CHAPTER 2: EXPLORING VXVM

by Volker Herminghaus

2.1 Getting Started

Getting started with the actual Veritas Volume Manager product is easiest if you first gain an overview of the concepts behind Veritas Volume Manager (or VxVM in short), what virtual objects it uses and how they act together to form a complete volume management layer. After the extensive problem introduction in the previous chapter let us gain some insight into what VxVM does and how it does it. You will be intrigued by how straightforward and clear the design, yet how powerful and extensible the resulting architecture is.

In the following walkthrough you will encounter several commands which are not generally useful for everyday work. This is because they tend to be rather low-level commands which have long since been superseded by much more convenient commands that take much less parameters, preparation and time than the ones used in this chapter. We still like to use the inconvenient ones because they make it much clearer what exactly happens inside VxVM. But there is no need to learn those commands any more. Whenever low-level commands are used they are marked appropriately by using a slanted fixed-width fond like this: ***vxlowlevel command you need not remember***. Normally the common higher-level commands are discussed close to the low level ones. They are set in a straight fixed-width font like this: **vxhighlevel command you should remember**.

V. Herminghaus and A. Sriba, *Storage Management in Data Centers*,
DOI: 10.1007/978-3-540-85023-6_2, © Springer-Verlag Berlin Heidelberg 2009

2.1.1 Hello, Volume!

Let us create the VxVM equivalent of Kernighan & Ritchie's famous "Hello, World!" C-language program: A thoroughly explained walk through of VxVM's essential data structures and virtual objects. By the end of the chapter you will know a fair amount about volume management in general and VxVM's virtual objects in particular. You will actually know more than most UNIX administrators that just use VxVM on a daily basis without ever wondering what is going on behind the scenes.

2.1.2 vxdisksetup: Turning Disks into VM Disks

Since VxVM must eventually store volume data on a magnetic surface there must be a way of integrating disks or LUNs into the VxVM object hierarchy, to take storage media away from operating system control and pass them over to VxVM control. And of course there is. Using a simple command - **vxdisksetup** <diskname> - you reserve a disk for VxVM use. Since a disk treated with this command is now virtualized by an additional software layer it has gained some rather handy features. For instance, you can now use software commands - *vxdisk online* and *vxdisk offline* - to switch the disk on and off; something that is not possible using just operating system commands.

Of course these commands do not actually make the disk stop rotating or keep the controller from responding; they merely simulate turning the disk on and off to the other virtualized layers above. But as long as you do not break the paradigm and stay inside the virtualized layers, this is for all practical purposes identical to actually switching the disk on and off.

What Does vxdisksetup Do?

It first checks the disk's physical properties, like size, type of storage array it resides on (if it is a LUN) and the device paths that DMP provides to the disk. It then uses the knowledge just gained to define an internal data structure called an access record through which it can do read and write I/O from and to the disk via DMP. The process of creating the access record is implemented by the command *vxdisk define*, which is called from inside the **vxdisksetup** script. The access record created by *vxdisk define* also contains flags like the physical and virtual online or offline state of the disk, the list of paths to the disk along with their state, possibly information about the location of the disk (the "**site**" variable), and the disk's universal device ID (**UDID**). Remember that all these variables are just software data which are used inside VxVM to represent a physical disk as accurately as possible while extending its abilities by things like reservation, software on/off or software failed/OK state. (More information about disk flags is provided in the **miscellaneous** chapter beginning on page 479.)

Next on the list of things that **vxdisksetup** needs to do, at least in the great majority of cases, is to reserve some private space on the disk medium in where it can persistently store all the information pertaining to the disk. For instance, if VxVM set the "failing" flag for a disk because an unrecoverable read error occurred on some user data, you would like this state information to be persistent across reboots. No offense intended, but only

Windows users try to fix problems by rebooting. In UNIX and other serious operating systems rebooting a system should not normally change a lot, except maybe enable a kernel level patch or driver configuration change to become relevant.

Sliced Format

VxVM, or more specifically the **vxdisksetup** command, reserves space for its private use by allocating a (normal) slice on the disk and marking it as a "Private Region" of VxVM by giving it the reserved tag 15 in the Solaris VTOC. That slice is a perfectly normal slice which can be inspected using Solaris' **format** or **prtvtoc** commands. The tags are not usually interpreted by the operating system; anyone could theoretically use any tag for their partitions. There are conventions, however, that can make life easier. For instance, if a slice has tag 3 this marks it as a swap device, and the Solaris install program will use it as a destination for its mini-root file system during install. There are also tags for "root", "boot", "home" and several others.

The "Private Region" used to be the smallest number of cylinders that would yield at least 1MB, although that has changed several times in recent releases. In any cae, its size is no longer relevant since the advent of the new CDS disk layout which has superseded the classic format and which will be discussed later.

