CHAPTER 10: ENCAPSULATION AND ROOT MIRRORING

by Dr. Albrecht Scriba

10.1 Introduction and Overview

Within the previous chapters of this book, we always created volumes based on freshly initialized disks. So from the application's point of view, the content of the volumes was uninitialized. Of course, there was some kind of data on the disks (any sequence of bits), but we didn't bother to restore them to an application usable state. Instead we created file systems after volume creation, initialized a starter database, and so on.

Now we turn to a somewhat different procedure to create volumes on already existing application data, so we can move raw device control to VxVM in order to apply all the nice features of an advanced volume management: adding redundancy, resizing, relayouting for performance reasons, online migration to another storage array, etc.

This procedure is called encapsulation, and you can hear a lot of strange myths about it. But the basic steps of this procedure are surprisingly simple, as we will see in this introductory chapter. Only two details need further investigation. They are presented in the "Technical Deep Dive" section: subdisk alignment and the infamous "B0" Ghost subdisk (see page 330) that often appears on encapsulated disks.

V. Herminghaus and A. Sriba, *Storage Management in Data Centers*,
DOI: 10.1007/978-3-540-85023-6_10, © Springer-Verlag Berlin Heidelberg 2009

So why did we call the basic steps surprisingly simple? Because all kinds of volume management have one thing in common, as shown in chapter 1: They all store application data in extents, i.e. in an ordered list of contiguous disk regions (e.g. logical partitions of AIX). Encapsulation basically places VxVM subdisks (just another kind of extent) exactly over the preexisting extents containing application data and orders these subdisks in exactly the same manner within a VxVM plex, as they were ordered in the former volume management. Adding a volume layer to this plex leads to an application driver showing the same data as before.

The very simplest way to store application data is a Solaris partition, so we only have one extent on one disk. This chapter focuses on the procedure of encapsulating a Solaris partition, although it might be adapted to more complex data structures of other volume managers.

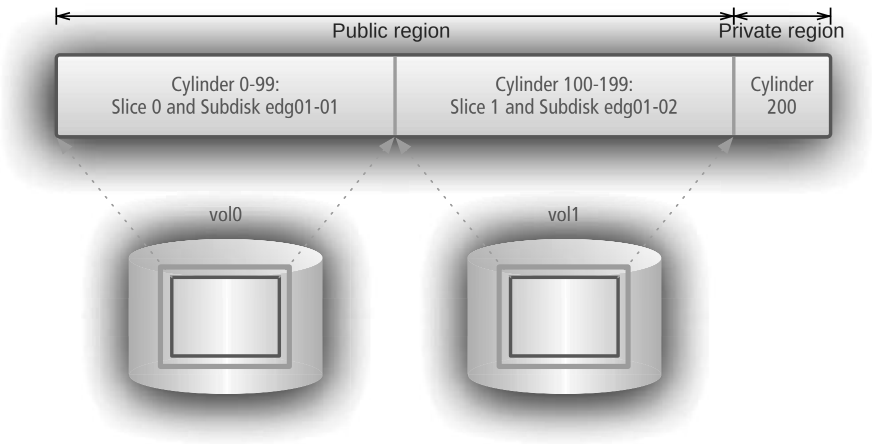


Figure 10-1: Encapsulation technique (slightly simplified)

Just to clear up whatever misunderstandings may remain: When an application (usually the file system) does I/O to a device there is always some kind of mechanism that maps the block number or extent specification to the physical layer of the device. For a Solaris partition, this function is almost trivial: it simply adds the offset of the partition start to the block number to get to the physical block. It needs meta information from the VTOC in order to do that. If we create a virtual object in VxVM that translates the logical block numbers of the virtual object to the same physical block numbers, then we can access the data by either of the two means: via partitioning and via VxVM. We could even add a third volume management product and add meta info for that, too, like Sun LVM (aka SDS). Of course we must not access the data in an uncoordinated way or risk losing data integrity. But again: In principle, all you need is meta data for the current device driver that points to the same extents (or maps the extents in the same way) as the original data and we can encapsulate whatever we want.

Placing a subdisk over a Solaris partition requires, as we already know, two preceding steps. We must prepare the disk for VxVM use (creating private and public region and initializing the former with a basic data structure), and we must build a disk group out of this disk or add the disk to an existing disk group. In other words: You cannot define VxVM objects without a disk group containing at least one active configuration copy within the private region.

Here is a commented list of actions that need to be performed in order to add a VxVM volume interface to already existing application data. Currently, we will not discuss the individual command lines. This is left to the "Technical Deep Dive" section of this chapter. Consider the following list as a look behind the scenes. You do not necessarily have to understand all the steps if you just want to encapsulate a few disks. There is a simple command for that, which is discussed right after this list. But you may eventually require this look behind the scenes if you want to really understand encapsulation, if you run into problems with encapsulating, or if you wish to encapsulate in a non-standard way (e.g. not all of the partitions of a disk).

10.2 The Secrets of Encapsulation

1. In addition to the existing partitions (the ones to be encapsulated) we create the well known VxVM partitions. For the **cdsdisk** layout a slice with tag 15 is put over the whole disk . For the **sliced** layout, a slice with tag 15 is put over an unused disk cylinder anywhere on the disk for the private region and a slice bearing tag 14 over the remainder of or over the whole disk (see below) as the public region. So it is obvious, that you cannot encapsulate a disk without some free space to put the private region on. Furthermore, you need one (for **cdsdisk**) or two (for **sliced**) unused partitions in order to keep all applications running during these preparatory steps. Otherwise you would need to stop at least some applications using some partitions on that disk, to remember offset and length of these partitions and to remove them. Finally, a **cdsdisk** layout cannot be used under the already mentioned restrictions: disks not supporting SCSI mode sense (such as IDE) and OS or EFI disks. In addition, since the offset of the private region of a CDS disk must be 256 sectors, we cannot encapsulate a partition located in that region.

2. We initialize the private region with the appropriate basic data set depending on the VxVM partition layout we created in step 1 (**cdsdisk** or **sliced**). Steps 1 and 2 resemble the usual procedure of disk initialization using **vxdisksetup**.
3. We create a new disk group out of this initialized disk or add the disk to an existing disk group. Note that there is no difference to the standard way of creating or adding to disk groups.
4. We define subdisks with the corresponding offset and length over the partitions we want to encapsulate, associate them with plexes, create volumes for the plexes and attach the plexes into the volumes, then start the volumes in order to enable I/O. Now we have an active volume driver for each encapsulated partition. Since **vxassist**, the easy to handle default top-down tool for volume creation, is unable to specify the exact offset of the subdisks, we have to resort to using low-level **vxmake** commands.
5. During all these steps applications can remain online until we want to switch to the freshly created volume drivers for I/O. Unfortunately, due to OS restrictions, we cannot replace the legacy drivers with the volume drivers while the application is running. So we must stop the applications and restart them using the volume drivers this time. You may adapt the **vfstab** as well or specify the new raw device drivers for your database.
6. In case you do not need the legacy partitions anymore, you may remove them from the disk's VTOC.
7. Now freedom awaits you! Your disks, disk groups, and volumes created by encapsulation of partitions behave exactly the same as standard disks, disk groups, and volumes. You may convert to CDS disks and a CDS disk group, add new disk members, create redundancy to the volumes, resize or relayout them, add logs, and so on — all that without interrupting running applications anymore.

Fortunately, VxVM provides a script called **vxencap** that performs all these steps in collaboration with the run level script **/etc/rcS.d/S86vxvm-reconfig** (Solaris 9) or by restarting the **svc:/system/vxvm/vxvm-reconfig** FMRI in Solaris 10. Here is the standard procedure for application and OS disks together with some comments:

```
# vxencap -g <DG> [-c] <dmname>=c#t#d#
```

This command collects data from the parameters (disk access, disk media, and disk group name; for optional layout parameters see the "Full Battleship" part) and from the disk itself (disk's VTOC) and stores them together with the new VTOC and a command summary in ASCII files under **/etc/vx/reconfig.d/disk.d/c#t#d#**. The **-c** option is used to create a new disk group, otherwise the disk is added to an already existing disk group. The script defaults to creating **cdsdisk** layout (and a CDS disk group), if possible and if not instructed otherwise by further parameters. In case you encapsulate an OS disk of Solaris 9, the **logging** mount option of some OS partitions in the last field of the vfstab will be changed to **nologging** (see the "Full Battleship" part).

