, either by design (by not defining a subdisk for parts of the plex) or by accident (as a result of the failure of a drive). As long as at least one plex can provide the data for the complete address range of the volume, the volume is fully functional.

4.3. Which Plex Organization?

vinum implements both concatenation and striping at the plex level:

- A *concatenated plex* uses the address space of each subdisk in turn. Concatenated plexes are the most flexible as 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 plexes, though the difference in CPU overhead is not measurable. On the other hand, they are most susceptible to hot spots, where one disk is very active and others are idle.
- A *striped plex* stripes the data across each subdisk. The subdisks must all be the same size and there must be at least two subdisks to distinguish it from a concatenated plex. The greatest advantage of striped plexes is that they reduce hot spots. By choosing an optimum sized stripe, about 256 kB, the load can be evened out on the component drives. Extending a plex by adding new subdisks is so complicated that vinum does not implement it.

vinum Plex Organizations summarizes the advantages and disadvantages of each plex organization.

Table 1. vinum Plex Organizations

Plex type	Minimum subdisks	Can add subdisks	Must be equal size	Application
concatenated	1	yes	no	Large data storage with maximum placement flexibility and moderate performance
striped	2	no	yes	High performance in combination with highly concurrent access

5. Some Examples

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 using gvinum(8). vinum stores a copy of its configuration database on each disk *device* under its control. This database is updated on each state change, so that a restart accurately restores the state of each vinum object.

5.1. The Configuration File

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

```
drive a device /dev/da3h
volume myvol
  plex org concat
    sd length 512m drive a
```

This file describes four vinum objects:

- The *drive* line describes a disk partition (*drive*) and its location relative to the underlying hardware. It is given the symbolic name *a*. This separation of symbolic names from device names allows disks to be moved from one location to another without confusion.
- The *volume* line describes a volume. The only required attribute is the name, in this case *myvol*.
- The *plex* line defines a plex. The only required parameter is the organization, in this case *concat*. No name is necessary as the system automatically generates a name from the volume name by adding the suffix *.px*, where *x* is the number of the plex in the volume. Thus this plex will be called *myvol.p0*.
- The *sd* line describes a subdisk. The minimum specifications are the name of a drive on which to store it, and the length of the subdisk. No name is necessary as the system automatically assigns names derived from the plex name by adding the suffix *.sx*, where *x* is the number of

the subdisk in the plex. Thus vinum gives this subdisk the name *myvol.p0.s0*.

