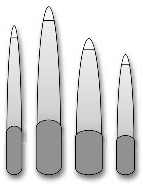
Another Kind of File System ;)

CHAPTER 12: FILE SYSTEMS

by Volker Herminghaus

Since you are reading a book about volume management for highly available UNIX systems in data centers we would like to assume that you have some basic understanding of what a file system is used for. We will spare both your time and ours describing the basic goals of a file system. But there are some essential differences between the various types of file systems that are very useful to point out. So let us begin with a short overview of the various file system types that have developed over time, and how their strengths increased. Yes, increased, not changed. As a general rule, file systems have indeed gotten a lot better over time without losing any important features relative to older implementations. Different from other aspects of computer technology one can safely say that so far, file system development has been a one-way road towards improvement, not heavily traded-off bloatware development as so many other parts of operating systems. At least if we ignore the non-sequential access methods of some older file systems. But those can be easily emulated on a higher level, and their omission from file system code probably improved performance rather than diminishing it.

12.1 Block Based File Systems

Most common file systems are based on allocating single individual blocks from the underlying partition or volume to store user data as well as file system metadata. There is always some higher-level data (the metadata mentioned above) that bundles these blocks together to the notion of a sequential file and stores meta information about the file and the total file system. This metadata has been stored in many different forms, including very arcane ones, over the course of time. As one of the most bizarre ones known to the current instance of humanity we would like to point out the "file system" of the Commodore 64 floppy disk (really!).

12.1.1 Just for Fun: Commodore 64's Rudimentary File Access

It was organized into a single directory (of limited length). The directory was stored in a fixed location. Each directory entry had a name and a starting block for the file. If a file was deleted, the first character of the name was overwritten by a special character to indicate the file's "deleted" state. There was no centralized information about the blocks that a file would consist of. Instead, the first block was pointed to by the directory entry, and the consecutive blocks were pointed to by the last byte of the previous block. In effect, this resulted in a linked list of blocks that could only be read sequentially. Even when a process (or should I say "**the program**", as this was a single-tasking system) wanted to access the end of the file the system still had to read through all of the file's blocks in order to find its way to the end – after all, it had to read the whole linked list!

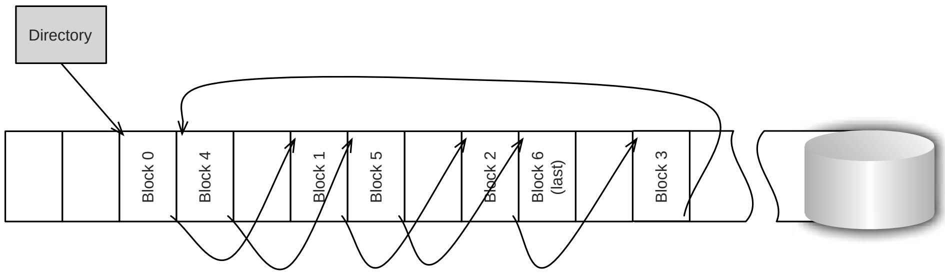

Figure 12-1: The extremely basic Commodore 64 file system mixed meta data (the linked list of block numbers) with user data. Seeking required reading all blocks up to the end point. This was because mass storage can only be read in blocks rather than directly addressed, and in order to retrieve the pointer to the next block, the whole data block had to be transferred.

12.1.2 FAT – Not a Big Improvement

A little better, although not much, was – and still is – the file system used by MS-DOS and, later, Windows "operating systems". It is called FAT (for File Allocation Table) file system and has a central root directory that allows for subdirectories. Files are still deleted by invalidating the first character of the file name. And the starting block is still stored in the file's directory entry. But one of the very few things that are better in this file system than in the Commodore 64's is that there now **is** a central location that holds a complete linked list of blocks for a file (the file allocation table). The FAT data structure works basically as an array of 12, 16, or 32-bit values in which each index is identical to the corresponding data block number in the data area of the file system. The first array element in the FAT corresponds to the first data block in the data portion of the file system, the one-hundredth element corresponds to the one-hundredth data block, etc. The values in the array are either NULL (in which case no further block follows) or the number of the next block in the file. In other words, there is still a linked list of blocks per file, but the list has been moved outside of the data portion of the file system, and into a separate data structure, which can be read independently without reading all of the file. Seeking in a file is therefore not awfully slow but merely pretty slow. Of course, the "great visionary" Bill Gates has provided enough lock-ins to make the FAT file system woefully inadequate for serious computing today and yesterday. For instance, the number of addressable blocks is limited and therefore the only way to scale the file system size up is to increase the size of the allocated units (called "clusters"). This is very wasteful because even for tiny files the file system must allocate at least one allocation unit. While most serious file systems have minimum allocation units of 1 KB, FAT file systems of a total size of merely 2 GB already require at least 32 KB allocation units. Apart from this, traversing linked lists really is not fast unless you are handling very few objects, i.e. blocks. This is not the case today, and neither FAT's feature set nor its performance can in any way compete with even the most primitive versions of e.g. UNIX file systems.