Be aware that the **vxencap** command does not change the partition table of the disk yet. This job is done by the additional init-scripts supplied by VxVM. Nothing happens to the disk 's VTOC before you go on to the next step:

```
# init 6          # reboot the system
```

During the next boot, the run level script **/etc/rcS.d/S86vxvm-reconfig** (Solaris 9) is invoked during the boot process. In Solaris 10, this is integrated into the service framework as the FMRI **svc:/system/vxvm/vxvm-reconfig**. It reads the disk configuration files created above and performs the necessary tasks to bring all data partitions under VxVM control by encapsulating them. Unfortunately, the disk group is always marked as boot disk group, even in case of a simple application disk. In the latter case, you should clear this entry by issuing the **vxdctl bootdg** command with the appropriate parameter: either the correct boot disk group or the reserved word **nodg** if there is no boot disk group yet (i.e. if your root disk is not yet encapsulated).

Since the Solaris OpenBoot PROM only recognizes partitions, encapsulating an OS disk will, of course, not remove those partitions that are required during the early boot process. But an OBP alias for the OS disk is created called **vx-<dmname>**. Note that the OBP **boot-device** list is not updated. Instead of a reboot, you may stop all applications using this disk if they are not required by the OS (unfortunately, an **umount /** command will not work, even with the **-f** option), execute the run level script manually with the start parameter, and restart your applications. If you compare the output of this script with the seven steps mentioned above, you can easily map them. Here's a walk-through of encapsulating a Solaris disk without rebooting. All partitions of the disk have been unmounted before we start:

```
# vxencap -g edg -c edg01=c2t4d3
  The c2t4d3 disk has been configured for encapsulation.
# /etc/rcS.d/S86vxvm-reconfig start
  VxVM vxvm-reconfig INFO V-5-2-324 The Volume Manager is now reconfiguring
(partition phase)...                                        (step 1)
  VxVM vxvm-reconfig INFO V-5-2-499 Volume Manager: Partitioning c2t4d3 as an
encapsulated disk.                                          (step 1)
  VxVM vxvm-reconfig INFO V-5-2-323 The Volume Manager is now reconfiguring
(initialization phase)...                                           (step 2)
  VxVM vxvm-reconfig INFO V-5-2-497 Volume Manager: Adding edg01 (c2t4d3) as an
encapsulated disk.                                          (step 3)
VxVM vxcap-vol INFO V-5-2-89 Adding volumes for c2t4d3...   (step 4)
Starting new volumes...                             (step 4)
VxVM vxcap-vol INFO V-5-2-444 Updating /etc/vfstab... (step 5)
Remove encapsulated partitions...                           (step 6)
```

10.3 Root Disk Encapsulation

Encapsulating the root disk is performed in a very similar way to encapsulating a normal data disk. However, it is different in that the target disk group normally does not exist yet, therefore it needs to be created on-the-fly by the **vxencap** command. This is done by supplying the parameter for "create disk group "**-c -g <dgname>**" to **vxencap**.

So the complete command chain for encapsulating the root disk is this:

```
# vxencap -c -g osdg -c rootdisk=c0t0d0
 The c0t0d0 disk has been configured for encapsulation.
# init 6
<System rebooting…>
```

As you can see, after successful encapsulation (including the reboot that is necessary to switch the access path from the standard **/dev/dsk/c#t#d#s#** devices to volume paths like **/dev/vx/dsk/rootvol**) the **/** file system is now mounted from a volume manager volume:

```
# df -k /
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/bootdg/rootvol
                     6196278 3534270 2600046    58%    /
```

The volume manager path to all boot file systems has automatically been persisted into the **/etc/vfstab** file:

```
# grep rootvol /etc/vfstab
/dev/vx/dsk/bootdg/rootvol  /dev/vx/rdsk/bootdg/rootvol  /  ufs  1  no  logging
#NOTE: volume rootvol () encapsulated partition c0t0d0s0
```

10.4 Root Disk Mirroring

Encapsulation seems to be an interesting thing from a technical point of view. But until now, no convincing advantages of OS disks under VxVM control are implemented. The usual counterpart of OS disk encapsulation, the root disk mirroring, is still missing.

Against several misunderstandings, we emphasize that root disk mirroring is NOT based on a different technique compared to regular volume mirroring. The OS mirror is NOT a physical copy of the encapsulated OS disk, therefore, it may be placed on completely different disk hardware. Once again: The physical position of the subdisks is independent from their virtual position within the plex. The only restrictions implemented in the **vxassist** command are reasonable: no striping in a plex or mirroring in a volume based on one disk device.

The regular mirror procedure (**vxassist mirror**) keeps plex layout attributes, while the plex internal subdisk concatenation is ignored. We may conclude that size and position of private and public region may not correspond on both disks, that the physical position of the mirror subdisks are very probably not identical to those on the original disk, and that the concatenation of the strange ghost subdisk (more on that in the technical deep dive beginning on page 330) and the main subdisk within a plex is not repeated on the mirror disk.

Nevertheless, mirrors of OS volumes differ in one **additional** feature: In order to boot from the disk providing the mirrored subdisks, they need partitions defined at exactly the same position. You may call this a reversed encapsulation: While encapsulation defines subdisks over partitions on the original disk, reverse encapsulation defines partitions over subdisks on the mirror disk.

Implementing an OS mirror basically does not differ from the regular volume mirroring: We need another disk device (due to boot capabilities this disk must use the **sliced** format), add it to the boot disk group and mirror the volumes. However, if you just run **vxassist mirror** on all the boot volumes, then the OBP device aliases are not updated to enable booting from the mirror disk, and the VTOC on the target disk will not be updated with the slice information pertaining to the newly created subdisks. I.e. the VTOC will not contain slices for those file systems which are required during the boot phase, and so the new boot mirror will not be actually bootable. This is because the step that we called reverse encapsulation is never performed by vxassist.

You can execute reverse encapsulation by calling the script **vxbootsetup -g <bootdg>**. Or you can make use of the **vxmirror** script for mirroring the boot volumes. The **vxmirror** script will automatically call **vxbootsetup** after mirroring all boot volumes. It also creates an OpenBoot PROM device alias as a mnemonic to enable easy booting from the alternate disk. The following steps mirror the boot disk after it has been successfully encapsulated:

```
# vxdisksetup -i c0t2d0 format=sliced privlen=1m # prepare a boot mirror
# vxdg -g osdg adddisk osdg02=c0t2d0     # add it to the boot disk group
# vxmirror -g osdg osdg01       # mirror everything and reverse encapsulate
! vxassist -g osdg mirror rootvol
! vxassist -g osdg mirror swapvol
! vxassist -g osdg mirror var
! vxassist -g osdg mirror opt
! vxbootsetup -g osdg   # this script does reverse encapsulation and devaliases
```

If you look at the VTOCs of the two bootable disk mirrors now, you will see that they differ significantly. This should prove that boot disk mirroring is definitely not a physical copy of a the boot disk, but merely a normal volume-by-volume copy, plus the reverse encapsulation:

```
# prtvtoc -h /dev/rdsk/c0t0d0s2
       0      2    00    4198320  12586800  16785119
       1      3    01          0   4198320   4198319
       2      5    00          0  78156480  78156479
       3     14    01          0  78156480  78156479
       4     15    01   78148320      8160  78156479
       5      7    00   16785120   4198320  20983439
       6      0    00   20983440  12586800  33570239
# prtvtoc -h /dev/rdsk/c0t2d0s2
       0      2    00    4206480  12586800  16793279
       1      3    01       8160   4198320   4206479
       2      5    00          0  78156480  78156479
       3     14    01       8160  78148320  78156479
       4     15    01          0      8160      8159
```

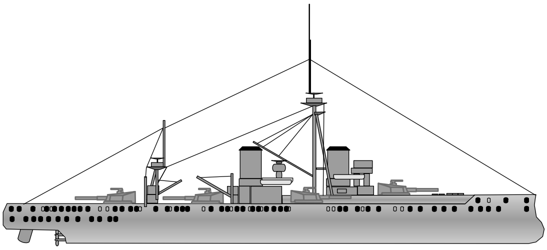
```
       5      7    00   16793280   4198320  20991599
       6      0    00   20991600  12586800  33578399
```