The **vxdisksetup** command, after reserving the private region, then reserves the rest of the disk by allocating another slice and marking it as the "Public Region" (tag 14). To an administrator, and to all UNIX tools that handle disks, the disk now looks like its space is completely used up; there is no free space on it that is available for the operating system after **vxdisksetup** is run.

The disk format generated by vxdisksetup as described above is called the "sliced" format, because private and public regions reside in visibly separate slices of the disk. Because it relies on the Solaris VTOC, such a disk can only be used on platforms which are compatible with Solaris VTOCs, namely Solaris and Linux. It is no longer generally useful since the advent of the new, universally compatible format called CDS (Cross-platform Data Sharing, discussed below) and only discussed here for reasons of conceptual clarity.

To initialize a disk as a sliced disk use the following command:

```
# vxdisksetup -i c#t#d# format=sliced
```

CDS Format

In more modern versions of VxVM the whole disk is marked as a "Private Region" by **vxdisksetup**, which has led to some confusion among users. The usual reason for this confusion is the (wrong) assumption that the private region held **only** metadata, while the public region held **only** user data. While this was true with the original **sliced** format, the whole truth seems to be that a disk that is under VxVM control needs two things in order to coexist with the operating system: First, a way of telling VxVM that it is valid for VxVM. That is achieved by using the special VxVM tag number 15 for the Private Region, and of course by initializing the Private Region with a valid data structure. And second, a way of telling the rest of the system to keep their hands off the disk because it's under VxVM control now. The latter is achieved by marking the whole disk as allocated through the allocation of a single slice covering the whole disk. Previously these two things were

done by the Private Region slice serving the first purpose and the Public Region slice serving the second purpose. A close look revealed that both could be served at the same time by just allocating the whole disk to one huge Private Region, thus telling VxVM that the disk is destined for VxVM and telling everybody else that the disk is fully allocated at the same time. (The Private Region holds all the metadata VxVM needs for allocation of virtual objects on the disk; VxVM never actually required the Public Region slice for that purpose.) This is what is done in the current default layout, called CDS for Cross-platform Data Sharing. CDS formatted disks can be freely moved between Solaris, Linux, AIX and HP-UX as long as they are all running VxVM 4.0 or above.

The last step that **vxdisksetup** does is initialize the Private Region with reasonable values. It does so only if it has been passed the **-i** command line switch, i.e. if the command was not just **vxdisksetup** <diskname> but **vxdisksetup -i** <diskname> instead. Initializing the private region is done by the ***vxdisk init*** command which again is called from inside the **vxdisksetup** script, so you never need to call it manually. To initialize a disk for VxVM as a CDS disk use one of the following commands:

```
# vxdisksetup -i c#d#t#                  # cds is the default format since 4.0
```

or

```
# vxdisksetup -i c#t#d# format=cds       # if default has been changed
```

It is also possible to encapsulate a disk that already contains file systems (either from the operating system's addressing scheme or from other volume management products) by a process called encapsulation. This process is extensively handled in its own chapter beginning on page 319.

So now, after initializing the disk, we are ready to go one step further up the hierarchy: Grouping disks together to build a base for creating volumes.

2.1.3 Disk Groups: Putting VM Disks into Virtual Storage Boxes

Any software that implements RAID functionality will have to use more than one physical disk if it tries to do anything useful. Trying to break the limits of physical disks while constraining oneself to a single disk does really not make any sense. So VxVM never works on individual disks when creating virtual objects of a higher level. Instead, the least thing that VxVM needs is a collection of disks that can be addressed as a container for virtual objects like volumes. This collection is called a **disk group** or DG. You may think of a disk group as a virtual box of disk media, which can be attached to and detached from hosts by using software commands, effectively moving storage between servers.

Of course it is debatable if we actually need to collect disks together into DGs in order to handle them. Why not just use all the disks connected to our system, and allocate from all of them? The answer has to do with highly-available systems or clusters: In order to have a highly available service we need to virtualize not just the mass storage layer but also the server itself. I.e. we need more than one physical instance of a server and then

map these multiple instances of server hardware onto one or more logical server instances, called "Logical Hosts" or "Service Groups". These service groups work on data that is alternatively, sometimes even concurrently, accessed by any of the hosts of the cluster. In order to allow a controlled failover of a service group from one physical host to another the data is encapsulated into individual groups of disks. Each service group contains (typically) one disk group that holds all the storage resources required by the service. If a service group fails on one host, the service group's resources are stopped on that host and then started on another host. This means that access to the disk group is disabled on the host on which the service has failed, and enabled on the spare host. If the disks were not organized into disk groups then it would be hard to pass control over just part of the storage (i.e. the storage required by one service group) to another host. Therefore the use of disk groups does indeed make a lot of sense. Disk groups are created by the **vxdg init** command. Because disk groups are made to move between systems they must be completely self-describing, i.e. **all the information about the disk group resides inside the disk group itself**. Otherwise it would be hard to move the disk group from one system to another without having to transfer some description file along with it. Because all information about a disk group is contained inside the disk group, a disk group must always contain at least one disk. It is not possible to have a disk group without a disk, so you must pass at least one disk on the command line when you create a disk group.