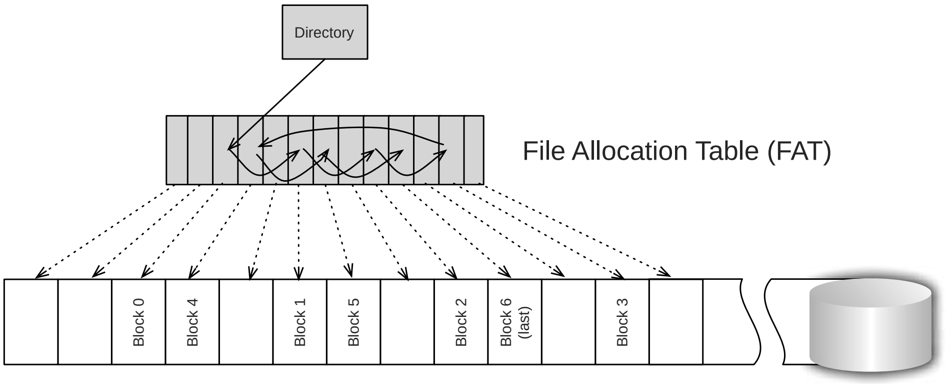

Figure 12-2: FAT file system moves the linkage information into a separate data structure, away from the user data, so it can be processed more quickly – somewhat.

Apart from its wastefulness FAT is extremely limited when it comes to file naming. FAT file names are inherently limited to an 8.3 pattern, and the extension to that naming pattern is incompatible with many file sharing protocols or shared media, and implemented in a very inefficient, hackish way. Ownership, group and other permissions are not supported, access times are not recorded, and support for multiple names referencing the same data is limited at best. In short, it is a very ancient, crufty file system that has survived solely because the majority of its users don't know better and are used to malfunctioning operating systems, so basically nobody ever bothered to fix it.

12.1.3 UFS – Finally Something Decent

What is known these days as UFS, or UNIX File System, is actually not the original AT&T System V UFS but the much more modern Berkeley FFS (Fast File System). The original UFS used a "**super block**" at a fixed location for finding the rest of the metadata and for maintaining file system global values such as clean/dirty state, size and degree to which the file system is full. It used a linked list for managing free blocks, and it invented the notion of an "inode", or **information node**, that holds all metadata pertaining to the file. The inode contained user and group ID of the file's owner, the numbers of the first ten blocks of the file, read and access time stamps as well as a time stamp for the last change to the inode, permission information, size, and a whole lot more. If the file was longer than the portion that can be addressed by the block numbers in the inode, then the next block number in the inode would not be the eleventh block of the file, but it would contain a so-called "indirect block", i.e. a block full of block numbers of file data blocks. Thus a file could be enlarged by a lot of blocks with just one extra block of metadata (the block containing the additional data block numbers, the indirect block). If that did still not suffice, then the next block number in the inode would be a double-indirect block containing the addresses of indirect blocks which in turn contained data block numbers. Number thirteen would be a triple-indirect block, and by then all conceivable storage devices of those times would have long been exhausted. Depending on the blocksize, a file could be as large as 2GB (that was at the time of 5 MB disk drives!). Remember: when looking for vision, first talk to guys like Brian Kernighan, Rob Pike & Dennis Ritchie, the inventors of the original UNIX system and the C language and much more. And always talk to Bill Gates last – just for having a good laugh at the end!

The Berkeley FFS, which has been commonly adopted by the UNIX crowd as the standard file system, has replaced the linked list of the original UFS with a bitmap representing the free blocks, along with some efficient algorithms that work on those bitmaps trying to find contiguous stretches of storage into which to store the data. It has also introduces the concept of "cylinder groups", effectively distributing file data and I/O across all of the volume address space. Basically, a cylinder group is a number of contiguous disk cylinders consisting of a superblock copy, block allocation bitmap, inode table, and data blocks. The metadata is centered in the data blocks to allow for better proximity between metadata and user data.