You can verify that the root file system (and **/usr, /var,** and **swap** as well) now contain two plexes, i.e. they are mirrored and therefore failsafe:

```
# vxprint -rtg osdg rootvol
[…]
v  rootvol      -            ENABLED  ACTIVE   4198320  ROUND     -        root
pl rootvol-01   rootvol      ENABLED  ACTIVE   4198320  CONCAT    -        RW
sd osdg01-B0    rootvol-01   osdg01   78148319 1        0         c0t0d0   ENA
sd osdg01-01    rootvol-01   osdg01   0        4198319  1         c0t0d0   ENA
pl rootvol-02   rootvol      ENABLED  ACTIVE   4198320  CONCAT    -        RW
sd osdg02-01    rootvol-02   osdg02   0        4198320  0         c0t2d0   ENA
```

For our convenience, the **vxbootsetup** program (the final command called internally by **vxmirror**) has created device aliases in the Solaris Boot PROM so that we can boot from either one of the mirrors by addressing them symbolically (**vx-odsg01** and **vx-osdg02**):

```
# eeprom nvramrc
nvramrc=devalias vx-osdg01 /pci@1f,0/ide@d/disk@0,0:a
devalias vx-osdg02 /pci@1f,0/ide@d/disk@2,0:a
```



The Full Battleship

10.5 Remarks to vxencap and OS Mirroring

The script **vxencap** provides some built-in intelligence to choose the proper disk layout: If possible, it prepares for a **cdsdisk**, otherwise for a **sliced** layout. We recall the reasons that may prevent using **cdsdisk** layout: no SCSI mode sense support, OS or EFI disk, or a data partition to be encapsulated at the beginning of the disk. If you want to force a **sliced** layout, you can make use of the option **-f**:

```
# vxencap -g <diskgroup> [-c] -f sliced <dmname>=c#t#d#
```

Unlike the default of VxVM 5.x for a private region size of 32 MB, **vxencap** will create a default private region length of 1 MB (or rounded up to the next cylinder boundary, if **sliced**). Assuming that encapsulation is performed mostly on the OS disk with its simple data structures, this default is indeed reasonable. In case you want to specify a different private region size, just use the option **-s**:

```
# vxencap -g <diskgroup> [-c] -s <size> <dmname>=c#t#d#
```

In order to promote other than the built-in defaults, enter the desired key-value pairs into **/etc/default/vxencap**:

```
format=sliced
privlen=4096
```

Regarding the VxVM object names to be created by way of encapsulation, **vxencap** allows for specification only of the disk group and the disk media name. We may guess, and we guess quite correctly, that the subdisk names are derived from the disk media name (**<dmname>-##**) and the plex names from the volume names (**<volname>-##**), as usual. But what about the volume names? In case of an OS disk the root device is named **rootvol** and the swap device is named **swapvol**. Other partitions of an OS disk are named after the last part of the current mount directory. In case of a non-OS disk, the volumes to be created are named after the disk media name, the partition number, and the usual **vol** extension (**<dmname><partition#>vol**).

The root disk of Solaris 9 encapsulated by VxVM in conjunction with the **logging** option of **ufs** on the root device generates subsequent kernel panics, when rebooting after a system crash. The cause is a well-known and aging programming error. Two workarounds are available. Either you do not install the current Solaris patch **113073-14** (or **113073-13**) in favor of the rather old version **113073-08**. Or you accept the modifications of **/etc/vfstab** performed by **vxencap** which turned the **logging** options of the root and the swap device into **nologging**. No solution is satisfactory: The first one is not tolerated by Sun support, the second one discards the file system logging feature, thus noticeably delaying the boot process after a system crash.

An encapsulation procedure initialized by **vxencap** requires some prerequisites fulfilled, as already mentioned in the "Easy Sailing" part: At least one (**cdsdisk** layout) or two (**sliced**) unused partition numbers must be available, and at least one disk cylinder must not be part of an OS or application partition in order to form the private region. While the former restriction can only be ignored by way of an exhaustive low-level procedure (see the "Technical Deep Dive" part), the latter provides a special exception built into **vxencap**. An OS disk providing the root device contains mostly also the swap device. A swap device does not hold data required after a reboot except for the memory pages dumped in case of a kernel panic. Therefore, the required space to form the private region may be and indeed will be cut off from the swap device by **vxencap**, if all disk cylinders of the OS disk are in use.

Although the **vxencap** script does not bother for peculiar partition numbers, you should ensure that the root device is stored on partition 0 and the swap device on partition 1. Otherwise, when creating the OS mirror by the standard **vxmirror** command, the invoked **vxbootsetup** script will completely fail, for it is strictly bound to the mentioned partition numbers.

Any encapsulation via **vxencap** and the related VxVM reconfiguration script will try to set the VxVM **bootdg** attribute. Quite correct, if your first encapsulation task points to the root disk! But wrong in any other case! Then, you should clear the **bootdg** attribute (stored in **/etc/vx/volboot**) before encapsulating the OS disk. Note that the **bootdg** disk group name is a reserved name, because it is a symbolic link to the actual boot disk group name under **/dev/vx/rdsk** and **/dev/vx/dsk**, respectively.

```
# vxdg bootdg
edg
# vxdctl list | grep bootdg
bootdg:  edg
# grep bootdg /etc/vx/volboot
bootdg edg
# vxdctl bootdg nodg
```

We already mentioned another weakness of the reconfiguration script: The OpenBoot PROM attribute **boot-device** is not updated during encapsulation. Well, assuming the default device alias entry **disk** typically pointing to the device we encapsulated, we encounter no restrictions. But the mirror disk provided with partitions and an NVRAM device alias by **vxbootsetup** to boot from should be added to the **boot-device** list. Unfortunately, the task must be executed manually:

```
# eeprom boot-device
```

```
boot-device=disk net
# eeprom boot-device='vx-osdg01 vx-osdg02 disk net'
```

VxVM 5.0 "delights" us with another programming error in **vxbootsetup**: The device alias definitions stored in the OpenBoot PROM NVRAM are not concatenated line-by-line as it should be, but by spaces, thus invalidating all but the first entry. Once again, repair it manually in order to use the aliases.

```
# eeprom nvramrc
nvramrc=devalias vx-osdg01 /pci@1f,0/ide@d/disk@0,0:a devalias vx-osdg02 /
pci@1f,0/ide@d/disk@2,0:a
# eeprom nvramrc='devalias vx-osdg01 /pci@1f,0/ide@d/disk@0,0:a
  devalias vx-osdg02 /pci@1f,0/ide@d/disk@2,0:a'
# eeprom nvramrc
nvramrc=devalias vx-osdg01 /pci@1f,0/ide@d/disk@0,0:a
devalias vx-osdg02 /pci@1f,0/ide@d/disk@2,0:a
```

Sometimes, you may read or hear the recommendation to mirror the OS disk by the script **vxrootmir**. We do not recommend so, because **vxrootmir <mirror-dmname>** just mirrors **rootvol** and provides the mirrored disk with an appropriate partition and an OpenBoot PROM device alias by invoking **vxbootsetup**. No other OS volumes are mirrored!

Technical Deep Dive

10.6 The Ghost Subdisk

Encapsulating a disk means placing volumes together with their related plexes and subdisks over partitions. So we expect a simple volume layout containing one plex each and just one subdisk within the latter exactly corresponding to the partitions. Nevertheless, in most cases we discover one strange volume whose plex contains two subdisks, one of them only one sector in size.

```
# vxprint -rtg osdg rootvol
[…]
v  rootvol      -            ENABLED  ACTIVE   815176   ROUND     -        root
pl rootvol-01   rootvol      ENABLED  ACTIVE   815176   CONCAT    -        RW
sd osdg01-B0    rootvol-01   osdg01   815175   1        0         c0t0d0   ENA
sd osdg01-01    rootvol-01   osdg01   0        815175   1         c0t0d0   ENA
pl rootvol-02   rootvol      ENABLED  ACTIVE   815176   CONCAT    -        RW
sd osdg02-01    rootvol-02   osdg02   0        815176   0         c0t2d0   ENA
```

This small subdisk, sometimes called "Ghost subdisk", is indeed cloak-and-dagger at first sight, but quite easy to understand at second thought as an unavoidable protection against disk failure for VxVM disks.

