io-blk#

io-blk is a virtual file system for QNX block file systems such as dos, cd, ext2, qnx4, udf, ... It is similar (but different) to those used in the various Unix flavours (such as BSD). If you are already familiar with vnode based virtual file systems (VFSs), then you can probably skip this page.

First, let's start off with some terminology and file system basics. A file system typically resides on some form of media such as a hark disk, floppy, usb drive, flash chip, ram, ... The media is divided into blocks of equal size. In the case of a hard disk, the smallest block size is a sector. A hard disk sector is still typically 512 bytes, but there have been reports that this may change to 4kB in the future. A group of one or more consecutive sectors is called a block. The number of sectors in a block is a power of two.

File system on-disk structures will generally fall into one of three categories: file data, file meta-data, and file system meta-data. File data is what we are all most familiar with. The data we read from, or write to the file is file data. The file meta-data, is information on the disk that describes the file. It includes file attributes such as its size, time stamps, and which media parts are assigned to that file. That is, it is the behind the scenes information that is necessary for accessing the file. The file system meta-data is all the supporting information required to maintain the file system. At the very least, it includes structures for tracking which media blocks are in use. It may also include structures for tracking which files are in use.

Let's take a look at some of the major structures and themese that can be found in a file system.

Directories are a special type of file. A directory is composed of directory entries. At the very least, each entry associates a name with a file, and has information about where to find that file on the media. Some file systems will use fixed length directory entries, while others may use variable length directory entries. Some file systems, such as the dos file system, will embed the file attributes and some file meta-data into the directory entry, while others such as ext2 will contain the file's inode number so that the attributes are separate from the directory entry.

What is an "inode"? An "inode" describes everything about the file except for its pathname. It contains all the file's attributes and information about where to find the file's data on the media. Some file systems embed the inode into the directory entry. Other file systems group all the inodes for the file system into an on-disk structure called the inode table. Either way, given an inode number, one can trace it to the inode, and from there, get the file meta-data and finally the file data. There are advantages and disadvantages to using the embedded approach and the separate inode table approach. First, the inode table offers more flexibility; it de-couples the file from the file name. This allows one to easily create both hard links and symlinks to the file. Hardlinks and symlinks are much more difficult to do with embedded inodes; in fact, the dos file system does not support them. However, this de-coupling can have a performance penalty. To open a file, or get information on a file, it must read the inode from the inode table (which exists in an area separate from the directory). When the medium is a hard or floppy disk, this may involve moving the disk head--a rather slow operation. The QNX4 file system can use a synthesis of the two. As most files are neither hard-linked nor sym-linked, it can embed the inode in the directory entry. However, if a hard-link or a sym-link is created to the file, it will use an external inode from that point forward.

Every file system needs some way to track which media blocks are in use, and which are not. One way to do this is to use a "freespace bitmap". Each bit in the bitamp maps to a block on the underlying media. The value of this bit indicates whether the block is currently in use or not. Similarly, when an "inode table" is used, the file system may use an "inode bitmap" to track which inodes are in use.

The file system also needs to track which media blocks are associated with a particular file. There are many ways of doing this. The dos file system uses a singly linked list strategy to track the blocks for any given file, with the starting block listed in the directory entry. The ext2 file system uses a tree system with up to three levels of indirect blocks. A direct block contains disk data. In indirect block contains is filled with direct block numbers. A doubly indirect block is filled with indirect block numbers. A triply indirect block is filled with

doubly indirect block numbers. The top-level block numbers of the tree are stored in the inode. The QNX4 file system uses extents. An extent describes a set of contiguous blocks.

Once again, each method has its own strengths and weaknesses. The dos method is easy to implement, but it has performance issues when one has to perform seek operations in the large and fragmented files. The ext2 method is a more complicated to implement, but it does not suffer the same performance loss on large fragmented files, and it allows sparse files. However, its meta-data trees may take up megabytes of media space. The QNX4 method of extents works very well with non-fragmented files--it allows fast seeking and the file's meta-data does not consume much disk space even on large files. However, if the file is both large and fragmented, things begin to to slow down and the meta-data takes more space on the media.

But what does this have to do with a VFS? Glad you asked. :) A VFS has to accomodate many different types of file systems that do things very differently. The basis of the io-blk VFS is the vnode. A "vnode" is an in-memory construct that stands for virtual inode. Not all file systems use inodes, and those that do use inodes, do not store the same information in them. To get the file systems that don't use inodes to work with the VFS, their implementatoin must fake the inodes. Just as inodes contain information for finding the media blocks that contain the file's data, a vnode contain information for tracking a file's cached data blocks. A VFS tries to abstract away the file system implementation as much as possible. Obviously, not everything can be abstracted away. At some point the VFS will have to perform a call-out to the specific file system to retrieve or write information.

I'm going to classify the call-outs into two categories--vnode operator callouts and vfs callouts. The vnode operator callouts operate on a vnode basis. They are used to update or retrieve information associated with a particular file. These callouts very, very closely follow the resource manager handlers. There are a few exceptions, but I won't go into that here. The second categroy are the VFS operator callouts. These callouts are associated with the file system--mounting, unmounting, getting the root directory, stat'ing the file system, ... The aim of the VFS is to take care of as much of the common file system operations as practical. For file system specifics, the VFS makes the appropriate callout. This makes implementing a file system much easier--all one has to do is fill in specifics for a particular file system. Even when we get down to the file system specifics, they are still going to share some functionality. Ultimately, each file system is going to have to perform media (typically disk) operations. VFS comes to the rescue again. It provides a standard interface for the file systems to interface with the various devb-xxx drivers such as devb-eide. The higher levels of the VFS perform callouts to the file system specifics. The file system specifics may make callouts to the lower levels of the VFS. It sounds all so very neat and tidy. Each used buffer is associated with a file system and a block number. Each used vnode is associated with a file system and an inode. The blocks and buffers also get associated with a vnode for caching purposes.

Any questions?