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

```
#  gvinum -> create config1
Configuration summary
Drives:         1 (4 configured)
Volumes:        1 (4 configured)
Plexes:         1 (8 configured)
Subdisks:       1 (16 configured)

  D a                     State: up       Device /dev/da3h      Avail: 2061/2573 MB
(80%)

  V myvol                 State: up       Plexes:       1 Size:      512 MB

  P myvol.p0            C State: up       Subdisks:     1 Size:      512 MB

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

This output shows the brief listing format of gvinum(8). It is represented graphically in A Simple vinum Volume.

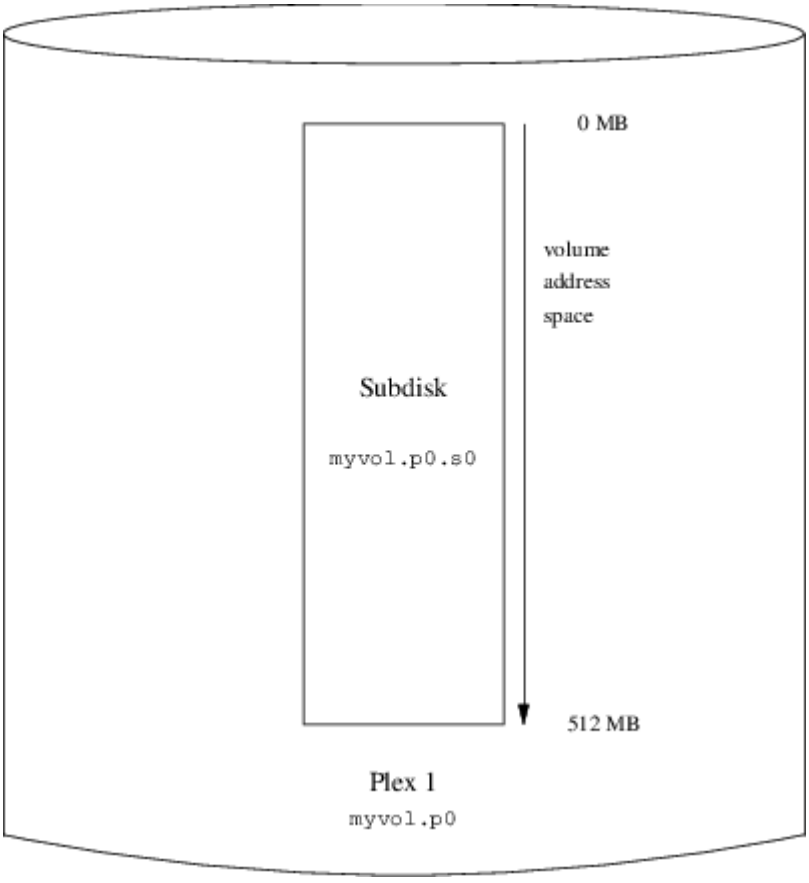

Figure 4. A Simple vinum Volume

This figure, and the ones which follow, represent a volume, which contains the plexes, which in turn contains the subdisks. In this example, the volume contains one plex, and the plex contains

one subdisk.

This particular volume has no specific advantage over a conventional disk partition. It contains a single plex, so it is not redundant. The plex contains a single subdisk, so there is no difference in storage allocation from a conventional disk partition. The following sections illustrate various more interesting configuration methods.

5.2. Increased Resilience: Mirroring

The resilience of a volume can be increased by mirroring. When laying out a mirrored volume, it is important to ensure that the subdisks of each plex are on different drives, so that a drive failure will not take down both plexes. The following configuration mirrors a volume:

```
    drive b device /dev/da4h
    volume mirror
      plex org concat
        sd length 512m drive a
      plex org concat
        sd length 512m drive b
```

In this example, it was not necessary to specify a definition of drive *a* again, since vinum keeps track of all objects in its configuration database. After processing this definition, the configuration looks like:

```
    Drives:         2 (4 configured)
    Volumes:        2 (4 configured)
    Plexes:         3 (8 configured)
    Subdisks:       3 (16 configured)

    D a                     State: up       Device /dev/da3h       Avail: 1549/2573 MB
(60%)
    D b                     State: up       Device /dev/da4h       Avail: 2061/2573 MB
(80%)

    V myvol                 State: up       Plexes:       1 Size:        512 MB
    V mirror                State: up       Plexes:       2 Size:        512 MB

    P myvol.p0            C State: up       Subdisks:     1 Size:        512 MB
    P mirror.p0           C State: up       Subdisks:     1 Size:        512 MB
    P mirror.p1           C State: initializing     Subdisks:     1 Size:        512
MB

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

A Mirrored vinum Volume shows the structure graphically.