The volume table of contents of the disk (VTOC) which is located at the very first sector of the disk, stores the partition table and some disk attributes. It is strictly necessary for normal disk I/O operations, for the device drivers need partition information to calculate I/O offsets. A damaged VTOC requires immediate recovery by re-labeling the disk.

```
# prtvtoc -h /dev/rdsk/c4t6d0s2
       2      5    01          0  35368272  35368271
       7     15    01          0  35368272  35368271
# dd if=/dev/zero of=/dev/rdsk/c4t6d0s2 bs=512 count=1
# prtvtoc -h /dev/rdsk/c4t6d0s2
prtvtoc: /dev/rdsk/c4t6d0s2: Unable to read Disk geometry
# format c4t6d0
[…]
format> label
Ready to label disk, continue? yes

format> quit
# prtvtoc -h /dev/rdsk/c4t6d0s2
       0      2    00          0    263872    263871
       1      3    01     263872    263872    527743
       2      5    01          0  35368272  35368271
```

```
       6      4    00     527744  34840528  35368271
```

As the example above demonstrated, the VTOC may be overwritten by standard device drivers. We have chosen the backup partition (slice 2) covering the whole disk which is not a device for regular I/O operations. Nevertheless, even a regular data partition may contain the VTOC. And indeed, most formatted disks provide a partition starting at cylinder 0, thus including the VTOC into the partition space. Therefore, the block device drivers for partitions skip the first 16 sectors of their raw device partition, the main super block being the first file system object is always placed at sector 16 of the raw device. Why 16 sectors, not just one? Well, not only the VTOC needs protection, but also a possible boot block stored at sector 1 to 15 of the root partition.

```
# fstyp -v /dev/rdsk/c4t6d0s0
ufs
magic   11954   format  dynamic time    Sat Oct  4 08:24:30 2008
sblkno  16      cblkno  24      iblkno  32      dblkno  2240
[…]
```

All the same, even the swap device driver skips the first 16 sectors (although a boot block is of no use on a swap device).

```
# swap -l
swapfile             dev   swaplo blocks  free
/dev/dsk/c0t0d0s1    32,1      16 2097632 2097632
```

Even a disk under VxVM control must not overwrite the VTOC. The **cdsdisk** layout of a VxVM disk always skips the first 256 sectors of a disk, thus protecting not only the Solaris VTOC, but also other OS specific structures of other operating systems.

```
# vxdisk list c4t1d0 | grep ^private:
private:   slice=2 offset=256 len=2048 disk_offset=0
```

A freshly initialized VxVM disk of **sliced** layout even skips the first cylinder of a disk by starting the private region at cylinder 1 and defining the public region on the remainder of the disk. What is more, the active private region skips the first sector of the private region as partition for a reason we will explain later.

```
# prtvtoc -h /dev/rdsk/c4t3d0s2
       2      5    01          0  35368272  35368271
       3     15    01       4712      4712      9423
       4     14    01       9424  35358848  35368271
# vxdisk list c4t3d0 | grep ^private:
private:   slice=3 offset=1 len=4455 disk_offset=4712
```

Unlike the **cdsdisk** layout, the start position of the private region of the **sliced** layout is not fixed. In fact, the private region may be placed at ANY position on the disk, not only at the beginning (after first cylinder, default) or the end of the disk (**vxdisksetup -ie c#t#d#**;

see the next topic of the current chapter). Consequently, the public region may cover the first cylinder together with the VTOC. Just an example of a (small) sliced disk whose private region is located at the end of the disk (not created by **vxdisksetup -ie** ...):

```
# prtvtoc /dev/rdsk/c4t8d0s2
[…]
*    4712 sectors/cylinder
*     176 cylinders
*     174 accessible cylinders
[…]
*                           First     Sector    Last
* Partition  Tag  Flags    Sector     Count    Sector  Mount Directory
       2      5    01          0    819888    819887
# fmthard -d 3:15:01:$((819888-4712)):4712 /dev/rdsk/c4t8d0s2
# fmthard -d 4:14:01:0:$((819888-4712)) /dev/rdsk/c4t8d0s2
# vxdisk -f init c4t6d0 format=sliced
# vxdisk list
[…]
c4t6d0s2     auto:sliced     -            -            online
# prtvtoc -h /dev/rdsk/c4t8d0s2
       2      5    01          0    819888  819887
       3     15    01     815176      4712  819887
       4     14    01          0    815176  815175
```

Red alert! A subdisk placed over the VTOC but arranged at a plex offset greater than 0 by way of subdisk concatenation may receive block or swap device I/O, thus overwriting the VTOC and making disk and volume unusable (VxVM 4.x, VxVM 5.0 keeps partition information in the kernel until the next reboot). The following picture uses the sector and cylinder numbers of the command output above. Note that it does not show an actual configuration possibility of VxVM, for VxVM does not allow a subdisk to be placed over the VTOC by a technique still to be explained.

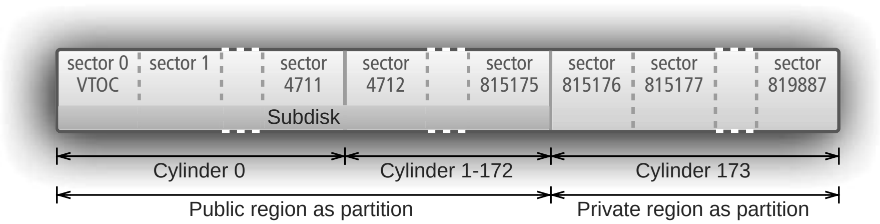


Figure 10-2: A subdisk covering the VTOC (virtual example)

Now assume two disks initialized by VxVM in the layout above and the subdisks of each disk concatenated within a plex.

Figure 10-3: **A VxVM volume built on subdisks concatenated within the plex and the location of the VTOCs covered by the subdisks (virtual example)**

Red alert, indeed! The VTOC being part of the second subdisk is located in the middle of the virtual address space provided by the plex. Even a block or a swap device driver skipping the first 16 sectors of the device (now a volume device!) may overwrite the VTOC of the second disk.

Therefore, any virtual volume manager providing concatenate or stripe capabilities MUST protect the VTOC, for it is not protected by the device drivers anymore. VxVM has implemented a quite simple solution: The start sector of the public region as subdisk container (not as partition) is moved one sector rearwards, thus starting immediately after the VTOC.

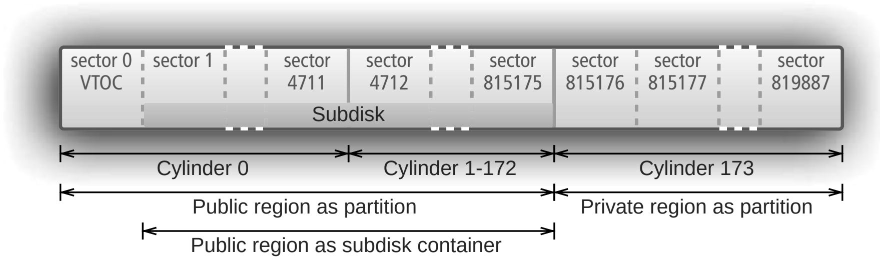


Figure 10-4: **Protecting the VTOC by moving the public region as subdisk container one sector rearwards**

Note that the subdisk offsets of the disk group configuration are calculated against the public region as subdisk container. Therefore, the subdisk drawn in the figure above has an offset of 0 sectors. Some commands to verify the disk layout and the subdisk position:

```
# vxdisk list c4t8d0
[…]
public:    slice=4 offset=1 len=815175 disk_offset=0
private:   slice=3 offset=1 len=4711 disk_offset=815176
[…]
# vxdg init adg adg01=c4t8d0 cds=off
# vxdg -g adg set align=1
# vxdg -g adg free
DISK         DEVICE       TAG          OFFSET    LENGTH    FLAGS
adg01        c4t8d0s2     c4t8d0       0         815175    -
# vxmake -g adg sd adg01-01 disk=adg01 offset=0 len=815175
# vxprint -stg adg
SD NAME         PLEX         DISK     DISKOFFS LENGTH   [COL/]OFF DEVICE   MODE
[…]
sd adg01-01     -            adg01    0        815175   -         c4t8d0   ENA
```