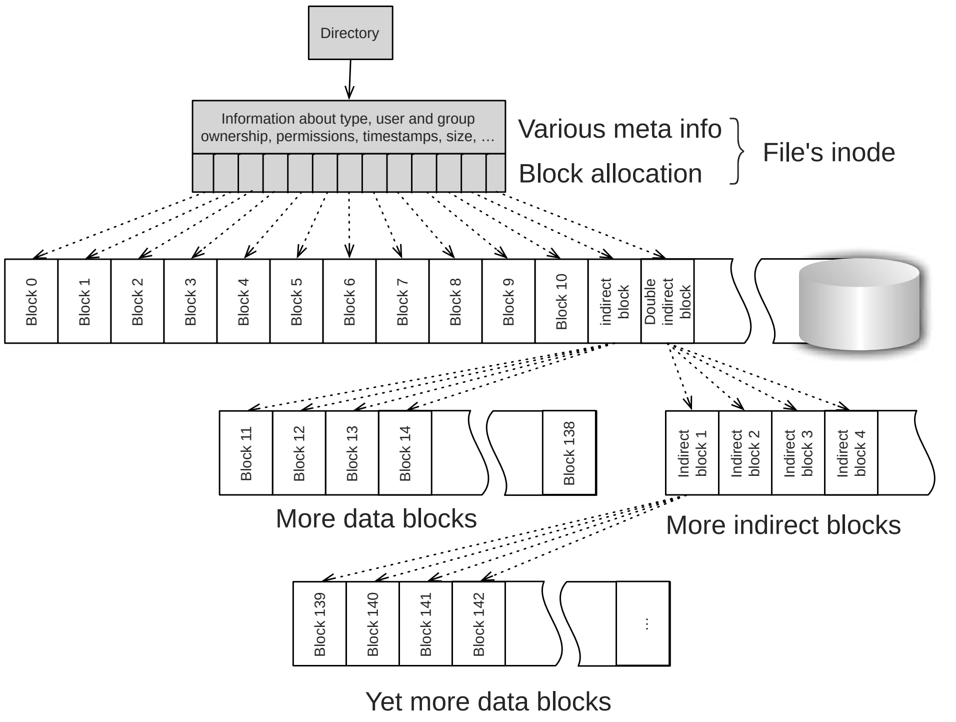

Figure 12-3: UFS uses inodes to concentrate meta information as well as block allocation information into one place. Linked lists pointing to data blocks are superseded by direct, indirect, double indirect and triple indirect block lists.

Allocation of space for new files is done by first selecting a cylinder group (of which there are usually many – **newfs** tells you the number of cylinder groups and the location of the superblock copies when you create a UFS file system), and then allocating blocks in a preferably contiguous fashion from that cylinder group. Multi-user I/O will therefore typically be distributed across several cylinder groups; a similar effect to striping on a volume level. This makes sure that the disk or volume is used in a balanced way, especially with respect to the usage of the (fixed) inodes for the data around the inode table.

Berkeley FFS is a pretty good file system, but it is still far from a really good one. For one thing, block-based addressing requires a relatively large amount of metadata – one block number per block. The fixed locations and sizes of UFS's metadata make it wasteful and inflexible in extreme cases. File systems with very few very large files are a bad case because a large number of inodes is reserved in vain. Likewise, the opposite case is bad, too. File systems with lots of small files will eventually run out of inodes even if there is plenty of space still left in the data portion of the file system. And lastly, advanced features like point-in-time-copies, shrinking of file systems, or reorganizing the files ("defragmenting") is not efficiently possible.

All of those advanced features, and many more advantages, can be done easily when one switches from block-addressing to extent-addressing. This is what the developers of VxFS have done. It is by no means easy to write an extent-based file system, which is why there are so few implementations. But when one has managed to get beyond the complexity of the problem, there is a whole world of new capabilities that await the user of that file system (if you need to know what extents are, refer to their explanation in the Basics chapter, page 2). Welcome to VxFS!

12.2 Extent Based File Systems

12.2.1 VxFS

Veritas File System, or VxFS in short, is a file system that uses extent-based rather than block-based allocation. What is an extent in this context? An extent in the context of VxFS is a contiguous stretch of any "power of 2" number of blocks/sectors/KBs: 1, 2, 4, 8, 16, 32, 64, 128, ... up to multi-gigabyte length extents of which only a few will cover almost all of even a very large volume. Now how is this extent-based space allocated? As we should be expecting from the guys at Veritas by now, it is allocated in a very clever way: The first extent that is allocated to as file is just large enough to hold the initial batch of data, let's say 64 KB, i.e. a 128-block extent. This extent (which, as usual, consists of a starting block number and a length) is written into the inode just like UFS writes the first allocated block into the first direct block pointer in the inode. But while UFS has used up one slot for a single 8 KB block, VxFS has used it for 64 KB - a fourth of the metadata (an extent consists of two values – starting block and length – while a block number only consists of one). Now when the file is grown beyond the 64 KB previously allocated, the next extent is allocated. This extent will not be 64 KB, but rather it will be twice that amount: 128 KB, or 256 blocks. The rationale behind this is that when a file is growing it may grow very large. We do not know how large, but the more it grows, the higher the probability that it will grow even larger. So every time the file size exceeds its allocation, the file size basically doubles because the size of the new extent is equal to twice the current size of the file (minus the initial extent size, to be precise). And with every new extent the degree of contiguousness increases, and the amount of metadata per block rapidly decreases.

While this may sound extremely wasteful, it is really not wasteful at all. Just wait for another one or two pages...

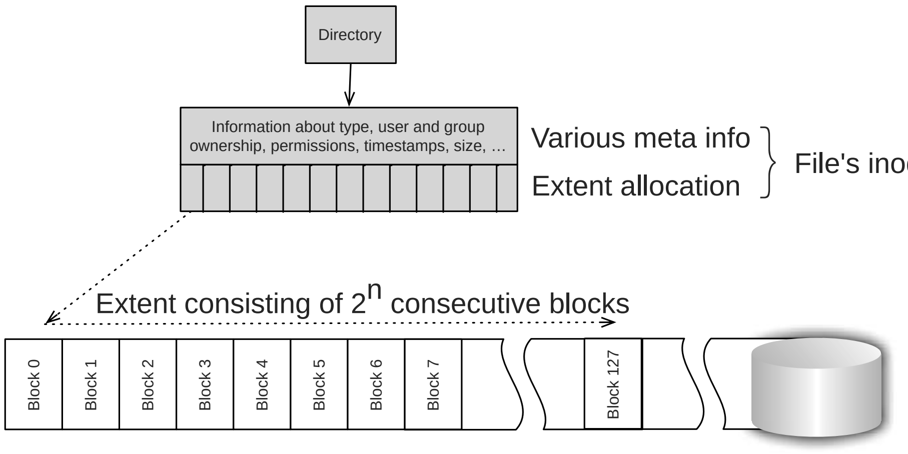
Directory
Information about type, user and group ownership, permissions, timestamps, size, ...
Various meta info
Extent allocation
File's inode
Extent consisting of 2^n consecutive blocks
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
Block 127

