The New Copy On Write File System#

Are you wondering about the QNX's new copy on write file system? This page will give you a bit of an overview as what it is about. This file system plugs into QNX's io-blk framework. If you are already familiar with that, it should be easier to pick up the implementation details.

This file system was designed and developed in response to a problem. When most file systems start up, they must perform an integrity check if they were not shut down correctly; sometimes they must do it even if the did. When your file system contains gigabytes of both file-data and meta-data, this can take a looooooong time. Such a feature is not particularly desirable in a real time system, or in embedded products. What if the file system could every time come up as fast as you can snap your fingers. What if you never had to do that file system integrity check, even if you lost power in the middle of an operation? That would be cool. That would be fs-qnx6.

How does it accomplish this? It never rewrites over live data, or live meta-data. If this is starting to sound like a promo or advertisement for it, well, we're just rather excited about it. Can you blame us? ;)

Like many other file systems, there are free space bitmaps, free inode bitmaps, superblocks, ..., but more on that in a minute. First, let's walk through the conceptual steps as to what happens when you modify a file. Imagine you are writing 100 bytes to a file. First, you have to identify the blocks in which the 100 bytes would reside. For the sake of argument, let us say it was one block--file block number 1234. We will have to allocate a new block for it. The old data in this block (if there was any) will be copied to the new block, and the new 100 bytes will be written to it. Notice that the old data remains intact, and is completely unmodified. So far, the file's meta data will still point to the old block. This needs to be updated as well. But wait! As the meta-data will be updated, we must allocate new blocks for block that gets modified. The file meta-data that maps the file's logical block to the file's physical block is done in a manner similar to UFS and Ext2. If file block number 1234 uses a two levels of block indirection, then at least three blocks will need to be allocated -- two indirect blocks, and the data block. But wait! If this would change the inode, then more blocks need to be allocated as well. Changing even a single byte can cause ripples that would necessitate the allocation of over a half dozen blocks.

If you're starting to think that this file system must be incredibly slow if it has to do all these steps when changing a single byte, you would be wrong. There are two main reasons. The first reason is that most people do not write a single byte at a time to a block device. The second reason is that a commit operation is not typically done after every operation. Thanks to the cache, operations can be grouped together. When the commit occurs, many changes are sent to the disk at once. The last step in a commit is to write out the new superblock (there are two of them) with the new sequence/transaction number. The superblock with the larger sequence/transaction number is the most recent.