We immediately recognize a disadvantage of this solution: The public region and the subdisk is one sector smaller than before. Well, you might believe that we easily could squander one sector. No, there is indeed a realistic scenario where we urgently need this sector. Assume an OS disk completely in use except for one cylinder we kept for the private region (or cut of from the swap partition by VxVM tools during encapsulation). The following figure just shows the OS root partition, for multiple partitions do not modify the basic problem:

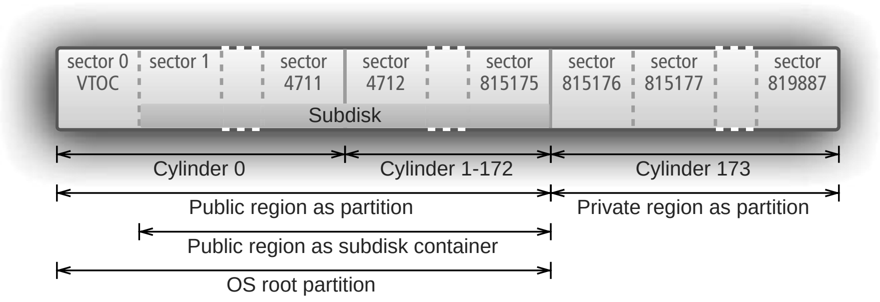


Figure 10-5: OS root partition not completely covered by the subdisk

The root volume based on the subdisk is one sector smaller than the root partition. However, one sector less in size is not the main problem. Look at the data at the beginning of the root device **as partition**:

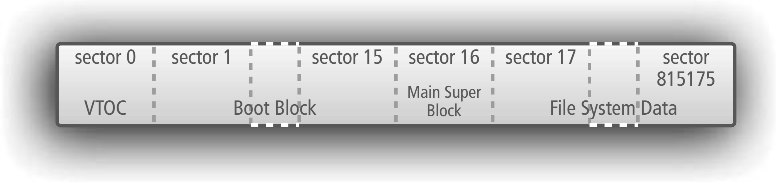


Figure 10-6: File system structure of the root device as partition

Now we compare it to the data at the beginning of the root device **as volume** based on the subdisk starting at disk offset 1 (which is offset 0 within the public region as subdisk container):

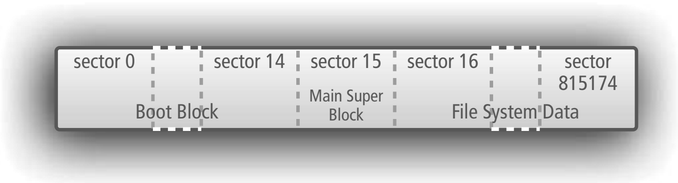


Figure 10-7: File system structure of the root device as volume

The OpenBoot PROM accesses the boot device as partition, so the missing VTOC and the wrong boot block position within the root volume are ignored. During kernel initialization, the root device is mapped by **`vxio`** as a kernel memory volume exactly on the root partition (including the VTOC), so the file system provides proper structures. But the remount of the root device as volume based on the disk group configuration stored within the private region of the OS disk during the single user mode/milestone will detect a corrupt file system: Sector 16 of **`rootvol`** does not contain the main super block!

Well, the main super block kept its correct position on the root partition. We simply need to adjust the offsets within the plex of **`rootvol`** by moving them one sector rearwards. That may be easily accomplished by the capabilities of a logical volume management with completely flexible subdisk architecture: Concatenate the main subdisk of **`rootvol`** to a subdisk just one sector in size, to the "Ghost subdisk"!

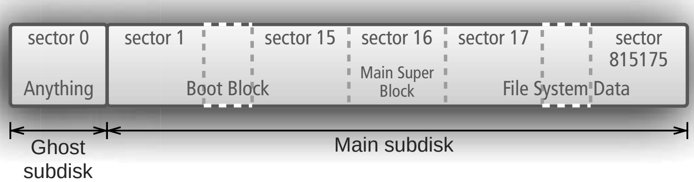


Figure 10-8: File system structure of the root device as volume with "Ghost subdisk"

Finally the main super block is located at the correct volume device offset. The Ghost subdisk is placed at the plex offset 0. But where to place the Ghost subdisk physically? A logical volume management discerns between the physical and the logical position of its building blocks. We might initialize another disk for the disk group in order to store the Ghost subdisk. What a ridiculous waste of space! A new disk for a subdisk of just 1 sector!

VxVM can do better. The private region **as configuration container** gets an offset of 1 sector compared to the beginning of the private region **as partition**. We may easily forego one sector of the private region, that only means a maximum of two configurable VxVM objects less than before (approx. 3000 objects in our example). But you cannot define a subdisk outside of the public region. No trouble whatsoever! We extend the public region partition to the boundaries of the whole disk, thus covering the private region partition as well. In order to prohibit subdisks within the configuration part of the private region, we limit the end of the public region as subdisk container to the first sector of the private region as partition.

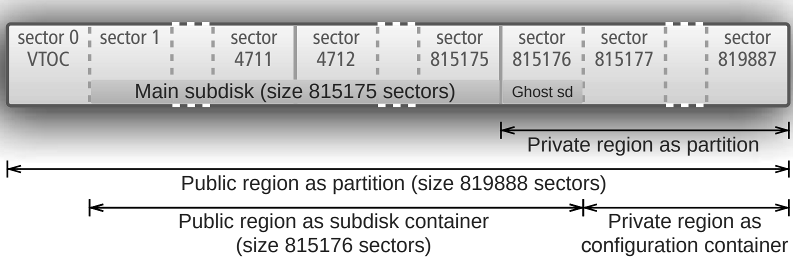


Figure 10-9: Extended public region and the position of the Ghost subdisk

The same disk configuration by command output:

```
# vxdisk list c4t8d0
[…]
```

```
public:    slice=3 offset=1 len=815176 disk_offset=0
private:   slice=4 offset=1 len=4171 disk_offset=815176
[…]
```

Given the usual drivers performing I/O (block or swap device), the Ghost subdisk will not be affected by any I/O, for they skip the first 16 sectors of the device. Therefore, the Ghost subdisk is NOT a performance drawback, although some people promoting a cheaper but less intelligent volume management try to make you believe so. It does not look really pretty, maybe. If you care for beauty, then install your Solaris system on a disk keeping the first (two) cylinders unused by OS partitions. Or you should mirror the volumes created by encapsulation (we assume **c0t0d0s2** and its disk media name **osdg01**) to a freshly initialized sliced VxVM disk (**osdg02**). That disk layout always skips the first cylinder and creates the private region and public region partitions mutually exclusive on the remaining disk (see OS mirror section above, p.324-326). Then, re-initialize and re-mirror the original OS disk by the following list of commands:

```
# vxplex -g bootdg -o rm dis $(
  vxprint -g bootdg -pne 'pl_sd.sd_dmname="osdg01"')
# vxdg -g bootdg rmdisk osdg01
# vxdisksetup -i c0t0d0 format=sliced privlen=1m
# vxdg -g bootdg adddisk osdg01=c0t0d0
# vxmirror -g bootdg osdg02 osdg01
```

10.7 Manual Encapsulation Walkthrough

10.7.1 Assumptions and Prerequisites

Two purposes are connected with the following section: to help an understanding of the basic encapsulation procedure and to show some kind of "worst case" scenario where the standard command line interfaces will fail. What does "worst case" exactly mean?