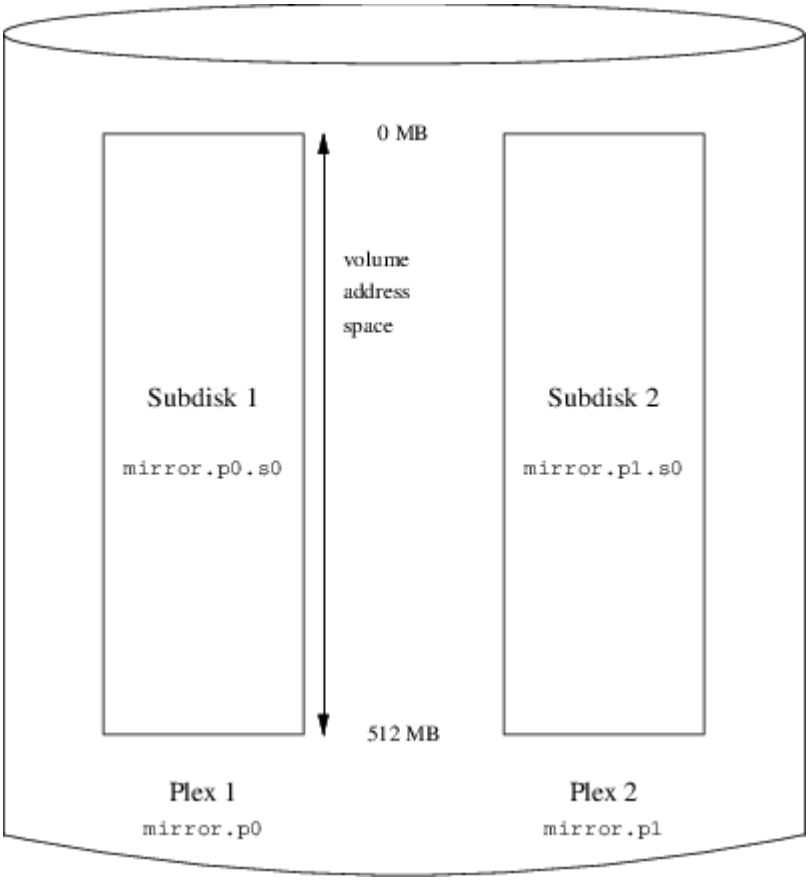

Figure 5. A Mirrored vinum Volume

In this example, each plex contains the full 512 MB of address space. As in the previous example, each plex contains only a single subdisk.

5.3. Optimizing Performance

The mirrored volume in the previous example is more resistant to failure than an unmirrored volume, but its performance is less as each write to the volume requires a write to both drives, using up a greater proportion of the total disk bandwidth. Performance considerations demand a different approach: instead of mirroring, the data is striped across as many disk drives as possible. The following configuration shows a volume with a plex striped across four disk drives:

```
    drive c device /dev/da5h
drive d device /dev/da6h
volume stripe
plex org striped 512k
  sd length 128m drive a
  sd length 128m drive b
  sd length 128m drive c
  sd length 128m drive d
```

As before, it is not necessary to define the drives which are already known to vinum. After processing this definition, the configuration looks like:

```
Drives:         4 (4 configured)
```

```
    Volumes:        3 (4 configured)
    Plexes:         4 (8 configured)
    Subdisks:       7 (16 configured)

    D a                     State: up       Device /dev/da3h        Avail: 1421/2573
MB (55%)
    D b                     State: up       Device /dev/da4h        Avail: 1933/2573
MB (75%)
    D c                     State: up       Device /dev/da5h        Avail: 2445/2573
MB (95%)
    D d                     State: up       Device /dev/da6h        Avail: 2445/2573
MB (95%)

    V myvol                 State: up       Plexes:       1 Size:        512 MB
    V mirror                State: up       Plexes:       2 Size:        512 MB
    V striped               State: up       Plexes:       1 Size:        512 MB

    P myvol.p0            C State: up       Subdisks:     1 Size:        512 MB
    P mirror.p0           C State: up       Subdisks:     1 Size:        512 MB
    P mirror.p1           C State: initializing     Subdisks:     1 Size:        512
MB
    P striped.p1            State: up       Subdisks:     1 Size:        512 MB

    S myvol.p0.s0           State: up       PO:        0  B Size:        512 MB
    S mirror.p0.s0          State: up       PO:        0  B Size:        512 MB
    S mirror.p1.s0          State: empty    PO:        0  B Size:        512 MB
    S striped.p0.s0         State: up       PO:        0  B Size:        128 MB
    S striped.p0.s1         State: up       PO:      512 kB Size:        128 MB
    S striped.p0.s2         State: up       PO:     1024 kB Size:        128 MB
    S striped.p0.s3         State: up       PO:     1536 kB Size:        128 MB
```