Figure 12-4: VxFS also uses inodes, but instead of pointing to individual blocks, it points to extents, which can be multi-megabytes in length. indirect and double indirect blocks are used, too. but are only seldom needed, because most file use far less than the ten extents that can be stored in the VxFS inode directly (VxFS allocation vastly prefers contiguousness over scattered allocation). An extent is a 32-bit block number plus a 16-bit block count, with the blocks in a file system being tunable to between 1 KB and 8 KB in length.

How Extents are Found

Extents are held in a free extent list, which is sorted by length and which can therefore be very rapidly searched and manipulated. In fact, you can look at the cree extent list of any VxFS file system by calling the file system administration tool, **fsadm**, with the appropriate options. The option "-E" lists extent information. Here is an example of what a freshly created and slightly filled 6 GB volume lists as the extent report:

```
# df -h -F vxfs
Filesystem            size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/avol   6.0G   106M   5.5G     2%    /vxfs_mnt
```

Note that the net capacity of the volume is equal to the volume size: the full 6 GB are usable! In the case of UFS, a lot of the volume's capacity is filled with fixed-location metadata like inode tables, bitmaps, super block backups, and other cylinder group info.

```
# fsadm -E /vxfs_mnt
Extent Fragmentation Report
      Total    Average      Average     Total
      Files    File Blks    # Extents   Free Blks
      503        177          1     6183245
  blocks used for indirects: 0
  % Free blocks in extents smaller than 64 blks: 0.01
  % Free blocks in extents smaller than  8 blks: 0.00
  % blks allocated to extents 64 blks or larger: 91.01
  Free Extents By Size
         1:          21          2:          28           4:          16
         8:          26         16:           3          32:           0
        64:           1        128:           1         256:           1
       512:           1       1024:           1        2048:           0
      4096:           1       8192:           0       16384:           1
     32768:           0      65536:           0      131072:           1
    262144:           1     524288:           1     1048576:           1
   2097152:           2    4194304:           0     8388608:           0
  16777216:           0   33554432:           0    67108864:           0
 134217728:           0  268435456:           0   536870912:           0
1073741824:           0 2147483648:           0
```

The pyramid-shaped columns of number above are actually the whole free space information for the file system: you see that there are 21 free extents of size 1, 28 of size 2, 16 of size 4, and so on, until finally most of the empty volume space is contained in 2 free extents of size **2097152**. The maximum extent size is 2 TB!

Efficiency of VxFS Allocation

You may wonder what happens in the following scenario: A file keeps growing and therefore larger and larger extents are being appended to it. But then, just a small portion is written into the most recently allocated (and therefor largest) extent, and then the file is closed and stops growing. What happens to the remaining, empty space in that last extent? There are several possible scenarios that one may think of:

1. The remaining space is simply wasted and remains allocated "just in case" it will eventually be grown again.

This would be a very poor design, as we can never guarantee that the file will actually grow. If it does not, the space would be permanently wasted. Because we cannot guarantee eventual growth of the file, there would have to be some kind of timeout mechanism that would free the extent after some time, or when space is urgently needed. But it would be very hard to do this right, and it may never be done right.

2. The used part of the extent is re-allocated to a new, smaller extent and its contents are copied to there. After the copying process has finished, the long extent is freed.

That would certainly be possible, but it has a number of disadvantages. The first one is: do we really want to copy half a terabyte, just because we allocated another terabyte and only filled half of it? This takes several hours of permanent read/write I/O! The second one is: if there are so few extents free in a VxFS file system (see the example above), and we fill them ideally, when and how are we ever to get new, smaller extents? This is answered by the third possibility, which is how VxFS actually goes about in the case sketched above

3. The remaining part of the large extent is cut into smaller extents, and those are added to the free list.

Once you understand VxFS's allocation its elegance becomes obvious. Let's say we have allocated a 2 GB extent but only used 14 MB of it when we close the file. VxFS now chunks the 2 GB extent newly by splitting it into new extents of appropriate sizes:

The part of the extent that was actually used (fourteen MB) is split into an eight MB and a four MB and a two MB extent. The data is not moved, instead the 2 GB extent is simply split into several smaller extents, all being contiguous inside the original 2 GB extent. Eight plus four plus two is fourteen, so all the data can be held in just these three extents. Now there is 2 GB minus 14 MB left in the previously allocated 2 GB extent. This space is also simply rearranged into smaller extents of 16, 32, 64, 128 MB and so on, until the "unused half" of the original extent, 1 GB, is reached. All these extents are added to the free extent list and can subsequently be used for new files or for growing existing files.

