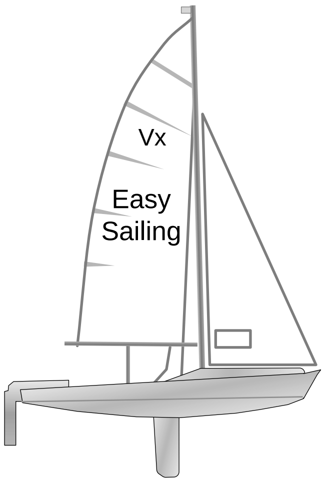
CHAPTER 3: INCORPORATING DISKS INTO VXVM

by Volker Herminghaus

In the previous chapter you gained an overview over what the concept behind Veritas Volume Manager (or VxVM) is, what storage objects are used and how they act together to form a complete volume management layer. So now, after the extensive problem introduction in chapter one and the in-depth look at VxVM objects from chapter two, let us get some actual work done. We will start with a machine that is already running VxVM, go through all steps necessary to incorporate a new disk into a VxVM disk group and discuss what is going on as well as alternative ways to do it.



3.1 Solaris Disk Handling

3.1.1 Getting a New Disk into Solaris

Once VxVM has been successfully installed and configured the command **vxdisk list** shows all disks attached to the system and known to VxVM. Attaching a new disk (or LUN, remember the two are the same to VxVM) to a Solaris machine does not make that device visible to the operating system. What is necessary in order for the OS to see a new disk? First of all the OS needs a device driver which it can use to physically "talk" to the HBA (host bus adapter) to which the disk is attached. We will assume for now that the HBA driver is already installed. The machine obviously also needs a SCSI protocol driver to use on top of the HBA driver so it can talk SCSI protocol with the disk. The SCSI driver is always installed, otherwise we would not have been able to boot from a SCSI disk. Theoretically, if you boot from a network or other non-SCSI device you could actually be missing the SCSI driver, but unless something really weird happened the SCSI driver would be automatically loaded by the kernel as soon as it is required. So let us pretend we have both the HBA and the SCSI protocol driver installed in the system already.

Now we physically connect a new disk device (or zone a new LUN) to the system. Does this make Solaris recognize the new device? Of course it does not. Even a reboot will go no step towards making the disk visible, unless we supply the reconfigure-flag (**reboot -- -r**) or create a file named **/reconfigure** before booting in order to make Solaris actually look for new devices. But we do not need to actually reboot; this would be quite an insult to the operating system, so to speak. What the OS (Solaris 9; 10 skips the first step) does when it boots in reconfigure mode is execute the command **luxadm -e create_fabric_device** to re-acquire FC targets that were previously configured by the admin using **cfgadm** and whose WWPNs (World Wide Port Names) the **cfgadm** command stored in **/etc/cfg/fp/fabric_WWN_map**. After these **luxadm** commands it eventually calls the command **devfsadm**. This command instructs the device file system configuration daemon process to inquire all devices on all controllers and check for new devices as well as ones that have been lost. For the new devices it will then create device nodes in **/dev** and **/devices** so that the device will be accessible via the UNIX file system tree.

This is when VxVM first gets a chance to see the device. Unless a valid device file exists Volume Manager cannot access the disk, nor can it somehow miraculously see things that the OS does not see or fix things that the OS cannot fix. VxVM, like any other product, first requires that the OS be capable of physical access to the device.

3.1.2 You Don't Format with "format"

Now that the OS sees the disk device we can almost switch to purely using VxVM commands so that we can finally get rid of all that physical disk legacy. But first, one more thing needs to be done by operating system means, and that is to format the disk, i.e. to put a valid Solaris label (or VTOC) on it so that the higher software layers can not just read and write individual blocks of the disk, but can actually get some reasonable meta information about it, like its size, number and locations of partitions etc. Why can that not be done without a valid label? The reason is actually very simple: When **devfsadm** finds a new disk and creates its device node it does not create just one device file but eight; one for each slice. Now imagine your new LUN is a re-used one from some antique, crufty operating system that stores some arcane partition table in block zero of the disk. We get this new LUN into our system, **devfsadm** creates eight device files so we can access each of the Solaris slices that can be registered in the VTOC block. But alas, there is not a single usable partition on this former Windows (or other) disk. All we can see is illegible stuff in an unintelligible format. At the byte offsets where we expect a 32-bit block number specifying the offset for partition three, for instance, there might even be a negative value! What would **newfs** do when we tried to create a file system on such a partition? It might actually skip backwards and overwrite important data unless the programmers took extra caution to prevent this.