1. The **vxencap** script needs at least one (**cdsdisk** layout) or two (**sliced**) unused partitions in order to create the VxVM partition (**cdsdisk**) or the private region and public region partitions (**sliced**). But unfortunately, all partitions are already in use and cannot be foregone.
2. Any encapsulation procedure needs a small amount of unused disk space in order to create the private region out of it and to store the new VxVM objects. But there is no free space on the disk to become the private region.
3. Encapsulation enables volume sizes beyond disk limits, data redundancy, performance tuning, and the flexibility of online volume management. Our example will deal only with data mirroring, for new worries will await us (further resize and relayout operations are plain sailing and therefore not discussed).
4. For cross platform compatibility (and another reason to be explained later), we wish to convert the file systems of type **ufs** on our partitions into **vxfs**.
5. As long as the first disk cylinder is in use by a partition holding application data, any encapsulation procedure will protect the VTOC by replacing sector 0 by the Ghost subdisk to recreate the proper device offsets (see previous section). The private region cannot be placed at the fixed disk offset of 256 sectors of the **cdsdisk** layout. Thus, a **sliced** layout must be created without the capabilities of "Cross Platform data Sharing". Both the Ghost subdisk colliding with the CDS disk group subdisk alignment of 8 kB and the wrong position of the private region hinder us to easily activate CDS features.
6. Veritas software is designed to aid high availability. But two file system based OS limitations force a temporary application interruption: It is impossible to replace the partition drivers by their corresponding volume drivers and to modify the file system layout from **ufs** to **vxfs**, while the application is running on the device. All we can do is to try to stop and restart the applications as quickly as possible. That calls suspiciously for a script running the commands in uninterrupted sequence.

Just look at the following command outputs to determine the current state of the partition based "applications" (the file systems each holding a large file are mounted):

```
# prtvtoc /dev/rdsk/c4t1d0s2
[…]
*    4712 sectors/cylinder
*    7508 cylinders
*    7506 accessible cylinders
```

```
[…]
*                          First     Sector    Last
* Partition  Tag  Flags    Sector     Count    Sector  Mount Directory
       0      0    00          0   5051264   5051263   /mnt0
       1      0    00    5051264   5051264  10102527   /mnt1
       2      5    01          0  35368272  35368271
       3      0    00   10102528   5051264  15153791   /mnt3
       4      0    00   15153792   5051264  20205055   /mnt4
       5      0    00   20205056   5051264  25256319   /mnt5
       6      0    00   25256320   5051264  30307583   /mnt6
       7      0    00   30307584   5060688  35368271   /mnt7
# df -k /mnt?
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c4t1d0s0    2474263 2098193  326585    87%    /mnt0
/dev/dsk/c4t1d0s1    2474263 2098193  326585    87%    /mnt1
/dev/dsk/c4t1d0s3    2474263 2098193  326585    87%    /mnt3
/dev/dsk/c4t1d0s4    2474263 2098193  326585    87%    /mnt4
/dev/dsk/c4t1d0s5    2474263 2098193  326585    87%    /mnt5
/dev/dsk/c4t1d0s6    2474263 2098193  326585    87%    /mnt6
/dev/dsk/c4t1d0s7    2478975 2098193  331203    87%    /mnt7
# for i in 0 1 3 4 5 6 7; do fstyp /dev/rdsk/c4t1d0s$i; done | uniq
ufs
```

10.7.2 Basic Considerations

Our desperate try to use the standard **vxencap** script and following the clean-up:

```
# vxencap -g edg -c edg01=c4t1d0
  VxVM vxencap ERROR V-5-2-213
It is not possible to encapsulate c4t1d0, for the following reason:
     <VxVM vxslicer ERROR V-5-1-754 Not enough free partitions.>
# rm -r /dev/vx/reconfig.d/disk.d/c4t1d0
```

Well, that did not work! Some thoughts and remarks on the list of difficulties will clear the way we need to walk on. Since our "worst case" assumption does not allow for a removal of any application data, we may shrink a device only at its end to free a small amount of space for the private region. This implies that we cannot shrink the first partition at its beginning in order to enable a **cdsdisk** layout, and that we must convert **ufs**, which cannot be shrunken, to **vxfs** at least on one partition at the very beginning. We choose slice 5 (for an imaginary reason) to be stopped early in order to convert the file system to **vxfs** and to shrink it by one cylinder.

Furthermore, the inevitable **sliced** layout requires two unused partition numbers. Partition 5, already suffering early application shutdown, will serve as the private region (even though located in the middle of the disk), partition 7 (once again for an imaginary reason) will define the public region and must be freed from application access as well.

10.7.3 Storing the Disk Layout

Since our procedure removes two partitions from the VTOC before the correspondent subdisks are created, storing the offset and the length of the partitions is required (except for the backup slice, of course). By the way, we will determine the size of a disk cylinder and of the whole disk in order to create the private and the public region in the proper size. Finally, we will convert the partition offsets into subdisk offsets (decremented by 1 due to the VTOC protection) and, for the first slice (slice 0, VTOC protection) and slice 5 (one cylinder split to become the private region), the partition lengths into subdisk lengths. Here is our first code fragment:

```
Disk=c4t1d0
File=/tmp/$Disk.$$

prtvtoc /dev/rdsk/${Disk}s2 |
nawk '
    $3=="sectors/cylinder" {print "SecPerCyl",$2; SecPerCyl=$2}
    $1~/^[0134567]$/ {
        if ($1==5) {print "OffsetPrivReg",$6-SecPerCyl+1; $5-=SecPerCyl}
        if ($4==0) {print "FirstPart",$1;$5--} else {$4--}
        print $1,$4,$5
    }
    $1==2 {print "SecOfDisk",$5}
' > $File
```

10.7.4 Defining Private and Public Region

Nothing happened to the disk and to the applications, because our script collected data in a non-intrusive manner. The next step requires application stop for partition 5: The file system needs to be unmounted in order to convert **ufs** to **vxfs** in order to shrink the file system by the size of one cylinder. Conversion of **ufs** to **vxfs** just inactivates the former **ufs** metadata by addressing the blocks holding file content by **vxfs** metadata. So, a file system check is required to free the device from invalid **ufs** structures (a full check without log replay, for there are still no valid log data). Finally, the current **vxfs** file system is shrunken by the size of one cylinder which requires a temporary mount.

```
NewSize=$(nawk '$1==5 {print $3}' $File)
umount /mnt5
vxfsconvert -y /dev/rdsk/${Disk}s5
fsck -F vxfs -o full,nolog -y /dev/rdsk/${Disk}s5
mount -F vxfs /dev/dsk/${Disk}s5 /mnt5
fsadm -F vxfs -b $NewSize -r /dev/rdsk/${Disk}s5 /mnt5
umount /mnt5
```

Defining subdisks, plexes, and volumes requires a disk initialized for VxVM (private and public region) as a disk group member. The previous step created disk space one cylinder in size and not used by the file systems. In order to initialize the disk for VxVM, we still need two unused partition numbers: The file system of partition 5 is unmounted, and partitions 5 and 7 are removed from the VTOC.

```
umount /mnt7
fmthard -d 5:0:0:0:0 /dev/rdsk/${Disk}s2
fmthard -d 7:0:0:0:0 /dev/rdsk/${Disk}s2
```

We redefine partitions 5 and 7 to become private and public region. The private region, formerly the last cylinder of slice 5, is located in the middle of the disk. Therefore, slice 7 as the public region must cover the whole disk in order to cover all application partitions. As we already know, VxVM does not worry about a private region being part of the public region. Disk initialization for VxVM is completed by writing a basic structure into the private region.

```
SecPerCyl=$(nawk '$1=="SecPerCyl" {print $2}' $File)
SecOfDisk=$(nawk '$1=="SecOfDisk" {print $2}' $File)
OffsetPrivReg=$(nawk '$1=="OffsetPrivReg" {print $2}' $File)
fmthard -d 5:15:01:$OffsetPrivReg:$SecPerCyl /dev/rdsk/${Disk}s2
fmthard -d 7:14:01:0:$SecOfDisk /dev/rdsk/${Disk}s2
vxdisk -f init $Disk format=sliced privlen=1m
```

10.7.5 Creating Subdisks, Plexes, and Volumes

We already overcame some difficult obstacles. Remember that still five of seven applications are running without interruption. The next steps define the typical VxVM objects to create the volume drivers on the disk spaces accessed by the partitions. But, of course, the disk must become a disk group member first. The following code part defines the default disk group for the script, initializes the disk group and creates the Ghost subdisk located at the first sector of the private region. The variable **FirstPart** stores the number of the partition starting at cylinder 0, because the Ghost subdisk is needed for offset alignment within the corresponding plex.

```
export VXVM_DEFAULTDG=edg
vxdg init edg edg01=$Disk cds=off
vxdg set align=1
# Ghost subdisk
vxmake sd edg01-B0 disk=edg01 offset=$((OffsetPrivReg-1)) len=1
FirstPart=$(nawk '$1=="FirstPart" {print $2}' $File)
```