VxFS actually manages by far the best trade-off between allocation size and amount of storage space wasted. It does not waste more than one KB per file, and at the same time, a single allocation unit can still be (currently) one TB in size. And if you used the appropriate tools provided by Veritas for this purpose, you could even pre-allocate one TB of contiguous space for your (data base) file and end up with just this one single extent descriptor in the inode, for a file of one TB!

We have prepared two little examples for you. The first example shows the relative outcomes for UFS versus VxFS on newly created file systems. In these cases, VxFS seems to have an advantage only on small volumes, while it seems to allocate much more space than UFS does on larger volumes. But the second example proves the opposite is right, and VxFS is actually much more efficient in handling space; it just doesn't show it as long as the file system is nearly empty. So let's look at the first example. We have prepared a number of volumes of various sizes and put a UFS on them, then we took the same volumes and put a VxFS on them. We ran **df -h** on both sets and will look at the results. First, the UFS file systems:

```
Filesystem             size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/100mvol    93M   1.0M    83M     2%    /ufs100m
/dev/vx/dsk/adg/1gvol   961M   1.0M   903M     1%    /ufs1g
/dev/vx/dsk/adg/10gvol   9.8G    10M   9.7G     1%    /ufs10g
/dev/vx/dsk/adg/50gvol    49G    50M    49G     1%    /ufs50g
/dev/vx/dsk/adg/500gvol   492G    64M   487G     1%    /ufs500g
```

Compare this to the VxFS file systems:

```
Filesystem             size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/100mvol   100M   2.1M    92M     3%    /vxfs100m
/dev/vx/dsk/adg/1gvol   1.0G    17M   944M     2%    /vxfs1g
/dev/vx/dsk/adg/10gvol    10G    20M   9.4G     1%    /vxfs10g
/dev/vx/dsk/adg/50gvol    50G    78M    47G     1%    /vxfs50g
/dev/vx/dsk/adg/500gvol   500G   191M   469G     1%    /vxfs500g
```

The first thing that jumps at you is that you see the actual volume size (e.g. 100 MB for the first volume) displayed in the file system **size** column. But that is just for VxFS! UFS shows a volume size of a mere 93 MB. And now look at the **avail** column. UFS shows just 83 MB versus VxFS showing 92 MB. So we have three questions to answer:

1. Why does UFS not show the full size of the volume, while VxFS does?

2. What took the additional 10 MB away from the UFS?
3. What took the 8 MB away from the VxFS?

The answer to question 1. is rather interesting: UFS dedicates a well defined set of blocks per file system to metadata such as inodes, bitmaps etc. That metadata is located at fixed locations in the file system and cannot be relocated. It takes up a significant portion of the volume and is not available for user data, only for meta data. In the given case, it is 7% of the total volume size, which reduces the available size to 93 MB out of 100 MB volume size (that ratio improves with increasing volume size as you can see from the 500 GB volume: 492 GB is the reported size, so 8 GB are wasted – a mere 1.5% instead of 7%). The ratio of meta data to user data can be tuned by manually supplying the appropriate parameters to **mkfs** when making the file system (e.g. **mkfs -o nbpi=4096** … ->.4 KB of user data are expected per inode instead of the default 2 KB).

The answer to question 2. is probably well-known: UFS reserves some amount of the space (10% for small volumes, less for large volumes) for the root user. The default is ((64 MBytes/volume size)*100), rounded down to the nearest integer and limited between 1% and 10%, inclusively. This space can not be allocated by other users. This reservation is intended to prevent a non-priviledged user from filling up the / or **/var** file system, which might render the system unusable. Because UFS reserves some space for the root user only, the super-user could then still log in and clean up; there is still some working space left over for him in the / or **/var** file system. So this space is not really lost, it is merely reserved and can only be used by root. It could also be tuned to a minimum by manual intervention or by supplying the appropriate parameters to **newfs** or **mkfs** when making the file system (e.g. **mkfs -o free=99** ... -> only one percent is reserved).

The answer to question 3 is more interesting: VxFS writes the free extent records into a relatively large area called an **extent_map**. This **extent_map** is actually a file that is not visible to the user (except by using the **ncheck** utility – see the chapter about Point-In-Time Copies to learn more about ncheck for VxFS). It is a meta-data file. Now why does the **extent_map** take up 8 MB? This sounds like a lot! What VxFS seems to be doing is create all whole lot of, maybe even all, permutations of the free extents. For instance, if you had a file system with 16 MB free, you could create just one free extent record for that space. But you could also create two eight MB ones, or four four MB ones, or one eight and two four MB ones and so on until you have enough extents of every conceivable size ready to allocate as soon as one is requested. It goes without saying that this size grows more quickly than linear with growing volume size, because twice as much free space can be recombined in many more than two times the number of ways.

That is just one reason why the ratio of available space to volume size improves for UFS relative to VxFS with increasing volume size. For instance, the 500 GB UFS file system seems to have much more space available (487 GB) than the VxFS one (469 GB). But you will see that, as the file system fills up, the amount of blocks used for the extent_map decreases rapidly, until the extent map finally takes up almost no space at all when the volume is completely full. You will see, as a result, that a VxFS can still be filled to exactly the size of the volume (500 GB) minus the amount of used space (191 MB), while UFS can only be filled to the reported size (492 GB), so in the given example it wastes 8192 MB, while VxFS's overhead is just 191 MB!