This is why every disk or LUN that is to be used in Solaris must always carry a valid VTOC. You use the format command to write a valid VTOC (which is called "labelling" in the **format** program). You can also actually issue a SCSI "format" instruction to a disk from the Solaris format command, but this is almost never done since all modern disks come pre-formatted. All they need is a valid Solaris VTOC. The same is essentially true for other operating systems as well, since they all need some kind of fixed entry point into the disk's or LUN's internal data structures, and this fixed entry point must be initialized before the medium can be used.

3.1.3 Finding New Disks in VxVM

So now that the LUN or disk is working, the OS sees it, and the LUN is even labeled correctly so the OS does not stumble over some leftover from the previous user. When will VxVM see this new disk? How do we even know if VxVM sees the disk?

Well, the last one is easy. The command **vxdisk list** shows all disks that VxVM knows, one per line, with their access records to the left and their status to the right. Here's an example:

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t10d0s2    auto:none       -            -            online invalid
```

Apparently in this case, VxVM only sees two disks: Their access records are **c0t0d0** (which happens to be the boot disk) and **c0t10d0**. Will it see new disks if we reboot? Yes, but that would be very Windows-like, and not very friendly towards the 14,786 users connected to the online banking system that we are currently working on. All we need to do is tell VxVM to go check for new disks. The right command for this is

```
# vxdisk scandisks
```

This command could even be further limited to look for just specific new disks, controllers and the like, but the given form is the most useful one since it is rare that someone wants to make VxVM see some of the new disks but not all etc. In these cases, **man vxdisk** does a much better job at explaining what command to issue than this book could.

Many of our readers will be surprised because they may be familiar with a different command to make VxVM scan for new disks. This command is

```
# vxdctl enable
```

This command (**vxdctl** stands for Veritas daemon control – it controls the Veritas Configuration Daemon **vxconfigd)** does indeed find new disks (and has done so in all recent releases), but it also does a lot more. It actually does a physical I/O to each disk to check whether the disk is still accessible. If the disk was previously uninitialized it may or may not try to re-read the VTOC in order to find a valid Private Region. It may then decide to read or not to read the Private Region to find out if it is valid or if it has become invalid etc. It also rebuilds the DMP tree and does a lot of other things. Unfortunately what exactly it does differs from version to version, so I would like to suggest you start out with **vxdisk scandisks** and only use the **vxdctl enable** command if the new disks do not pop up right away. Let's try integrating my newly attached disk, which we expect to be **c0t11d0**:

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t10d0s2    auto:none       -            -            online invalid
```

It's not there, because the OS has not created the device node yet.

```
# devfsadm      # make all necessary device nodes
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t10d0s2    auto:none       -            -            online invalid
```

It's there, but VxVM's cache does not know it needs to be refreshed. All display commands for VxVM objects read from cached data, as mass storage access could slow the machine or process down.

```
# vxdisk scandisks      # refresh VxVM's cache
# vxdisk list
DEVICE       TYPE           DISK         GROUP        STATUS
c0t0d0s2     auto:none      -            -            online invalid
c0t10d0s2    auto:none      -            -            online invalid
c0t11d0s2    auto:none      -            -            online invalid
```

Here we are!