Here is an example that creates a disk group named **mydg** from three disks: **c0t2d0**, **c0t3d0**, and **c0t4d0**:

```
# vxdg init mydg mydg01=c0t2d0 mydg02=c0t3d0 mydg03=c0t4d0
```

Now after having created a disk group containing several individual virtual disks we can look at what parts of the storage are unused by asking VxVM to report the free extents. This is done by using the low-level command ***vxdg free***. Since there may be many disk groups in use by our system VxVM always requires that we pass it the name of the disk group with the command. So the correct command is actually ***vxdg -g mydg free*** (the **-g** flag obviously stands for "group"). The result is a list of extents, one per disk, which are free for user data.

```
# vxdisk -g mydg free
DISK         DEVICE       TAG          OFFSET    LENGTH    FLAGS
mydg01       c0t2d0s2     c0t2d0       0         17909696  -
mydg02       c0t3d0s2     c0t3d0       0         17702192  -
mydg03       c0t4d0s2     c0t4d0       0         17679776  -
```

The free extents (remember: offset and length, both given in 512-byte blocks) are printed in boldface. Each disk has a single free extent starting at block 0 and covering all the free space available to VxVM.

Disk groups are covered in their own chapter beginning on page 71.

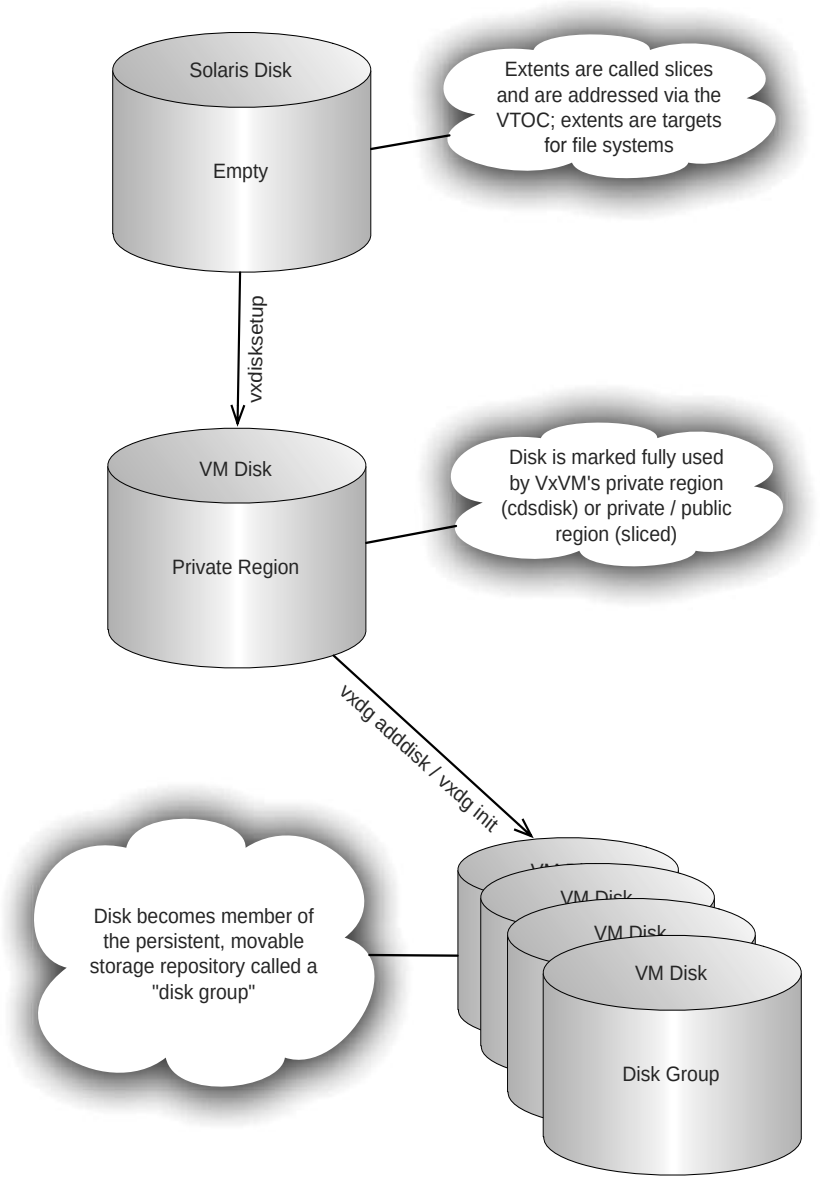

Figure 2-1: **Initializing a disk for VxVM creates a so-called VM disk, which is marked "completely full" for the operating system (but not for VxVM). It can then be added to a disk group using the appropriate command, and subsequently used as persistent backing store for virtual volumes.**