The other reason why the **df** report for VxFS shows much less space available than one would derive from subtracting the used value from the size value is that VxFS pre-allocates lots of inodes when the file system is created. Each inode takes up 256 bytes (by default; it can be tuned to 512 bytes but there is no good reason to do so). These pre-allocated inodes are hidden from the usage because they are not really active. On the other hand, they are subtracted from the size in the output of the available column because that is a more realistic value in case you fill the file system with small files. The difference between UFS and VxFS here lies in the fact that UFS cannot claim the inodes back, while VxFS can. UFS therefore wastes lots of space for inodes that may never get used, while VxFS keeps all this dynamic and allocates only the amount of storage for meta data that it actually needs.

The second example will prove that. We have created a 1 GB volume for UFS and a 1 GB volume for VxFS. We will mount the two file systems and then fill them with data. The UFS will be filled by the non-priviledged user **luser** until the file system reports an overflow. Then, the root user will fill up the rest. We will then do the same for VxFS.

```
# newfs /dev/vx/rdsk/adg/1gvol
newfs: construct a new file system /dev/vx/rdsk/adg/1gvol: (y/n)? y
[...UFS file system is created...]
# mount /dev/vx/dsk/adg/1gvol /ufs1g
# df -h /ufs1g
Filesystem                   size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/1gvol  961M   1.0M   903M     1%    /ufs1g
```

File system has been created and mounted, 961 MB are free for data, 903 MB of which is free for non-priviledged users. UFS has wasted 39 MB on meta-data at fixed locations. We will now switch to the **luser** account and fill up as much of the file system as we can.

```
# su - luser
luser $ dd if=/dev/zero of=/ufs1g/USERDATA bs=1024k
dd: unexpected short write, wrote 344064 bytes, expected 1048576
903+0 records in
903+0 records out
luser $ exit
# df -h /ufs1g
Filesystem                   size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/1gvol  961M   904M     0K   100%    /ufs1g
```

The **luser** could only write 903 MB plus 344064 bytes (see error message form **dd**).

```
# dd if=/dev/zero of=/ufs1g/ROOTDATA bs=1024k
dd: unexpected short write, wrote 663552 bytes, expected 1048576
58+0 records in
58+0 records out
# df -h /ufs1g
Filesystem                   size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/1gvol   961M   961M     0K   100%    /ufs1g
```

Another 58 MB plus 663552 bytes could be used by **root**.

```
# umount /ufs1g
# mkfs -F vxfs /dev/vx/rdsk/adg/1gvol
[...VxFS file system is created...]
# mount -Fvxfs /dev/vx/dsk/adg/1gvol /vxfs1g
# df -h /vxfs1g
Filesystem                    size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/1gvol    1.0G    17M   944M     2%    /vxfs1g
# su - luser
luser $ dd if=/dev/zero of=/vxfs1g/USERDATA bs=1024k
dd: unexpected short write, wrote 691200 bytes, expected 1048576
1007+0 records in
1007+0 records out
luser $ exit
# df -h /vxfs1g
Filesystem                    size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/1gvol    1.0G   1.0G     0K   100%    /vxfs1g
```

In the case where a VxFS is sitting on top of the volume, the **luser** can fill the volume up with 1007 MB plus 691200 bytes of data – much more than even **root** was able to in the case of UFS! Exactly the 17 MB that were reported **used** by the VxFS could not be allocated, every byte of the rest could. But how did we get from 944 MB available space up to 1007 MB? The trick is that, while we are filling up the volume, we are allocating free extents by the thousands. VxFS is allocating them from the **extent_map** file, which takes up a lot of space in an empty file system, as we explained in the first example. The meta data for all permutations of these free extents are of course invalid once we have allocated the space that they point to, so these entries are then freed, which makes the extent_map considerable shorter. It turns out that the more the file system fills up the more meta data is freed from the **extent_map**, and thus the more free space becomes available for user data. Clever idea, is it not?

Now the question that any critical UNIX administrator must ask is: Why does VxFS not reserve any space for the root user? Is that not dangerous?

The answer is: No! If it were dangerous, don't you think the appropriate mechanism would have been implemented in VxFS. The explanation is that VxFS (on Solaris, at least), can not be used for any of the boot file systems, like /, /var, /usr, or /opt. And because it cannot be used for any of these file systems, there is no danger that a **luser** fills up the "VxFS root-FS" and the system gets stuck. There is simply no possibility to create something like a "VxFS root-FS".

But the smaller amount of meta data compared to UFS is not the primary advantage. It is just "yet another nice thing" about VxFS. The **really** nice things that are enabled by using extents instead of blocks (thereby limiting the complexity of handling contiguous allocation and reducing the amount of metadata) are discussed in the following section.