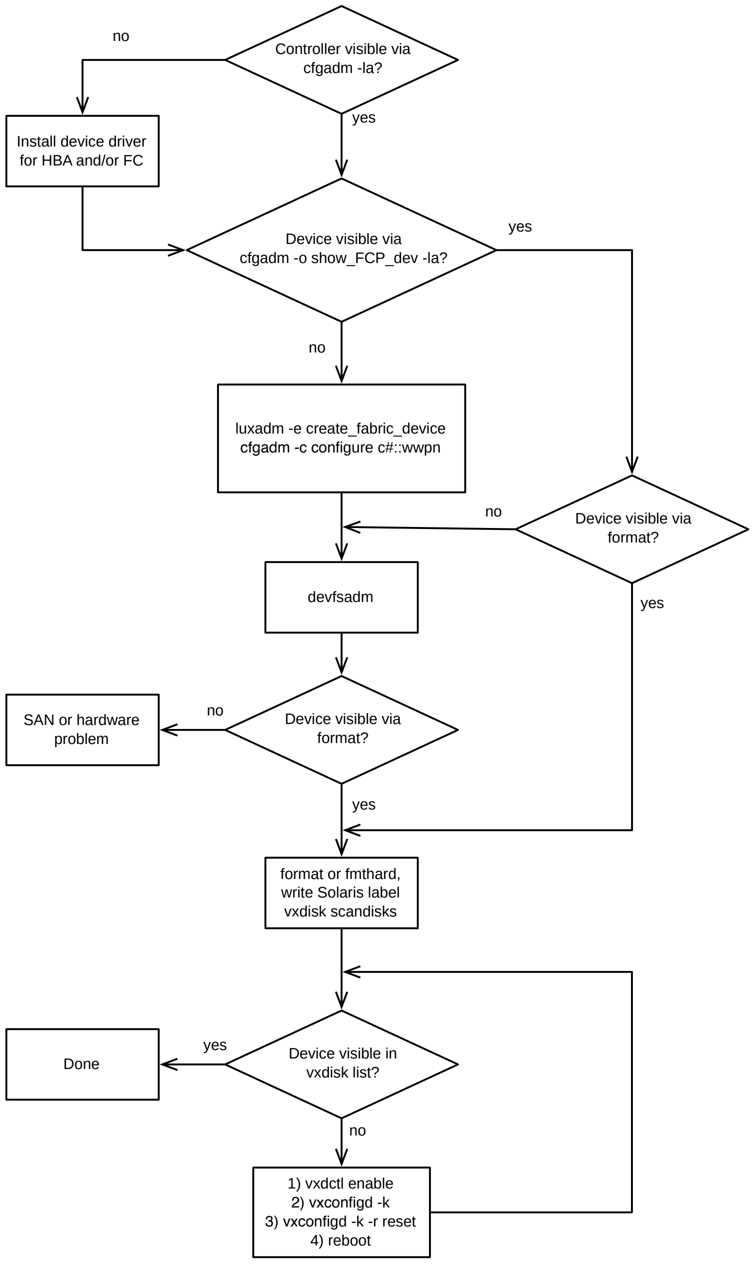
3.1.4 What if My New Disk is Not Found?

If **vxdisk scandisks** and **vxdctl enable** both fail to yield the desired result then there are several further steps you can take in order to make VxVM recognize disk changes. Note that recognition of new LUNs is normally painless, but recognition of disks that had their VTOC changed, or that had their Private Region overwritten or restored (something we like to do in VxVM trouble shooting courses) may require further measures. These may also need to be taken if for instance you get new LUNs that actually have valid Private Regions but that stem from outdated disk groups which had been freed (or worse: deleted on raw disk level) from another server before.

These further measures are, in the order they should be tried:

```
# vxdisk scandisks # tells vxconfigd to scan for new disks
# vxdctl enable # tells vxconfigd to scan a lot of things for a lot of changes
# vxconfigd -k # kills and restarts the configuration daemon
# vxconfigd -k -r reset # additionally resets most of the VxVM drivers'
                        # internal data structures
# reboot # resets all of VxVM's internal data structures
```

The last command is usually a no-no and is only necessary if you encounter a bug in DMP or VxVM or if you are running Cluster Volume Manager, which is much more difficult to handle in special situations than plain VxVM.