We have now virtualized our physical hard disks to serve as a VxVM-addressable backing store for our storage objects The next step is to allocate space from this backing store. We used to allocate space by defining a partition (or slice) in the VTOC of a hard disk. However, the limitations of the VTOC format would not let us define more than 8 slices per disk. After all, the VTOC is only a single 512-byte block that cannot hold much information. There just isn't enough space available in the VTOC for allocating little extents and mixing them together in some fancy way to get a mirrored or striped volume. Remember the VTOC is plain Solaris' equivalent of the VxVM private region, but much too small to hold enough information for any serious kind of virtualization. It was never designed for doing that.

So how do we go about allocating an extent from a VxVM disk? There are two ways to do it. One is a low-level command called ***vxmake*** which is used to create almost any kind of VxVM object. The other one is a high-level command called **vxassist** that creates, allocates and connects all the objects required for the task that you give it (for instance, creating a four-way striped mirror in one go).

If you don't want to learn about the internal objects now, you can skip to page 53 at the end of this chapter, where we show the easy way to create volumes. But do come back to this part eventually if you need to understand VxVM!

2.2 The Hard Way: a Low-level Walkthrough

2.2.1 Subdisks: Extents for Persistent Backing Store

You're still reading? Good! In order for ***vxmake*** to create the right object we need to pass it the object type first. The object type we want is an extent, i.e. part of a disk. In VxVM terminology this is called a **subdisk or sd**. The ***vxmake*** command needs to know which type of object to create (sd), which disk group to use (mydg), what the name of the subdisk to create (mydg01-01), the disk to allocate storage from (mydg01), and the extent information, i.e. offset and length (offset 4000, length 2 GB). We have chosen an offset of 4000 for no specific reason; it just looks nicer than an offset of 0.

The low-level command that is actually run looks something like this:

```
# vxmake -g mydg sd mydg01-01 disk=mydg01 len=2g offset=4000
```

This will allocate from VxVM's virtual disk **mydg** an extent of 2GB length beginning at offset 4000 blocks from the beginning of the virtual disk's public region (wherever that is; VxVM keeps track of it so we do not actually care) and note its existence under the name "**mydg01-01**" in the Private Region. After the extent has been allocated, VxVM makes sure no other extent will use blocks from the extent we just allocated; any attempts to do so will fail. We can then check **vxdg free** again and see that the extent we just allocated is extracted from the free storage pool and is no longer available.

```
# vxdg -g mydg free
DISK         DEVICE       TAG          OFFSET    LENGTH    FLAGS
mydg01       c0t2d0s2     c0t2d0       0         4000      -
```

```
mydg01       c0t2d0s2     c0t2d0       4198304   13711392  -
mydg02       c0t3d0s2     c0t3d0       0         17702192  -
mydg03       c0t4d0s2     c0t4d0       0         17679776  -
```

As you see, the disk mydg01 now lists two free extents: one from 0 to 4000, the other one starting at 2 GB behind 4000 and extending all the way to the end of the public region.

We can also use a print command – vxprint – to check and see that the object was indeed allocated. The command for this is **vxprint -g mydg**.

```
# vxprint -g mydg
TY NAME         ASSOC        KSTATE   LENGTH   PLOFFS   STATE    TUTIL0  PUTIL0
dg mydg         mydg         -        -        -        -        -       -

dm mydg01       c0t2d0s2     -        17909696 -        -        -       -
dm mydg02       c0t3d0s2     -        17702192 -        -        -       -
dm mydg03       c0t4d0s2     -        17679776 -        -        -       -

sd mydg01-01    -            ENABLED  4194304  -        -        -       -
```

The length of the subdisk (in the last line of output) is shown as 4194304 blocks of 512 bytes, which is exactly 2 GB, as you can verify like this:

```
# echo 4194304/1024/1024/2 | bc -l
2.0000000000
```

2.2.2 Plexes: Mapping Virtual Extents to Physical Extents

The next layer up in the virtual storage hierarchy is a data structure that implements concatenation, striping or striping with parity. In order to do so it maintains an internal mapping table which maps its address space to appropriate extents on VxVM subdisks. In doing so this virtual layer is able to implement some of the RAID levels. For instance, concatenation is implemented by mapping individual subdisks sequentially into the plex's address space. The virtual object that implements this behavior is called the **plex** (plex is a latin suffix meaning "-fold", as in twofold, manifold). The plex's address space is identical to the address space of a virtual volume, i.e. it is block-addressed, starts at an offset of zero and extends contiguously and typically without holes (although there can be "sparse" plexes) to the end. In that respect it is similar to a partition, which also exposes a single, linear, block-addressed space to the layer above it (usually the file system or raw device layer).