12.3 Advanced File System Operations

On-the-Fly File System Optimization

Yes, you can optimize a VxFS just like you could optimize old FAT file systems (which needed it a lot more, due to its very poor allocation algorithms). The difference is that you can do so while the file system is active and undergoing user I/O. It is actually a very good idea to optimize your file systems (especially database table space files) daily, e.g. just before you start backing them up. The utility is the same fsadm that we used above to check the free extent list: **fsadm.** This time, the parameter is not **-E**, but **-e**. While **-E** simply outputs the report about extents and about possible extent optimization, **-e** actually causes the **fsadm** command to execute the optimization. If the file system had been in use for a long time and especially if it had run almost full for some time, you may see a significant advantage in file system performance after optimization compared to before. Here is an example of the (very simple) syntax

```
# fsadm -e /vxfs_mnt
```

There is also an optimization feature for the directories. During directory optimization, the most recently used files are moved to the beginning of the directory, some empty directory slots are created to speed up file creation, and small directories will be moved into their inode in order to save disk seeks. Several other directory optimizations are performed but due to extensive operating system caching strategies as well as the fact that enterprise-critical applications typically store their data in databases instead of flat file systems, these may not dramatically improve your performance. For completeness (and because directory optimization certainly does not hurt) we give you the necessary options for directory optimizations: They are **-D** for the report and **-d** for the actual optimization.

If you wonder if it is a good idea to do extent and directory optimization on the most critical file systems on a routine basis then the easy answer is: yes! Do it daily, just before you back up the file system, for example. If you do it routinely, there will be very little to optimize every day, and so the command will not take a long time to run. If you want a number: one minute per 20 GB-30 GB of file system data is a good average value for file system optimization (extents and directories) using a typical file system on 2008 hardware.

Sample command to optimize extents and directories:

```
# fsadm -de /vxfs_mnt
```

Storage Checkpoints

Being able to remember the file system state at a particular point in time (see also the chapter about point-in-time copies) is a very nice thing, especially if the overhead in terms of storage space, processing time, and I/O latency is minimal. The VxFS file system provides a point-in-time scheme that satisfies all of these needs, as well as supplying the point-in-time copies as read-write full file systems which can be used recursively to create a whole tree of point-in-time copies. These point-in-time copies are called **storage checkpoints**. Being a feature of VxFS rather than VxVM, they have nothing whatsoever to do with the underlying volume or partition, i.e. you can use storage checkpoint on a partition as well as a SunVM metadevice or a VxVM volume. The only prerequisite is that the file system is of type VxFS, because only the VxFS driver knows how to handle them.

All of the storage checkpoints of a file system can be active at the same time, can be read from and written to, and can again be used as the basis of more point-in-time copies. It does much of what a source code control system is destined to do: keeping track of different instances of data that derive from the same ancestor. And like a source code control system, you can even consolidate your file system to a preferred instance, for instance if you want to reduce complexity or free storage space. Other features that are available with storage checkpoints are the ability to automatically discard them if the volume (or partition) fills up and many other things which are more rarely used. All of this can be done on the fly, provided the main (or original) file system is mounted.

In order to reduce redundancy we would like to point you to their in-depth description in the Point-In-Time-Copies chapter beginning on page 282.

File System Conversion from UFS

Using the relatively simple command **vxfsconvert** the user is able to convert a UFS file system into a VxFS file system. The process takes about as long as an **fsck** run and it is not harmful if the system crashes or faults in the meantime, i.e. the whole process is transactional. What **vxfsconvert** does is read the UFS meta data (much like **fsck** does), gather them, allocate new VxFS meta data, and write that new meta data into free blocks of the UFS file system. When the whole file system has been processed and all VxFS meta data has been written into blocks that UFS considers free, the user is asked whether to commit the transaction. If the reply is positive, the **vxfsconvert** command then writes a new superblock onto the file system. This superblock points to the VxFS meta data rather than the UFS meta data. From the VxFS meta data point of view, the same files are referenced as were references from the UFS file system. Those blocks that contained UFS meta data, allocated or not, are marked free in the VxFS meta data, so they can be overwritten when the file system is mounted read-write.

Before the file system can be mounted, you must first do a full fsck on the new VxFS file system, i.e. **fsck -F vxfs -o full $RAWDEVICE**. This is because VxFS is a rather complicated file system and the **vxfsconvert** command does not need to duplicate things that **fsck_vxfs** implements anyway. We therefore pass the rest of the work to **fsck_vxfs**, which cleans up the file system for us. It does complain a lot, but all the complaints can be safely ignored. It is advisable to specify the **-y** flag to fsck to have **fsck** run continuously rather than stopping to ask for every file.

Here is a full run of converting a UFS on Solaris 10 into VxFS. The file system is almost completely full with data from a previous benchmark run:

```
# ls -l /ufs100m
total 48
drwxr-xr-x  18 root     root          512 Aug 23 19:38 0
drwxr-xr-x  18 root     root          512 Aug 23 19:38 1
drwxr-xr-x  18 root     root          512 Aug 23 19:40 10
```

```
drwxr-xr-x  18 root     root         512 Aug 23 19:41 11
drwxr-xr-x  18 root     root         512 Aug 23 19:41 12
drwxr-xr-x  18 root     root         512 Aug 23 19:41 13
drwxr-xr-x  18 root     root         512 Aug 23 19:41 14
drwxr-xr-x  18 root     root         512 Aug 23 19:42 15
drwxr-xr-x  18 root     root         512 Aug 23 19:38 2
drwxr-xr-x  18 root     root         512 Aug 23 19:38 3
drwxr-xr-x  18 root     root         512 Aug 23 19:39 4
drwxr-xr-x  18 root     root         512 Aug 23 19:39 5
drwxr-xr-x  18 root     root         512 Aug 23 19:39 6
drwxr-xr-x  18 root     root         512 Aug 23 19:40 7
drwxr-xr-x  18 root     root         512 Aug 23 19:40 8
drwxr-xr-x  18 root     root         512 Aug 23 19:40 9
drwx------   2 root     root        8192 Aug 23 19:29 lost+found
```