no
Controller visible via cfgadm -la?
yes
Install device driver for HBA and/or FC
Device visible via cfgadm -o show_FCP_dev -la?
yes
no
luxadm -e create_fabric_device
cfgadm -c configure c#::wwpn
no
Device visible via format?
yes
devfsadm
SAN or hardware problem
no
Device visible via format?
yes
format or fmthard, write Solaris label vxdisk scandisks
Done
yes
Device visible in vxdisk list?
no
1) vxdctl enable
2) vxconfigd -k
3) vxconfigd -k -r reset
4) reboot

Figure 3-1: Flow chart outlining incorporation of new LUNs into Solaris and VxVM

3.1.5 Leaving Physics Behind – Welcome to VxVM!

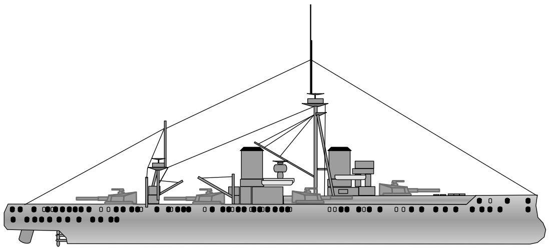
Now that the new disk is a fully valid member of the Solaris (or other OS) crowd we can finally incorporate it into VxVM, thus virtualizing it and leaving device physics behind (at least to the degree possible). Virtualizing a disk is done by one simple command:

```
# vxdisksetup -i <accessname>
# vxdisksetup -i c4t9d3
```

The **-i** flag will initialize the Private Region of the disk. If the disk had already been initialized before and need not be re-initialized you can omit the **-i** flag. For instance, in the Solaris OS, calling **vxdisksetup** without **-i** will just install the Private (and Public) partitions in the VTOC without initializing the Private Region.

There is one specialty with the accessname that Solaris admins sometimes do not get right at the first try: The access name does not include the slice number (slice 2) of the "whole disk" slice. It is merely a path like **c4t9d3** rather than **c4t9d3s2** The reason for this is VxVM's cross-platform interoperability: Other OSes simply do may not have slice addressing. For example, both HP-UX and AIX have a logical volume management incorporated into the kernel, so slices look really outdated to them. Since the command syntax is supposed to vary as little as possible between the different UNIX dialects only the common denominator — the standard addressing by controller, target, and disk — is used.

After treating the disk or LUN with **vxdisksetup -i** it now has a valid Private Region and can be incorporated into one and only one disk group. From now on it will be easy for many chapters until we get to the really hard stuff. The part where light speed is too slow for us. The part where buffer credits take a big toll on WAN performance. The part where disks fail and need to be replaced. But all of that is still a long way to go. Now it is just simply stuff that works, so let us look forward to it! You can skip the rest of the chapter and proceed with page 71 if you want. In the rest of this chapter we will only delve more deeply into VxVM disk formats and into what else can be done with the **vxdisk** command.


The Full Battleship

3.2 VxVM disk handling

3.2.1 VxVM Disk Formats

What we saw in the example above was this output:

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t10d0s2    auto:none       -            -            online invalid
```

The disks' type is **auto:none**, and their status is **online** and **invalid**. What do these values mean? While the access record should be obvious the type and status are not, so let's take a little diversion to find out more about those.

A type of **auto** means that VxVM has received the disk from the operating system by inquiring the disk device list. Old operating systems sometimes could not do this, so you had to define your own disk access records. But this is no longer necessary on any current hardware, so your devices should always be shown as "**auto**". The sub-type behind **auto:** refers to the initialization scheme or "format" that was used to incorporate the LUN into VxVM. For instance, **none** means that the disk does not have a Private Region. The subtype **sliced** means that Private and Public Region are located on different slices of the disk, and **cdsdisk** means that the whole disk is tagged as a Private Region. In our case, neither disk has a Private Region and both have been auto-discovered so they are both of type **auto:none**.

The status column shows two values: **online** and **invalid**. The first one refers to the operating status of this disk. A disk is **online** only if the OS can access it and the disk has not been offlined in software by the **vxdisk offline** command. This is how the software ON/OFF switch works:

```
# vxdisk offline c0t10d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
```