Concatenation is not the only type of mapping that the plex object can implement. It can also do striping and striping with parity (RAID-5). These layout types are implemented by mapping several subdisks into the address space in an interleaved manner. In the simple case they all start at a plex-offset of zero and extend all the way to the end of the plex. The plex's access methods implement skipping from column to column appropriately, and

generating the parity checksum in the case of a RAID-5 plex.

In a more complicated case, several subdisks will have to be concatenated inside one or more column(s) so they will start at their respective offsets inside the plex, rather than at offset zero. In the case of a striped layout the mapping of plex addresses to subdisk addresses is done via internal logic of the plex. The plex methods know the layout of the plex as well as the number of columns and the stripe size. With this information the plex is able to calculate the correct physical extent on a subdisk for any given logical extent in a plex, and issue the I/Os to the SCSI driver appropriately.

This is how a plex is created with a low-level command:

```
# vxmake -g mydg plex myvol-01

# vxprint -g mydg
TY NAME         ASSOC        KSTATE   LENGTH   PLOFFS   STATE    TUTIL0  PUTIL0
dg mydg         mydg         -        -        -        -        -       -

dm mydg01       c0t2d0s2     -        17909696 -        -        -       -
dm mydg02       c0t3d0s2     -        17702192 -        -        -       -
dm mydg03       c0t4d0s2     -        17679776 -        -        -       -

sd mydg01-01    -            ENABLED  4194304  -        -        -       -

pl myvol-01     -            DISABLED 0        -        -        -       -
```

Note that no disk is specified. The plex actually does not have any backing store now; it is just like an empty balloon that needs to be filled before use. As you can see in the last line of output above, where the plex is displayed, its length is zero.

So now we created this virtual object called a plex, and we had created another virtual object before called a subdisk, which can be used as backing store for plexes. Let us bring the two together, or **associate** them, using another low-level command. This is the *vxsd assoc* command (*sd* obviously stands for subdisk, *assoc* stands for associate):

```
# vxsd -g mydg assoc myvol-01 mydg01-01
# vxprint -g mydg
TY NAME         ASSOC        KSTATE   LENGTH   PLOFFS   STATE    TUTIL0  PUTIL0
dg mydg         mydg         -        -        -        -        -       -

dm mydg01       c0t2d0s2     -        17909696 -        -        -       -
dm mydg02       c0t3d0s2     -        17702192 -        -        -       -
dm mydg03       c0t4d0s2     -        17679776 -        -        -       -

pl myvol-01     -            DISABLED 4194304  -        -        -       -
sd mydg01-01    myvol-01     ENABLED  4194304  0        -        -       -
```

Now the plex and subdisk belong together and the plex' size has grown from zero to the size of the subdisk it contains. There is a slightly easier way of creating a plex with subdisks that works by specifying the *sd=<sd-list>* parameter on the *vxmake* command

line.

Two more low-level commands for creating plexes are provided as samples here: The first one will create a concat plex containing just the subdisk that was allocated above, the second one will create a three-column striped plex with a stripe size of 1MB (2048 blocks) out of three subdisks which we have not created here (so the command would actually fail if you typed it in without creating the subdisks first. It is meant as a sample for the command syntax):

```
vxmake -g mydg plex myvol-01 sd=mydg01-01
vxmake -g mydg plex myvol-02 sd=mydg02-01,mydg03-01,mydg4-01 \
        layout=stripe ncol=3 stripewidth=2048
```

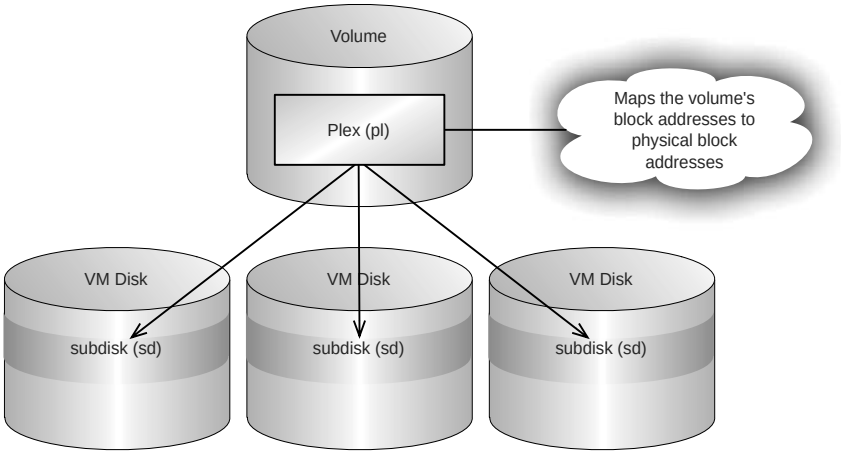

Figure 2-2: **Plexes do the translation work. Each plex maps the contents of the whole volume onto persistent storage (i.e. subdisk). Having more than one plex in a volume therefore is equivalent to mirroring. Up to 31 plexes can be in a volume.**