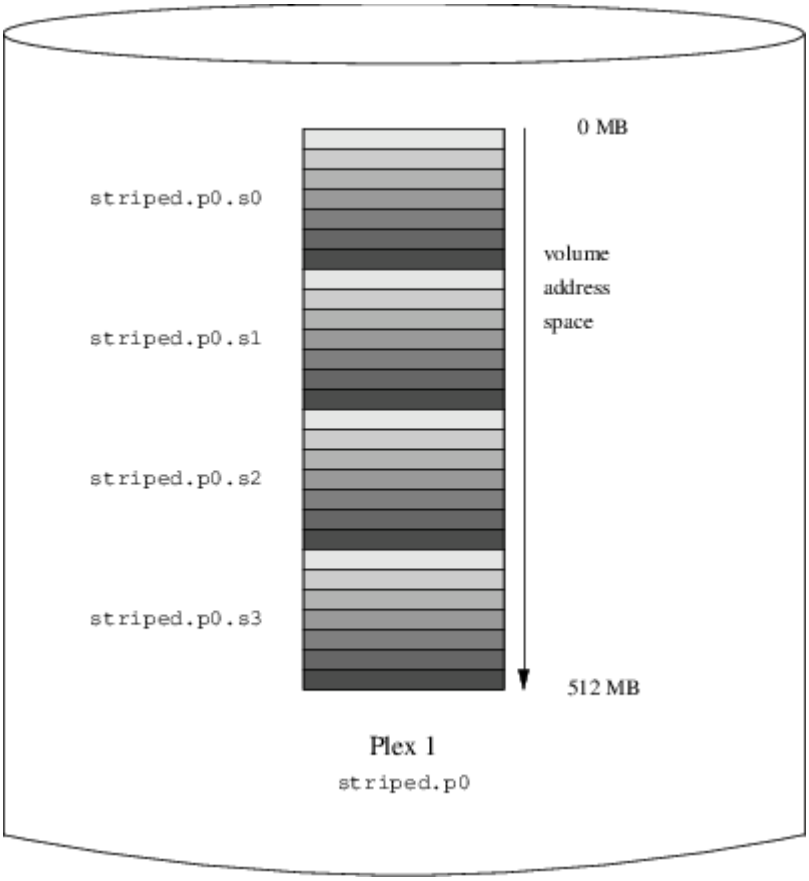

Figure 6. A Striped vinum Volume

This volume is represented in A Striped vinum Volume. The darkness of the stripes indicates the position within the plex address space, where the lightest stripes come first and the darkest last.

5.4. Resilience and Performance

With sufficient hardware, it is possible to build volumes which show both increased resilience and increased performance compared to standard UNIX® partitions. A typical configuration file might be:

```
volume raid10
  plex org striped 512k
    sd length 102480k drive a
    sd length 102480k drive b
    sd length 102480k drive c
    sd length 102480k drive d
    sd length 102480k drive e
  plex org striped 512k
    sd length 102480k drive c
    sd length 102480k drive d
    sd length 102480k drive e
    sd length 102480k drive a
    sd length 102480k drive b
```

The subdisks of the second plex are offset by two drives from those of the first plex. This helps to ensure that writes do not go to the same subdisks even if a transfer goes over two drives.

A Mirrored represents the structure of this volume.