```
c0t10d0s2    auto           -           -           offline
# vxdisk online c0t10d0
# vxdisk list
DEVICE       TYPE           DISK        GROUP       STATUS
c0t0d0s2     auto:none      -           -           online invalid
c0t10d0s2    auto:none      -           -           online invalid
```

What does the **invalid** status refer to? It refers not to the general status but instead to the VxVM-only status. For instance, **invalid** means that a valid Private Region could not be found on this disk. If we create a Private Region by initializing the disk using **vxdisksetup -i**, then the invalid flag will go away, and the format will change (in this case, to **cdsdisk**):

```
# vxdisksetup -i c0t10d0
# vxdisk list
DEVICE       TYPE           DISK        GROUP       STATUS
c0t0d0s2     auto:none      -           -           online invalid
c0t10d0s2    auto:cdsdisk   -           -           online
```

3.2.2 CDSDISK AND SLICED

As you can see the invalid status has disappeared because a valid Private Region was found. The type and format of **auto:cdsdisk** expresses that the Private and Public Region do not reside in separate slices but the most modern disk format is used. CDS stands for Cross-platform Data Sharing. A CDS-disk is a disk that has been initialized by the **vxdisksetup** command with a Private Region spanning all of the disk. But there is more to it: The **vxdisksetup** command also created compatibility volume labels for all the major UNIX-based operating systems so that the disk can be used interchangeably between Solaris, Linux, AIX, and HP-UX. This is achieved by writing a Solaris-compatible VTOC at block zero (where Solaris expects its VTOC), and in addition writing an AIX-compatible volume label at the offset where AIX expects its volume label, and putting a third, HP-UX compatible volume label at the place where HP-UX expects its volume label. Linux compatibility comes for free because Linux can read Solaris VTOCs and will use it when it sees one. It was only possible to implement the CDS format because fortunately, these three volume labels do not reside in overlapping blocks. Windows, for example, uses block zero as the master boot record. This overlaps with the Solaris VTOC block and because the two formats are mutually incompatible the Windows version of VxVM will not recognize a CDS disk and cannot share disk groups with UNIX operating systems.

We can also initialize a disk with different formats than CDS. Before VxVM 4.0 the default format was **sliced**. To compare a sliced disk with a CDS disk let us initialize one of each and compare their VTOCs. First, a look at the CDS disk:

```
# vxdisksetup -i c0t10d0
# vxdisk list
DEVICE       TYPE           DISK        GROUP       STATUS
```

```
c0t10d0s2    auto:cdsdisk    -            -            online
# vxdisksetup -i c0t10d0 format=sliced
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t10d0s2    auto:sliced     -            -            online

# prtvtoc /dev/rdsk/c0t10d0s2
* /dev/rdsk/c0t10d0s2 partition map
*
* Dimensions:
*     512 bytes/sector
*     133 sectors/track
*      27 tracks/cylinder
*    3591 sectors/cylinder
*    4926 cylinders
*    4924 accessible cylinders
*
* Flags:
*   1: unmountable
*  10: read-only
*
*                          First     Sector    Last
* Partition  Tag  Flags    Sector     Count    Sector  Mount Directory
       2      5    01          0  17682084  17682083
       7     15    01          0  17682084  17682083
```

Note that partition 7 is identical to partition 2 in size and position – they both cover the whole disk. Only the tag is different: 5 (backup) for slice 2, and 15 (VxVM Private Region) for slice 7.

Let's see how much space is left for VxVM. In order to do that, we must put the disk into a DG and then check the DG for free extents:

```
# vxdg init adg adg01=c0t10d0
# vxdg -g adg free
DISK         DEVICE       TAG          OFFSET    LENGTH    FLAGS
adg01        c0t10d0s2    c0t10d0      0         17616288  -
```

OK, there are 17616288 blocks free on the CDS disk for VxVM. Now let's compare that to the sliced layout on the same disk.