2.2.3 Volumes: Virtual Partitions for Any Purpose

So now we have virtualized disks, disk groups, subdisks, and plexes. Can we finally put a file system onto something now? Actually we can not. While a subdisk object is by its nature very much the same as a partition, namely an extent of blocks on a magnetic disk, it does not provide a device driver node in the **/dev** directory onto which we could issue an **mkfs** or **newfs** command, let alone mount it into the file system tree. Even the plex object, with its clever logic able to distribute I/Os across several disks in a number of ways (concat, stripe, RAID-5) does not provide us with a device driver node. So how are we going to use

the raw, plain magnetic storage (subdisk) or the cleverly organized magnetic storage (plex)? The answer is simple: we need another object type to build on top of the plex. Why does VxVM not just use the plex and provide a device driver so we can write into a plex? The reason is that a plex, being just a mapping layer for redirecting virtual extents to physical extents, contains only one single instance of the address space, not several instances. In other words, it does not provide a mirroring functionality. Because Volume Manager needs to implement mirroring, too, there must be a container object that can hold a number of plexes. In order to implement mirroring VxVM puts several plexes into a single container called a volume. This container object (volume) implements a device driver node in side the **/dev** directory hierarchy, and thus lets the file system driver address the volume. The volume will multiplex write I/Os to all plexes inside the volume, and satisfy read I/Os from any one of the plexes in the container using each plex's individual mapping function to address the correct pieces of backing store for each plex. In other words, the volume object is the type of virtual object that offers the device driver that we can write into and read out of. Without a volume around it, a plex is just a cleverly mapped, yet inaccessible stretch of blocks. The volume objects implements the UNIX device node that we need to access the plex or plexes which ultimately offer the mapped backing store for the volume.

A volume object is created by e.g. the following low-level command:

```
# vxmake -g mydg vol myvol usetype=fsgen len=2g
```

Ignore the parameter ***usetype=fsgen*** for now; it is used to tell VxVM the intended purpose of the new volume and is syntactically required for the command. Let's see what we get:

```
# vxprint -g mydg
TY NAME         ASSOC        KSTATE   LENGTH   PLOFFS   STATE    TUTIL0  PUTIL0
dg mydg         mydg         -        -        -        -        -       -

dm mydg01       c0t2d0s2     -        17909696 -        -        -       -
dm mydg02       c0t3d0s2     -        17702192 -        -        -       -
dm mydg03       c0t4d0s2     -        17679776 -        -        -       -

pl myvol-01     -            DISABLED 4194304  -        -        -       -
sd mydg01-01    myvol-01     ENABLED  4194304  0        -        -       -

v  myvol        fsgen        DISABLED 4194304  -        EMPTY    -       -
```

Again, the new object is not yet connected to any other objects. It is a standalone, empty volume object with no plex inside it. Now let us put the plex into the volume. Again, the volume is just an empty data structure or virtual object that bears some flags and other information, like the information that it is 2 GB in size (we specified ***len=2g***, just like we did with the subdisk). The operation that puts a plex into a volume is called **attaching** a plex. A plex attach operation has the effect of copying the volume data onto the plex, overwriting the plex's previous contents. Of course this only happens if the volume already holds data (i.e. if there already is at least one plex in the volume), which is not the case here. So in our case nothing will be copied when we issue the plex attach command, but the volume

will subsequently contain the data that is stored in the plex's blocks:

 # vxplex att myvol myvol-01

OK, so now the subdisk is inside the plex so that the plex has backing store to do its I/O to. And the plex is attached to a volume so that the volume can take actual I/O requests from the device driver interface (which only the volume provides). Where is the device driver that corresponds to our precious, hand-made volume? Let us look into the **/dev** directory!

In the **/dev** directory you find the usual subdirectories for buffered (**dsk**) and unbuffered (**rdsk**) disk access. Buffered is also referred to as "block I/O" and unbuffered as "raw" or "character I/O".

What's new in the **/dev** directory since we installed VxVM is a new directory **/dev/vx**, with **vx** being short for Veritas. Inside **/dev/vx** you will find **dsk** and **rdsk** directories, just like the ones we saw in **/dev**. The difference is, the **/dev/*dsk** directories contain device drivers to physical devices, so you will find the usual **c#t#d#s#** or similar device names inside of them. Opposed to the physical view offered by **/dev/*dsk**. the **/dev/vx/*dsk** directories offer a view of the virtualized storage. So in these directories we obviously do not need to cope with clunky controller names, but we get the elegant ones that we chose for our virtual objects, nicely arranged in a hierarchical order. What we see is the device node with VxVM's major number (in this case: 270) and the volume's minor number (31000):

```
# ls -l /dev/vx/dsk/mydg
total 0
brw-------   1 root     root     270, 31000 Nov  3 14:38 myvol
```

In general, the device paths result from the prefix **/dev/vx/*dsk**, followed by a directory that carries the name of the disk group (**mydg**) followed by the device driver that carries the name of the volume (**myvol**).