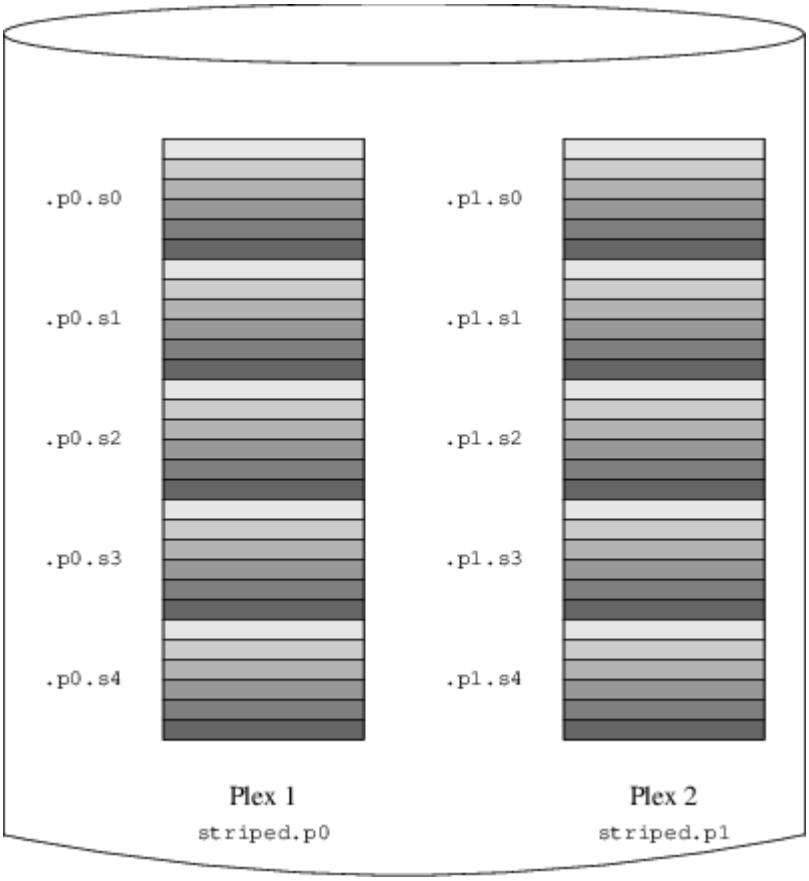

Figure 7. A Mirrored, Striped vinum Volume

6. Object Naming

vinum assigns default names to plexes and subdisks, although they may be overridden. Overriding the default names is not recommended as it 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 be up to 32 characters long.

vinum objects are assigned device nodes in the hierarchy /dev/gvinum. The configuration shown above would cause vinum to create the following device nodes:

- Device entries for each volume. These are the main devices used by vinum. The configuration above would include the devices /dev/gvinum/myvol, /dev/gvinum/mirror, /dev/gvinum/striped, /dev/gvinum/raid5 and /dev/gvinum/raid10.
- All volumes get direct entries under /dev/gvinum/.
- The directories /dev/gvinum/plex, and /dev/gvinum/sd, which contain device nodes for each plex and for each subdisk, respectively.

For example, consider the following configuration file:

```
    drive drive1 device /dev/sd1h
    drive drive2 device /dev/sd2h
    drive drive3 device /dev/sd3h
    drive drive4 device /dev/sd4h
    volume s64 setupstate
      plex org striped 64k
        sd length 100m drive drive1
        sd length 100m drive drive2
        sd length 100m drive drive3
        sd length 100m drive drive4
```

After processing this file, gvinum(8) creates the following structure in /dev/gvinum:

```
    drwxr-xr-x  2 root  wheel       512 Apr 13
16:46 plex
    crwxr-xr--  1 root  wheel   91,   2 Apr 13 16:46 s64
    drwxr-xr-x  2 root  wheel       512 Apr 13 16:46 sd

    /dev/vinum/plex:
    total 0
    crwxr-xr--  1 root  wheel   25, 0x10000002 Apr 13 16:46 s64.p0

    /dev/vinum/sd:
    total 0
    crwxr-xr--  1 root  wheel   91, 0x20000002 Apr 13 16:46 s64.p0.s0
    crwxr-xr--  1 root  wheel   91, 0x20100002 Apr 13 16:46 s64.p0.s1
    crwxr-xr--  1 root  wheel   91, 0x20200002 Apr 13 16:46 s64.p0.s2
    crwxr-xr--  1 root  wheel   91, 0x20300002 Apr 13 16:46 s64.p0.s3
```

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.