```
# vxdg destroy adg
# vxdisksetup -i c0t11d0 format=sliced
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t10d0s2    auto:sliced     -            -            online
# prtvtoc /dev/rdsk/c0t10d0s2
* /dev/rdsk/c0t10d0s2 partition map
```

```
*
* Dimensions:
*     512 bytes/sector
*     133 sectors/track
*      27 tracks/cylinder
*    3591 sectors/cylinder
*    4926 cylinders
*    4924 accessible cylinders
*
* Flags:
*   1: unmountable
*  10: read-only
*
* Unallocated space:
*       First     Sector    Last
*       Sector     Count    Sector
*           0      3591      3590
*
*                           First     Sector    Last
* Partition  Tag  Flags    Sector     Count    Sector  Mount Directory
       2      5    01          0  17682084  17682083
       3     15    01       3591     68229     71819
       4     14    01      71820  17610264  17682083
```

We see that cylinder zero is left unallocated; the Private Region starts at sector 3591 which is exactly one cylinder offset from zero (see the "Dimensions" section at the beginning of the output as well as the "Unallocated space" section in the middle). The Private Region itself resides on slice 3 and has tag 15, while the Public Region resides on slice 4 and has tag 14 (the identifier tag for Public Region).

```
# vxdg init adg adg01=c0t11d0
VxVM vxdg ERROR V-5-1-6478 Device c0t10d0s2 cannot be added to a CDS disk group
```

Oops. The default format for disk groups is CDS. Only DGs that have the CDS flag can be cross-imported between platforms. In order for that to work, all the disks in such a DG must have the CDS layout. If we want to use sliced disks to create a DG, then we must turn the CDS flag for the disk group off by specifying **cds=off** when creating the DG.

```
# vxdg init adg adg01=c0t11d0 cds=off
# vxdg -g adg free
DISK         DEVICE       TAG          OFFSET     LENGTH    FLAGS
adg01        c0t10d0s2    c0t10d0      0          17610256  -
```

In this case we are left with 17610256 free blocks for VxVM. Due to the empty cylinder and varying internal size of the Private Region the free block count differs between CDS and **sliced** layout.

3.2.3 How to Mix CDS and Sliced Disks in a Disk Group?

Since the advent of the **cdsdisk** format there has been a lot of confusion about what to use when and how to mix sliced disks and **cdsdisks** in the same disk group. The concept is – as always – rather straightforward. But this has been obfuscated by lack of understanding of the concepts and by the confusing names: Disk media can be of CDS format, and disk groups can have a CDS flag set. They sound the same, but they are not the same. So let us clear this up and reveal its simplicity:

Idea: If disk metadata (VTOC, label) were cross-platform readable then because VxVM is also cross-platform, a VxVM disk group could be used cross-platform.

Implementation: VxVM for HP-UX knows how to write a HP-UX compatible label, VxVM for AIX knows how to write an AIX compatible label, VxVM for Solaris knows how to write a Solaris compatible label. Combine this cross-platform know-how and have **vxdisksetup** write all three label types on every disk.

Problem: If there is a legacy disk of **sliced** format in a disk group then this disk will not be cross-platform readable and accordingly, the disk group can not be imported on other platforms. Do we want VxVM to find out after importing 999 of 1000 disks in a disk group that the 1000th disk is sliced and the disk group cannot be imported? No, we don't. We want to know before trying to import a DG if the import is going to fail anyway.

Solution: Create an extra flag in the disk group data structure that can be set to ON if and only if all disks in the disk group are cross-platform shareable. Make this value the default for DG creation so that the DG's functionality is automatically enhanced by the CDS feature. If the users do not want CDS to work they can (or must) switch it off be setting the CDS flag to OFF for the DG like this:

```
# vxdg -g mydg set cds=off
```

Limitation: Whenever a sliced disk is added to a disk group that disk group's CDS flag must first be reset to OFF. Likewise, when creating a new DG from sliced disks the flag must be reset in the disk group creation process by specifying **cds=off**.

Unfortunately, the latter is not done automatically by VxVM.

Note that a Solaris **boot disk is always sliced** because the OS needs to find a partition with the root tag to boot from. So the CDS format cannot be used for the boot disk.

If a disk is initialized with a **sliced** layout then enough space is reserved to be able to convert it into a **cdsdisk** later. That is why you saw the first cylinder unallocated from the **sliced** disk in the example before. Conversion to **cdsdisk** layout is done by the utility **/usr/lib/vxvm/bin/vxcdsconvert**.

3.2.4 Other Disk Formats

In addition to **cdsdisk** and **sliced** there are two more formats: **none** and **simple**. The **none** format is actually not a real format, i.e. you cannot specify it to **vxdisksetup -i**. It is displayed if no Private Region can be found on the disk.

The **simple** format can be specified to **vxdisksetup** and yields a VTOC that looks similar to that of a **cdsdisk**, i.e. it has Private and Public Region together in one single, large slice. The difference is that the slice is not slice 7 as for a **cdsdisk**, but slice 3, and the first cylinder is again unallocated in case a later conversion to **cdsdisk** is desired. A **simple** disk also does not hold the AIX and HP/UX compatibility labels.

Come to think of it: why would anyone ever use a sliced disk layout except for a boot disk, where it is required to use individual slices? Keeping Private and Public Regions separate in a way that is visible to the operating system and user is not optimal and seems unnecessary. Use **cdsdisk** whenever possible!

3.2.5 Encapsulation Overview – Integrating Legacy Data

The **vxdisksetup** command alters the partition table in a rather radical way: all previous data partitions are wiped off the disk unless one of them is currently mounted! If we do not want to copy the data from our partition-based file system to a new volume-based one, how do we get the data under VxVM control?

The answer is encapsulation. This is a different way of bringing a LUN or disk under VxVM control. The idea is to have VxVM find some free space on the disk where it can put its Private Region (which is only a few MB in size), and find one (for **simple** layout) or two (for **sliced** layout) free slots in the VTOC in order to store the pointers to the Private Region and Public Region. Then, the encapsulation process reads the information about existing extents (i.e. partitions or slices) from the VTOC and maps them into VxVM subdisk objects (which are, as you know, nothing but extents). It then writes them into the disk group's Private Region database. Additionally, so that the user can access the subdisks, each subdisk is put into a straight concat plex which is then put into a normal volume object. All this is done by one simple command: **vxencap**. The names for the virtual objects are appropriately chosen by the encapsulation command if they can be derived from their partition tags: **rootvol, swapvol** etc. or they are simply derived from the controller paths that were used to mount them.