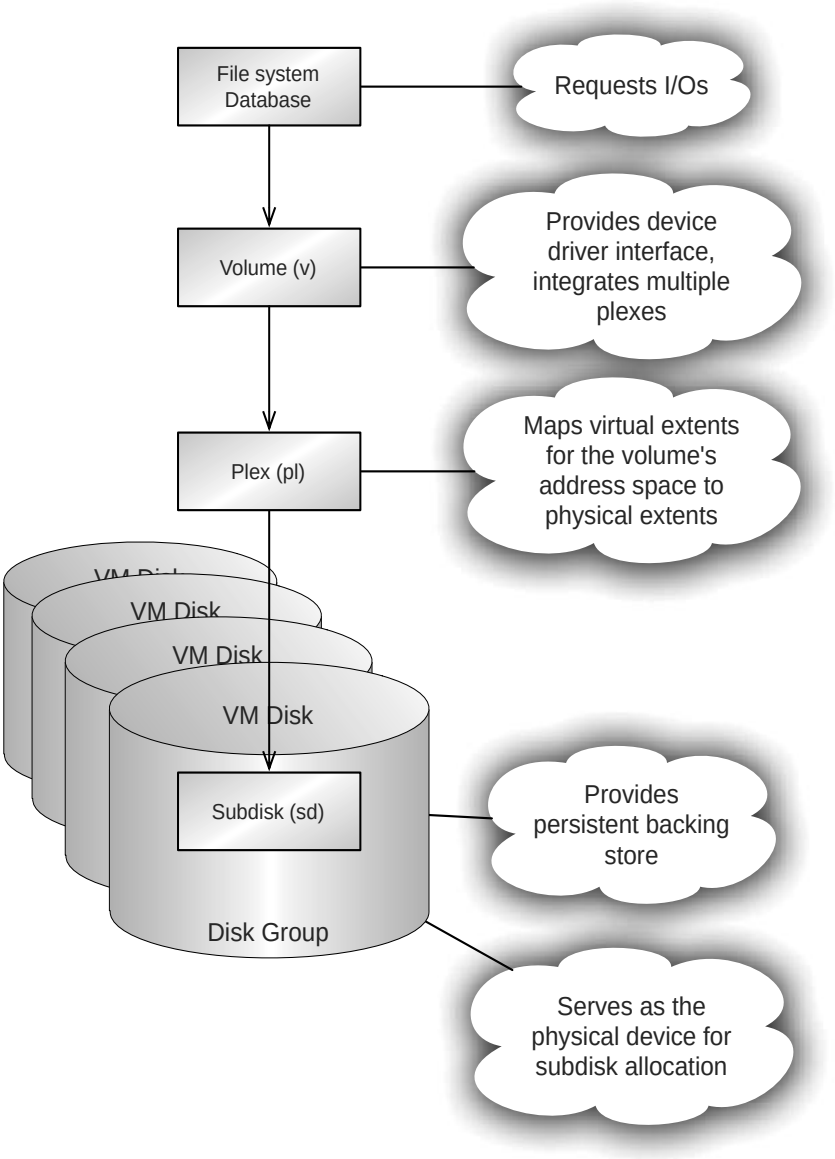

Figure 2-3: The hierarchy of virtual storage objects in VxVM.

So now we will try putting a file system on the volume. That should work just like on any partition, because as you should remember VxVM must strictly adhere to emulating a partition's behavior towards the higher level drivers. If a volume behaved in any way differently from a partition then the file system or database or whatever else it is that uses the volume might get unexpected results and panic the machine. That is why VxVM undertakes

great efforts to maintain behavioral consistency with a partition.

Yet when we now try to actually do anything useful with our manually created volume we get an error:

```
# newfs /dev/vx/rdsk/mydg/myvol
Are you sure ...? y
/dev/vx/rdsk/mydg/myvol: no such device or address
```

2.2.4 Volume Start: Prepare for Takeoff

Why doesn't VxVM let us access our self-made volume? The reason is actually quite simple: The volume needs to be checked for consistency before use. After all, a partition only holds a single data container: the extent corresponding to the slice itself. A volume however might hold several copies of the data, up to 32 to be exact. What if these are not perfectly identical? Then the partition paradigm would really be broken, because if the file system gets different contents every time it reads the same extent., this will cause major problems. For that reason VxVM needs to check the consistency of the volume before you can start using it. Of course it does not compare the whole volume contents but rather it employs a clever concept of states and state transitions with which it can derive if the volume's contents **must** be OK or if they **might not** be. In the latter case, the volume's contents indeed need to be resynchronised, while in the former nothing needs to be done.