6.1. Creating File Systems

Volumes appear to the system to be identical to disks, with one exception. Unlike UNIX® drives, vinum does not partition volumes, which thus do not contain a partition table. This has required modification to some disk utilities, notably newfs(8), so that it does not try to interpret the last letter of a vinum volume name as a partition identifier. For example, a disk drive may have a name like /dev/ad0a or /dev/da2h. These names represent the first partition (a) on the first (0) IDE disk (ad) and the eighth partition (h) on the third (2) SCSI disk (da) respectively. By contrast, a vinum volume might be called /dev/gvinum/concat, which has no relationship with a partition name.

To create a file system on this volume, use newfs(8):

```
# newfs /dev/gvinum/concat
```

7. Configuring vinum

The GENERIC kernel does not contain vinum. It is possible to build a custom kernel which includes vinum, but this is not recommended. The standard way to start vinum is as a kernel module. kldload(8) is not needed because when gvinum(8) starts, it checks whether the module has been loaded, and if it is not, it loads it automatically.

7.1. Startup

vinum stores configuration information on the disk slices in essentially the same form as in the configuration files. When reading from the configuration database, vinum recognizes a number of keywords which are not allowed in the configuration files. For example, a disk configuration might contain the following text:

```
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 265b
plexoffset 0b
sd name myvol.p0.s1 drive b plex myvol.p0 state up len 1048576b driveoffset 265b
plexoffset 1048576b
sd name myvol.p1.s0 drive c plex myvol.p1 state up len 1048576b driveoffset 265b
plexoffset 0b
sd name myvol.p1.s1 drive d plex myvol.p1 state up len 1048576b driveoffset 265b
plexoffset 1048576b
sd name myvol.p2.s0 drive a plex myvol.p2 state init len 524288b driveoffset 1048841b
plexoffset 0b
sd name myvol.p2.s1 drive b plex myvol.p2 state init len 524288b driveoffset 1048841b
plexoffset 524288b
sd name myvol.p2.s2 drive c plex myvol.p2 state init len 524288b driveoffset 1048841b
plexoffset 1048576b
sd name myvol.p2.s3 drive d plex myvol.p2 state init len 524288b driveoffset 1048841b
plexoffset 1572864b
sd name bigraid.p0.s0 drive a plex bigraid.p0 state initializing len 4194304b driveoff
set 1573129b plexoffset 0b
sd name bigraid.p0.s1 drive b plex bigraid.p0 state initializing len 4194304b driveoff
set 1573129b plexoffset 4194304b
sd name bigraid.p0.s2 drive c plex bigraid.p0 state initializing len 4194304b driveoff
set 1573129b plexoffset 8388608b
sd name bigraid.p0.s3 drive d plex bigraid.p0 state initializing len 4194304b driveoff
set 1573129b plexoffset 12582912b
sd name bigraid.p0.s4 drive e plex bigraid.p0 state initializing len 4194304b driveoff
set 1573129b plexoffset 16777216b
```

The obvious differences here are the presence of explicit location information and naming, both of

which are allowed but discouraged, and the information on the states. vinum does not store information about drives in the configuration information. It finds the drives by scanning the configured disk drives for partitions with a vinum label. This enables vinum to identify drives correctly even if they have been assigned different UNIX® drive IDs.

7.1.1. Automatic Startup

Gvinum always features an automatic startup once the kernel module is loaded, via loader.conf(5). To load the *Gvinum* module at boot time, add `geom_vinum_load="YES"` to /boot/loader.conf.

When vinum is started with `gvinum start`, 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. It then updates the configuration, if necessary, from progressively older drives.

8. Using vinum for the Root File System

For a machine that has fully-mirrored file systems using vinum, it is desirable to also mirror the root file system. Setting up such a configuration is less trivial than mirroring an arbitrary file system because:

- The root file system must be available very early during the boot process, so the vinum infrastructure must already be available at this time.
- The volume containing the root file system also contains the system bootstrap and the kernel. These must be read using the host system's native utilities, such as the BIOS, which often cannot be taught about the details of vinum.