To sum up, encapsulation consists of the following steps:

1) Allocate free space for the Private Region
2) Allocate free VTOC slots for Public and Private Region
3) Map existing slices to subdisks and associate them with plexes and volumes
4) Remove all slices from the VTOC that are not required during the boot phase
5) Modify **/etc/vfstab** to update slice mounts to mounts of the new volumes
6) Demand a reboot from the user if the root disk has been encapsulated, or wait for the user to reboot if a data disk has been encapsulated.

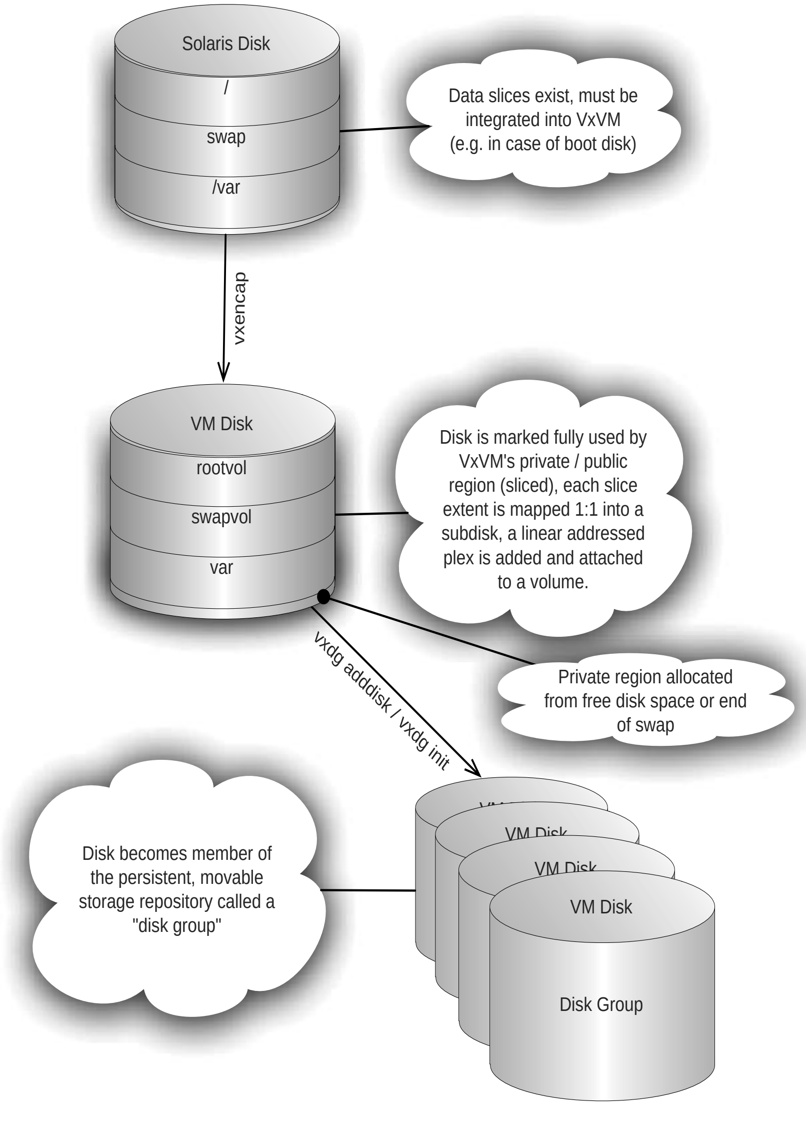

Solaris Disk
/
swap
/var
Data slices exist, must be integrated into VxVM (e.g. in case of boot disk)
vxencap
VM Disk
rootvol
swapvol
var
Disk is marked fully used by VxVM's private / public region (sliced), each slice extent is mapped 1:1 into a subdisk, a linear addressed plex is added and attached to a volume.
vxdg adddisk / vxdg init
Private region allocated from free disk space or end of swap
Disk becomes member of the persistent, movable storage repository called a "disk group"
VM Disk
VM Disk
VM Disk
Disk Group

Figure 3-2: Encapsulating a disk creates volumes from existing partitions by allocating subdisks that line up exactly with the original location of the slices, associating these subdisks with plex objects and attaching them to volume objects.

After the reboot you can mount the new volumes at the same place where you used to have the slices mounted. Because they are no longer bound to their slice legacy you can now do cool stuff with the volumes, like change their layout, mirror them, resize them or move them to somewhere else. Of course this does not work well with those volumes that are required by the boot process because their VTOC entries must not be deleted and therefore these volumes are stuck to legacy behavior. There are ways around this but they are what hackers describe as **non-trivial**.

Encapsulation of a disk is performed by issuing the **vxencap** command and passing it the right parameters: accessname, target DG and – if applicable – a flag to create the target DG and its format. Here are two examples, the first one for a standard disk, the second one for the root disk:

```
# vxencap -g mydg mydg02=c0t2d0  # mydg exists, so add this disk to it
# vxencap -g bootdg -f sliced -c rootdisk=c0t0d0 # -c: create new "bootdg"
```

Read more about encapsulation in Albrecht Scriba's dedicated chapter beginning on page 319, which should leave no question about encapsulation unanswered. It takes you on a tour so low-level that it will show you how to encapsulate a whole disk manually, without using the **vxencap** command! It will show you precisely and in detail how VxVM's volume management works, how it addresses its storage and how one can incorporate data from any other volume management software provided the basic addressing schemes are compatible (i.e. no bit-slicing or content-addressing or similarly weird stuff is involved.

3.2.6 Summary

You should now know much more than you ever wanted to know about initializing disks for VxVM. If you are still hungry for more, please read the man pages for **vxdisk** and **vxdisksetup**. There is a lot more to **vxdisk** than what is written here, but this book was intended not to become more than twice the weight of a 2008 notebook computer...