It would be a rather silly idea if VxVM checked the volume's consistency permanently, or if it checked before every single I/O. We want the checking done exactly once; when we begin using the volume, and never thereafter until we give up control of the volume by deporting the disk group for some other host to use it. That other machine is out of our control so we must re-check the volume when we get it back. Checking and thus enabling the volume is called "starting" the volume. It is done by executing the simple command:

```
# vxvol -g mydg start myvol
```

or, to start all volumes in the disk group:

```
# vxvol -g mydg startall
```

Now we really have a virtual volume that can be used for everything that a partition can be used for: Buffered file system I/O, unbuffered or "raw" database I/O, creation of file systems, **ufsdump/ufsrestore** and everything else.

```
# newfs /dev/vx/rdsk/mydg/myvol
/dev/vx/rdsk/mydg/myvol:  4194304 sectors in 2048 cylinders of 32 tracks, 64 sectors
        2048.0MB in 47 cyl groups (44 c/g, 44.00MB/g, 11008 i/g)
super-block backups (for fsck -F ufs -o b=#) at:
 32, 90208, 180384, 270560, 360736, 450912, 541088, 631264, 721440, 811616,
 3334496, 3424672, 3514848, 3605024, 3695200, 3785376, 3875552, 3965728,
 4055904, 4146080,
```

2.3 The Easy Way: vxassist

If the above sounded complicated to you then please remember that the low-level commands are there just for you to understand the basic data structures and virtual objects. You never actually create them in the way described above. The real way of creating volumes, together with all the virtual objects they contain (plexes and subdisks) is to use a single, high-level command named **vxassist**. This command will do all the smart thinking and hard working for you and present you with a volume that is ready to go and looks exactly like you specified it (unless you did not bother to specify a lot, in which case **vxassist** will choose reasonable defaults). Once you have a disk group with a few disk in it, all of the above could have been done by this very simple command:

```
vxassist -g mydg make myvol 2g
```

And the striped volume (which we did not even finish due to lack of subdisks) would have been created, including allocation of subdisks, creation of plex, and starting of the volume, by this command:

```
vxassist -g mydg make myvol2 layout=stripe ncol=3 stripewidth=2048
```

2.3.1 Summary

This chapter contained a hard-core walkthrough of VxVM's basic virtual objects and how they connect together to form a virtualized container for you to use instead of a simple hard disk or LUN partition. Do not be intimidated by the complexity of what you just read; using VxVM is normally quite simple, as you could see from the last paragraph. Getting up to speed in VxVM is fairly straightforward and easy. But if you are working in a data center you are probably not served well by a simple guide to VxVM. You need to understand what's

behind the scenes, how stuff is done in VxVM and why. My favorite comparison for that is based on cars: If someone teaches you how to drive by telling you when to press which pedal and when to move the gear shift level to where, then you will be able to drive to work every day. But when you are doing VxVM in a data center environment you are not "driving to work". You are doing the equivalent of driving the "Rallye Paris-Dakar". Nobody can prepare you for what happens on such a trip. You need to know the inner workings of your car as precisely as possible. How does the steering actually work, what does the center differential do, how does tire pressure, camber, anti-roll bars and toe-in settings interact with the shock absorbers and springs etc. That is why this book drills right down to the low level objects and shows their interactions. Then, once you understand those, it shows you how to do the simple stuff easily.

The easy parts of this book are marked with a little "Easy Sailing" boat. The comprehensive listing of all interesting features and how to use them is marked by "The Full Battleship", and explanations about implementation details as well as much of a technical reasons for VxVM's design and behavior are covered in sections marked with the "Technical Deep Dive" submarine. You can select the chapter section as you like, they do not usually depend on each other.

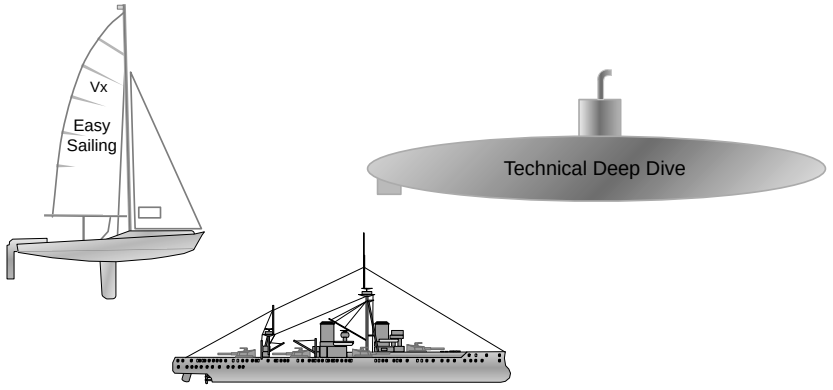

Figure 2-4: The vessels that will guide you through this book, and what they stand for: Easy Sailing, The Full Battleship, and the Technical Deep Dive