Because the file system is full we need a little more space for the conversion. It is sometimes tricky to add a whole lot to a file system in one go. If it does not work, split it into several increments. The increments may get larger very quickly. Note that growing in increments is only necessary with file systems that have almost no free space left!

```
# vxresize ufs100mvol +1m
# vxresize ufs100mvol +4m
# vxresize ufs100mvol +25m
# umount /ufs100m
```

Here we actually convert the UFS file system to a VxFS one. We'll measure the time it takes. Remember there are about 50.000 files in the file system, plus the machine is currently heavily loaded.

```
# time vxfsconvert /dev/vx/rdsk/adg/ufs100mvol
UX:vxfs vxfsconvert: INFO: V-3-21842: Do you wish to commit to conversion? (ynq)
y
```

If we reply "**n**" then nothing happens. The superblock is not updated, and we can mount the UFS again. But we chose "**y**", so the superblock is updated.

```
UX:vxfs vxfsconvert: INFO: V-3-21852:  CONVERSION WAS SUCCESSFUL

real    1m15.96s
user    0m2.86s
sys     0m1.22s
```

Now we need the VxFS-savvy **fsck** program to clean up behind our conversion:

```
# fsck -F vxfs -y -o full /dev/vx/rdsk/adg/ufs100mvol
super-block indicates that intent logging was disabled
cannot perform log replay
```

```
pass0 - checking structural files
pass1 - checking inode sanity and blocks
pass2 - checking directory linkage
pass3 - checking reference counts
pass4 - checking resource maps
fileset 1 au 0 imap incorrect - fix (ynq)y
fileset 1 au 0 iemap incorrect - fix (ynq)y
fileset 999 au 0 imap incorrect - fix (ynq)y
fileset 999 au 0 iemap incorrect - fix (ynq)y
[…]
fileset 999 au 5 iemap incorrect - fix (ynq)y
fileset 999 au 6 imap incorrect - fix (ynq)y
fileset 999 au 6 iemap incorrect - fix (ynq)y
no CUT entry for fileset 1, fix? (ynq)y
no CUT entry for fileset 999, fix? (ynq)y
au 0 emap incorrect - fix? (ynq)y
au 0 summary incorrect - fix? (ynq)y
au 0 state file incorrect - fix? (ynq)y
au 1 emap incorrect - fix? (ynq)y
[…]
au 3 summary incorrect - fix? (ynq)y
au 4 state file incorrect - fix? (ynq)y
au 4 emap incorrect - fix? (ynq)y
au 4 summary incorrect - fix? (ynq)y
au 4 state file incorrect - fix? (ynq)y
fileset 1 iau 0 summary incorrect - fix? (ynq)y
fileset 999 iau 0 summary incorrect - fix? (ynq)y
[…]
fileset 999 iau 5 summary incorrect - fix? (ynq)y
free block count incorrect 0 expected 30955 fix? (ynq)y
free extent vector incorrect fix? (ynq)y
OK to clear log? (ynq)y
flush fileset headers? (ynq)y
set state to CLEAN? (ynq)y
```

Once the file system has been checked we can now mount it using **mount -F vxfs …**:

```
# mount -F vxfs  /dev/vx/dsk/adg/ufs100mvol /mnt
# df -h /mnt
/dev/vx/dsk/adg/ufs100mvol   130M   100M    30M    77%    /mnt
```

As you can see we now have a 130 MB volume which is filled with 100 MB (hey, we told you it was full, didn't we? That's why we added another 30 MB before the conversion). The remaining 30 MB are free for user data (unlike UFS, which would have taken a large bite out of that for its own statically allocated meta data(..

Suggested reading: If you want to know more about file systems, and especially clustered file systems, "Shared Data Clusters" by Dilip M. Ranade (lead technical engineer for Veritas cluster file systems) is a great book.

12.3.1 Summary

This chapter gave an introduction into file systems based on an overview of the development of file systems and especially their allocation mechanisms. It shows that the use of extent-based allocation is superior to block-based allocation because of the smaller overhead per block, less compute-intensive algorithms and more direct access to the resulting disk blocks for any I/O. Also, VxFS uses the available space almost ideally, unlike UFS, which always allocates fixed amounts of metadata for some assumed worst case scenarios (like having very few, very large files, or having very many very little files). VxFS allocates its metadata dynamically and is therefore much more flexible and space-efficient.

Some of the features of VxFS were outlined, especially growing and shrinking file systems online, optimizing existing file systems, and using storage checkpoints. Both rely heavily on extent-based allocation and especially storage checkpoints would be very hard to implement efficiently with any block-based allocation scheme.