In the following sections, the term "root volume" is generally used to describe the vinum volume that contains the root file system.

8.1. Starting up vinum Early Enough for the Root File System

vinum must be available early in the system boot as loader(8) must be able to load the vinum kernel module before starting the kernel. This can be accomplished by putting this line in /boot/loader.conf:

```
geom_vinum_load="YES"
```

8.2. Making a vinum-based Root Volume Accessible to the Bootstrap

The current FreeBSD bootstrap is only 7.5 KB of code and does not understand the internal vinum structures. This means that it cannot parse the vinum configuration data or figure out the elements

of a boot volume. Thus, some workarounds are necessary to provide the bootstrap code with the illusion of a standard `a` partition that contains the root file system.

For this to be possible, the following requirements must be met for the root volume:

- The root volume must not be a stripe or `RAID`-5.
- The root volume must not contain more than one concatenated subdisk per plex.

Note that it is desirable and possible to use multiple plexes, each containing one replica of the root file system. The bootstrap process will only use one replica for finding the bootstrap and all boot files, until the kernel mounts the root file system. Each single subdisk within these plexes needs its own `a` partition illusion, for the respective device to be bootable. It is not strictly needed that each of these faked `a` partitions is located at the same offset within its device, compared with other devices containing plexes of the root volume. However, it is probably a good idea to create the vinum volumes that way so the resulting mirrored devices are symmetric, to avoid confusion.

To set up these `a` partitions for each device containing part of the root volume, the following is required:

1. The location, offset from the beginning of the device, and size of this device's subdisk that is part of the root volume needs to be examined, using the command:

   ```
   # gvinum l -rv root
   ```

   vinum offsets and sizes are measured in bytes. They must be divided by 512 to obtain the block numbers that are to be used by `bsdlabel`.

2. Run this command for each device that participates in the root volume:

   ```
   # bsdlabel -e devname
   ```

   devname must be either the name of the disk, like da0 for disks without a slice table, or the name of the slice, like ad0s1.

   If there is already an `a` partition on the device from a pre-vinum root file system, it should be renamed to something else so that it remains accessible (just in case), but will no longer be used by default to bootstrap the system. A currently mounted root file system cannot be renamed, so this must be executed either when being booted from a "Fixit" media, or in a two-step process where, in a mirror, the disk that is not been currently booted is manipulated first.

   The offset of the vinum partition on this device (if any) must be added to the offset of the respective root volume subdisk on this device. The resulting value will become the `offset` value for the new `a` partition. The `size` value for this partition can be taken verbatim from the calculation above. The `fstype` should be `4.2BSD`. The `fsize`, `bsize`, and `cpg` values should be chosen to match the actual file system, though they are fairly unimportant within this context.

That way, a new `a` partition will be established that overlaps the vinum partition on this device. `bsdlabel` will only allow for this overlap if the vinum partition has properly been marked using the `vinum` fstype.

3. A faked `a` partition now exists on each device that has one replica of the root volume. It is highly recommendable to verify the result using a command like:

```
# fsck -n /dev/devnamea
```

It should be remembered that all files containing control information must be relative to the root file system in the vinum volume which, when setting up a new vinum root volume, might not match the root file system that is currently active. So in particular, /etc/fstab and /boot/loader.conf need to be taken care of.

At next reboot, the bootstrap should figure out the appropriate control information from the new vinum-based root file system, and act accordingly. At the end of the kernel initialization process, after all devices have been announced, the prominent notice that shows the success of this setup is a message like:

```
Mounting root from ufs:/dev/gvinum/root
```

8.3. Example of a vinum-based Root Setup

After the vinum root volume has been set up, the output of `gvinum l -rv root` could look like:

```
...
Subdisk root.p0.s0:
        Size:        125829120 bytes (120 MB)
        State: up
        Plex root.p0 at offset 0 (0  B)
        Drive disk0 (/dev/da0h) at offset 135680 (132 kB)

Subdisk root.p1.s0:
        Size:        125829120 bytes (120 MB)
        State: up
        Plex root.p1 at offset 0 (0  B)
        Drive disk1 (/dev/da1h) at offset 135680 (132 kB)
```

The values to note are `135680` for the offset, relative to partition /dev/da0h. This translates to 265 512-byte disk blocks in `bsdlabel's terms. Likewise, the size of this root volume is 245760 512-byte blocks. /dev/da1h, containing the second replica of this root volume, has a symmetric setup.