Within a loop over all application partition numbers, the subdisks are placed over the partitions, and the plexes and finally the volumes are built out of them. The volumes are started, the still mounted file systems unmounted, the underlying partitions removed, the file systems converted to **vxfs** and checked, and finally remounted as **vxfs** based on the volume drivers.

```
for i in 0 1 3 4 5 6 7; do
    nawk '$1=='$i' {print $2,$3}' $File | read Offset Len
    vxmake sd edg01-0$i disk=edg01 offset=$Offset len=$Len
    if ((i==FirstPart)); then
        vxmake plex vol$i-01 sd=edg01-B0,edg01-0$i
    else
        vxmake plex vol$i-01 sd=edg01-0$i
    fi
    vxmake vol vol$i plex=vol$i-01 usetype=fsgen
    vxvol start vol$i
    if ((i!=5 && i!=7)); then
        umount /mnt$i
        fmthard -d $i:0:0:0:0 /dev/rdsk/${Disk}s2
    fi
    if ((i!=5)); then
        vxfsconvert -y /dev/vx/rdsk/edg/vol$i
        fsck -F vxfs -o full,nolog -y /dev/vx/rdsk/edg/vol$i
    fi
    mount -F vxfs /dev/vx/dsk/edg/vol$i /mnt$i
done
```

Wow, the worst part has completed! Our script has (successfully, we hope) executed the time-critical parts of the conversion. All applications are online once again and do not need to be stopped for the following volume and file system management tasks. Lean back for a few seconds and breathe deeply! Then, have a look at the complete script once again with some comments, output redirections, and further output displaying the time required to execute the steps (41 sec. totally, most applications stopped just for a few seconds).

```
# cat ./encap_advanced
#!/bin/ksh

Disk=c4t1d0
File=/tmp/$Disk.$$

# Store disk layout
echo $(date +%H:%M:%S): Storing disk layout
prtvtoc /dev/rdsk/${Disk}s2 |
nawk '
    $3=="sectors/cylinder" {print "SecPerCyl",$2; SecPerCyl=$2}
    $1~/^[0134567]$/ {
        if ($1==5) {print "OffsetPrivReg",$6-SecPerCyl+1; $5-=SecPerCyl}
        if ($4==0) {print "FirstPart",$1;$5--} else {$4--}
        print $1,$4,$5
    }
    $1==2 {print "SecOfDisk",$5}
' > $File
```

```
# Convert /mnt5 to VxFS, then shrink it to create space for private region
echo $(date +%H:%M:%S): Convert /mnt5 to VxFS and shrink
NewSize=$(nawk '$1==5 {print $3}' $File)
umount /mnt5
vxfsconvert -y /dev/rdsk/${Disk}s5 >/dev/null 2>&1
fsck -F vxfs -o full,nolog -y /dev/rdsk/${Disk}s5 >/dev/null
mount -F vxfs /dev/dsk/${Disk}s5 /mnt5
fsadm -F vxfs -b $NewSize -r /dev/rdsk/${Disk}s5 /mnt5 >/dev/null
umount /mnt5

# Delete partitions 5 and 7
echo $(date +%H:%M:%S): Delete partitions 5 and 7
umount /mnt7
fmthard -d 5:0:0:0:0 /dev/rdsk/${Disk}s2
fmthard -d 7:0:0:0:0 /dev/rdsk/${Disk}s2

# Initialize disk as VxVM disk
echo $(date +%H:%M:%S): Initialize disk as VxVM disk
SecPerCyl=$(nawk '$1=="SecPerCyl" {print $2}' $File)
SecOfDisk=$(nawk '$1=="SecOfDisk" {print $2}' $File)
OffsetPrivReg=$(nawk '$1=="OffsetPrivReg" {print $2}' $File)
fmthard -d 5:15:01:$OffsetPrivReg:$SecPerCyl /dev/rdsk/${Disk}s2
fmthard -d 7:14:01:0:$SecOfDisk /dev/rdsk/${Disk}s2
vxdisk -f init $Disk format=sliced privlen=1m

# Create disk group, build subdisks, plexes, volumes
echo $(date +%H:%M:%S): Create disk group
export VXVM_DEFAULTDG=edg
vxdg init edg edg01=$Disk cds=off
vxdg set align=1
# Ghost subdisk
vxmake sd edg01-B0 disk=edg01 offset=$((OffsetPrivReg-1)) len=1
FirstPart=$(nawk '$1=="FirstPart" {print $2}' $File)

for i in 0 1 3 4 5 6 7; do
    echo $(date +%H:%M:%S): Create volume vol$i
    nawk '$1=='$i' {print $2,$3}' $File | read Offset Len
    vxmake sd edg01-0$i disk=edg01 offset=$Offset len=$Len
    if ((i==FirstPart)); then
        vxmake plex vol$i-01 sd=edg01-B0,edg01-0$i
    else
        vxmake plex vol$i-01 sd=edg01-0$i
    fi
    vxmake vol vol$i plex=vol$i-01 usetype=fsgen
    vxvol start vol$i
    if ((i!=5 && i!=7)); then
```

```
        umount /mnt$i
        fmthard -d $i:0:0:0:0 /dev/rdsk/${Disk}s2
    fi
    if ((i!=5)); then
        echo $(date +%H:%M:%S): Convert vol$i to VxFS
        vxfsconvert -y /dev/vx/rdsk/edg/vol$i >/dev/null 2>&1
        fsck -F vxfs -o full,nolog -y /dev/vx/rdsk/edg/vol$i >/dev/null
    fi
    echo $(date +%H:%M:%S): Mount vol$i to /mnt$i
    mount -F vxfs /dev/vx/dsk/edg/vol$i /mnt$i
done

# ./encap_advanced
16:05:11: Storing disk layout
16:05:11: Convert /mnt5 to VxFS and shrink
16:05:16: Delete partitions 5 and 7
16:05:16: Initialize disk as VxVM disk
16:05:18: Create disk group
16:05:20: Create volume vol0
16:05:22: Convert vol0 to VxFS
16:05:25: Mount vol0 to /mnt0
16:05:25: Create volume vol1
16:05:26: Convert vol1 to VxFS
16:05:34: Mount vol1 to /mnt1
[…]
16:05:48: Create volume vol7
16:05:49: Convert vol7 to VxFS
16:05:52: Mount vol7 to /mnt7
```

We check the results. Note that the private region as configuration container indeed covers just 1 MB of the private region as partition, as instructed by our initialization parameters.

```
# vxdisk list c4t1d0
[…]
info:      format=sliced,privoffset=1,pubslice=7,privslice=5
[…]
public:    slice=7 offset=1 len=35368271 disk_offset=0
private:   slice=5 offset=1 len=2048 disk_offset=25251608
[…]
# vxprint -rtg edg
[…]
dm edg01        c4t1d0s2     auto     2048     35368271 -

v  vol0         -            ENABLED  ACTIVE   5051264  ROUND     -        fsgen
pl vol0-01      vol0         ENABLED  ACTIVE   5051264  CONCAT    -        RW
sd edg01-B0     vol0-01      edg01    25251607 1        0         c4t1d0   ENA
```

```
sd edg01-00     vol0-01      edg01    0        5051263  1        c4t1d0   ENA

v  vol1         -            ENABLED  ACTIVE   5051264  ROUND    -        fsgen
pl vol1-01      vol1         ENABLED  ACTIVE   5051264  CONCAT   -        RW
sd edg01-01     vol1-01      edg01    5051263  5051264  0        c4t1d0   ENA
[…]
v  vol7         -            ENABLED  ACTIVE   5060688  ROUND    -        fsgen
pl vol7-01      vol7         ENABLED  ACTIVE   5060688  CONCAT   -        RW
sd edg01-07     vol7-01      edg01    30307583 5060688  0        c4t1d0   ENA
# df -k /mnt?
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/edg/vol0 2525632 2098928  400042    84%    /mnt0
/dev/vx/dsk/edg/vol1 2525632 2098928  400042    84%    /mnt1
/dev/vx/dsk/edg/vol3 2525632 2098928  400042    84%    /mnt3
/dev/vx/dsk/edg/vol4 2525632 2098928  400042    84%    /mnt4
/dev/vx/dsk/edg/vol5 2523276 2098928  397833    85%    /mnt5
/dev/vx/dsk/edg/vol6 2525632 2098928  400042    84%    /mnt6
/dev/vx/dsk/edg/vol7 2530344 2098928  404460    84%    /mnt7
# for i in 0 1 3 4 5 6 7; do fstyp /dev/vx/rdsk/edg/vol$i; done | uniq
vxfs
```