The bsdlabel for these devices might look like:

```
...
```

```
8 partitions:
#        size   offset    fstype   [fsize bsize bps/cpg]
  a:   245760      281    4.2BSD     2048 16384     0  # (Cyl.    0*- 15*)
  c: 71771688        0    unused        0     0        # (Cyl.    0 - 4467*)
  h: 71771672       16     vinum                      # (Cyl.    0*- 4467*)
```

It can be observed that the `size` parameter for the faked `a` partition matches the value outlined above, while the `offset` parameter is the sum of the offset within the vinum partition `h`, and the offset of this partition within the device or slice. This is a typical setup that is necessary to avoid the problem described in Nothing Boots. The entire `a` partition is completely within the `h` partition containing all the vinum data for this device.

In the above example, the entire device is dedicated to vinum and there is no leftover pre-vinum root partition.

8.4. Troubleshooting

The following list contains a few known pitfalls and solutions.

8.4.1. System Bootstrap Loads, but System Does Not Boot

If for any reason the system does not continue to boot, the bootstrap can be interrupted by pressing `space` at the 10-seconds warning. The loader variable `vinum.autostart` can be examined by typing `show` and manipulated using `set` or `unset`.

If the vinum kernel module was not yet in the list of modules to load automatically, type `load geom_vinum`.

When ready, the boot process can be continued by typing `boot -as` which `-as` requests the kernel to ask for the root file system to mount (`-a`) and make the boot process stop in single-user mode (`-s`), where the root file system is mounted read-only. That way, even if only one plex of a multi-plex volume has been mounted, no data inconsistency between plexes is being risked.

At the prompt asking for a root file system to mount, any device that contains a valid root file system can be entered. If /etc/fstab is set up correctly, the default should be something like `ufs:/dev/gvinum/root`. A typical alternate choice would be something like `ufs:da0d` which could be a hypothetical partition containing the pre-vinum root file system. Care should be taken if one of the alias `a` partitions is entered here, that it actually references the subdisks of the vinum root device, because in a mirrored setup, this would only mount one piece of a mirrored root device. If this file system is to be mounted read-write later on, it is necessary to remove the other plex(es) of the vinum root volume since these plexes would otherwise carry inconsistent data.

8.4.2. Only Primary Bootstrap Loads

If /boot/loader fails to load, but the primary bootstrap still loads (visible by a single dash in the left column of the screen right after the boot process starts), an attempt can be made to interrupt the primary bootstrap by pressing `space`. This will make the bootstrap stop in stage two. An attempt can be made here to boot off an alternate partition, like the partition containing the previous root file

system that has been moved away from `a`.

8.4.3. Nothing Boots, the Bootstrap Panics

This situation will happen if the bootstrap had been destroyed by the vinum installation. Unfortunately, vinum accidentally leaves only 4 KB at the beginning of its partition free before starting to write its vinum header information. However, the stage one and two bootstraps plus the bsdlabel require 8 KB. So if a vinum partition was started at offset 0 within a slice or disk that was meant to be bootable, the vinum setup will trash the bootstrap.

Similarly, if the above situation has been recovered, by booting from a "Fixit" media, and the bootstrap has been re-installed using `bsdlabel -B` as described in stage two, the bootstrap will trash the vinum header, and vinum will no longer find its disk(s). Though no actual vinum configuration data or data in vinum volumes will be trashed, and it would be possible to recover all the data by entering exactly the same vinum configuration data again, the situation is hard to fix. It is necessary to move the entire vinum partition by at least 4 KB, to have the vinum header and the system bootstrap no longer collide.