10.7.6 Mirroring and Preparing for CDS

Data redundancy still lacks. While planning volume mirroring, we keep in mind that we want to migrate to a CDS disk group. Therefore, we initialize the second disk of our disk group in the **cdsdisk** layout. As long as the **cds** disk group attribute is cleared, we may mix both disk layouts within a disk group. Our slight hope to easily convert disks and disk group by the standard **vxcdsconvert** command, so that CDS capabilities are activated, is (we might have expected it) immediately dashed.

```
# vxdisksetup -i c4t2d0 privlen=1m
# vxdg -g edg adddisk edg02=c4t2d0
# vxdisk -g edg list
DEVICE       TYPE            DISK         GROUP        STATUS
c4t1d0s2     auto:sliced     edg01        edg          online
c4t2d0s2     auto:cdsdisk    edg02        edg          online
# vxcdsconvert -g edg -o novolstop group evac_subdisks_ok=yes privlen=1m
VxVM vxcdsconvert ERROR V-5-2-2763 c4t1d0s2: Public and private regions overlap
VxVM vxcdsconvert ERROR V-5-2-3120 Conversion process aborted
```

Well, no flight in a luxurious airplane, instead a long exhausting way on foot? No, we still may make use of efficient VxVM scripts. But in order to achieve the desired result by not too long a list of commands, we must use our brains a little bit. A simple volume mirroring executed by **vxmirror** would produce a result we could not go on with (we assume the same disk and cylinder size for the mirror disk). Why? A requirement for the conversion to a CDS disk group is still not met: the subdisk alignment to 8 kB blocks. Neither all volume sizes nor all subdisk offsets nor all subdisk lengths on the mirror disk are or would be integer multiples of 8 kB:

```
# vxprint -g edg -vF '%name %len'|nawk '{printf "%s %.2f\n",$1,$2/16}'
vol0 315704.00
vol1 315704.00
vol3 315704.00
vol4 315704.00
vol5 315409.50
vol6 315704.00
vol7 316293.00
# vxprint -g edg -se 'sd_dmname="edg02"' -F '%name %offset %len' |
  nawk '{printf "%s %10.2f %10.2f\n",$1,$2/16,$3/16}'
edg02-01     150.50  315704.00
edg02-02  315854.50  315704.00
edg02-03  631558.50  315704.00
edg02-04  947262.50  315704.00
edg02-05 1262966.50  315409.50
edg02-06 1578376.00  315704.00
edg02-07 1894080.00  316293.00
```

Since the original disk has **sliced** layout and must be remirrored, we may ignore the subdisk offsets unsuitable to a CDS disk group. But the wrong volume length bothers us. What is the next integer multiple of the current volume length? How many sectors are missing? Nevertheless, adding the difference of the desired and the current volume size to the volume length in order to create an integer multiple of 8 kB fails:

```
# vxprint -g edg -F %len vol5
5046552
# echo $(((5046552/16+1)*16))
5046560
# echo $((5046560-5046552))
8
# vxresize -g edg -F vxfs -x vol5 +8 edg01
VxVM vxassist ERROR V-5-1-436 Cannot allocate space to grow volume to 5046560
blocks
VxVM vxresize ERROR V-5-1-4703 Problem running vxassist command for volume vol5,
in diskgroup edg
```

Is it indeed impossible to allocate just eight blocks on the original disk? We built the private region out of a partition one cylinder in size, i.e. 4712 sectors given our example. But we fixed the size of the private region as configuration container by 1 MB (= 2048 sectors), therefore 4712 - 2048 = 2664 sectors should be available for new subdisks (1 sector already in use by the Ghost subdisk). We remember that in most cases the "Cannot allocate space" error message is misleading: There is enough space, but layout restrictions would be violated. The current layout restriction is the default **diskalign** attribute of **vxassist** enforcing subdisk creation at cylinder boundaries. Once recognized as the source of our troubles, we simply turn it of and retry the resize operation:

```
# vxassist help showattrs
#Attributes:
 layout=nomirror,nostripe,nomirror-stripe,nostripe-mirror,nostripe-mirror-col,nostripe-mirror-sd,
noconcat-mirror,nomirror-concat,span,nocontig,raid5log,noregionlog,diskalign,nostorage
[…]
# echo layout=nodiskalign > /etc/default/vxassist
# vxassist help showattrs
#Attributes:
 layout=nomirror,nostripe,nomirror-stripe,nostripe-mirror,nostripe-mirror-col,nostripe-mirror-sd,
noconcat-mirror,nomirror-concat,span,nocontig,raid5log,noregionlog,nodiskalign,nostorage
[…]
# vxresize -g edg -F vxfs -x vol5 +8 edg01
# vxmirror -g edg edg01
! vxassist -g edg mirror vol0
! vxassist -g edg mirror vol1
! vxassist -g edg mirror vol3
! vxassist -g edg mirror vol4
! vxassist -g edg mirror vol5
! vxassist -g edg mirror vol6
! vxassist -g edg mirror vol7
```

The **nodiskalign** attribute ensures that the mirror procedure of **vxmirror** (based on **vxassist mirror** commands) will not fit the subdisks at cylinder boundaries anymore, will place the first subdisk immediately after the private region of the mirror disk (having **cdsdisk** layout), and therefore will not run out of space. The mirror disk already fulfills all criteria to become part of a CDS disk group.

The original disk must be re-initialized as a CDS suitable disk and completely re-mirrored by the same procedure.

```
# vxplex -g edg -o rm dis $(vxprint -g edg -pne 'pl_sd.sd_dmname="edg01"')
# vxdg -g edg rmdisk edg01
# vxdisksetup -i c4t1d0 privlen=1m
# vxdg -g edg adddisk edg01=c4t1d0
# vxdisk -g edg list
DEVICE       TYPE            DISK         GROUP        STATUS
c4t1d0s2     auto:cdsdisk    edg01        edg          online
c4t2d0s2     auto:cdsdisk    edg02        edg          online
# vxprint -dtg edg
DM NAME         DEVICE       TYPE     PRIVLEN  PUBLEN   STATE

dm edg01        c4t1d0s2     auto     2048     35365968 -
```

```
dm edg02        c4t2d0s2     auto     2048     35365968 -
# vxmirror -g edg edg02
! vxassist -g edg mirror vol0
! vxassist -g edg mirror vol1
! vxassist -g edg mirror vol3
! vxassist -g edg mirror vol4
! vxassist -g edg mirror vol5
! vxassist -g edg mirror vol6
! vxassist -g edg mirror vol7
```

10.7.7 Converting to CDS

Now the final steps! VxFS is already a cross platform compatible file system (except for Windows), but the disk group is still not completely prepared for CDS. The subdisk alignment needs to be changed to 8 kB, and the **cds** attribute of the disk group must be set. The former **vxassist** defaults are reset, a file system defragmentation may be useful, and appropriate **/etc/vfstab** entries should contain the volume drivers and **vxfs**.

```
# vxprint -Gg edg -F '%align %cds'
1 off
# vxdg -g edg set align=16
# vxdg -g edg set cds=on
# vxprint -Gg edg -F '%align %cds'
16 on
# rm /etc/default/vxassist
# for i in 0 1 3 4 5 6 7; do fsadm -F vxfs -de /mnt$i; done
# vi /etc/vfstab
[…]
/dev/vx/dsk/edg/vol0  /dev/vx/rdsk/edg/vol0  /mnt0  vxfs  2  yes  -
[…]
```