Daniel Perry
cs5460 / fall 2005
Homework 2 - File systems

1. **File system optimization vs consistency**
   Because of the cost of actually physically reading/writing from/to disk is to expensive, optimizations for file systems will deal mainly with reducing those functions. For example, in class we've talked about some redundancy or logging, but to get the best performance in writing and reading directly from disk, you would only actually record data once (no redundancy or logging), preferably in a large contiguous chunk of the disk. However the optimization that would give the best improvement in optimization would be in cacheting the contents of disk of memory. In an extreme example, assuming you have enough memory, the best performance would probably come when you cache the entire disk in memory, only perform changes to this representation in memory, and then on shutdown or at a certain time, record the state of the in memory file system to physical disk.
   Although caching the file system in memory improves performance the most, it also allows for the worst consistency if the machine were to crash. If all changes are only maintained in volatile memory, then when the machine crashes, we will lose all changes, that aren't recorded to the disk.
   Ways we can deal with this situation, while still taking advantage of caching is to limit the amount of time that changed data can remain only in memory without being reflected on disk. For example establish a rule, that changes must be written to disk every 30 milliseconds or something like that. This way you benefit from caching in memory (especially with reads), but because you write back changes at specific time intervals, the data is safe, up to the moment in time you last wrote it to disk.
   Additionally, when writing the data to disk, it is important to do it in a way that will either be easy to recover from using some disk utility like fsck, or using logging or some other method so that if the machine crashes while you are writing to disk, you can still recover from it when you reboot.
2. **File System Organization**
   To read a file 1,024,000 bytes (or 1000 straight blocks), in the established setup, you would need to read 1005 blocks = 1000 data blocks + 5 overhead blocks.
   Broken down:
   You would first read the 12 direct pointers each pointing to a data block of 1024 bytes = 12,288 bytes of data.
   You would next read in the 2 indirect pointer blocks (overhead) plus the 512 data blocks they point to, each block with 1024 bytes = 524,288 bytes, bring the total data bytes to 536,576 bytes (524 data blocks).
   (It may be important to point out that, since each block is 1024 bytes and an address is 4 bytes, I assumed each indirect block and doubly indirect block could hold 256 addresses.)
   You would next read in the 1 doubly indirect block (overhead), as well as 2 indirect pointer blocks (overhead) that it points to (it points to a total of 256, but you only need 2 more). From those you would read in all 256 blocks of the first, and only 220 blocks of the second (since that's all that you need to reach the file size of 1,024,000 bytes).
   That brings the total to 1000 data blocks, and 5 overhead blocks (2 indirect + 1 doubly indirect + 2 more indirect).
   Note: if for whatever reason you were unable to stop reading in blocks in the middle of an

indirect pointer (ie have to read 256 not just 220) it would add 36 blocks to the total.

3. **RAID Systems**
   Assuming that we have the same number of disks in both cases, no – the best RAID1 could do is the same as RAID0 with striping.
   Explained:
   With the same number of disks, the only situation where RAID1 could have better performance than RAID0 is if RAID1 could read parts-of-a-block from different disks, reconstruct them in the controller – then it would have better performance as long as in that read, the number of disks was greater than the number of blocks to be read. In that situation, the per block read time for RAID1 would be lower than for the RAID0, around (time for one block)/(number of disks). But, once the number of blocks to be read was equal to the number of disks, that advantage would disappear since with RAID0 striping, would allow it to read contiguous blocks simultaneously – making it just as fast as RAID1 in that situation, and even faster than RAID1 when the number of blocks became large (since it would be trying to read one block from several disks at the same time – which would be slower than simultaneously reading different blocks from different disks).
   Without that special sub-block-read setup, the best a RAID1 can do is the same as a RAID0 (again this is assuming the same number of disks in each configuration), since RAID1 mirrors the data across all disks, in a read operation, you could read, for example, block 1 from disk 1, block 2 from disk 2, ..., block n from disk n, block n+1 from disk 1, etc. This would give the effectual speedup that striping the data gives you. Therefore RAID1 could only do at best the same as RAID0 without sub-block read and reconstruction.
   Where RAID1 has the advantage always over RAID0 is in fault-tolerance.
4. **Time Travel File system**
   The simplest and most straightforward approach to this problem I can think of is to use a log-based file system, keep the log around for as far back as you would like to be able to fork. When you want to fork, create a complete copy of the file system, and undo all the logged transactions back to your mount time.
   However, an obvious problem to this approach is you would need a double the disk space for every fork, and in addition to logs, you would need to keep files around that were “deleted” so that at roll back you could “undelete” them, unless you wanted to completely rebuild the system from nothing using the logs – which probably wouldn't be best implementation.
   So my idea considers those initial ideas and tries to address the problems.
   Instead of keeping track of a log for that whole time, it would pobably be more efficient ot implement some sort of a version control system – like cvs – but at a lower level and for all files in the system. The basics of the file system are inodes, data blocks, and special data blocks like folders. The version control system then could keep track of changes made to inodes, only recording the difference between it and the previous version. It could record the different versions according to the time at which it was done. Because inodes and special folder data blocks are normally of generally the same size and make up (I mean all inodes are the same size and all folders are mainly the same size), then they could each have there own method of version control. Then general data blocks could have their own version control setup, probably as groups (groups of blocks pointed to by the same inode). This would result in generally three

types of data structures of which to keep appropriate versioning: inodes, folder data, and real data blocks.
As far as implementation, we would be keeping track of generally three date situations – creation date (only one), modifications to original create (multiple), and deletion date (only one). We would of course initially write the data at creation, then at each successive modification, we would need to keep track of both the date and differences between it and it's previous version. Then at deletion, we would want to keep track of only when it was deleted (we would of course NOT de-allocate the data, but only mark the inode/folderfile/group of data blocks as deleted). This information will be what we need when we remount the file system at a specific time.
We will consider the inode situation first. The creation of an inode would be very much the same as in a normal file system. At modification, the inode would have changes made to the data block addresses it contains. The difference between those addresses and old addresses could be saved in a linked "special" data block marked as such. At the point when the inode was supposed to be deallocated or effectively deleted, it would only mark itself as deleted, and record what time it happened at.
The data blocks associated with that inode, when they are initially allocated would also need to keep track of that time and record the data directly to disk. At each modification the difference between the current version and new version would need to be recorded, probably in another specially marked data block. When the file data was deleted it would only mark itself as deleted, but never really deallocate. I would attempt to handle the record of changes according to groups of data blocks instead of individually, though this would add additional care in the cases where you add or subtract the number of data blocks used in a file.
The folder data blocks would be handled in a similar way, but would be a much simpler situation as all folder data blocks would have a similar data structure – not true of anonymous data blocks.
This setup makes it so that inodes and data blocks are not reused.
Making the system usable
Since all these changes are kept in cvs type versions system, it would be pointless to have to walk back through all the different change points to read a file, so one addition to the system would be to keep the current version with all changes in memory. That way on read it wouldn't have to sort through all the different versions. Also, perhaps a physical copy of the current version on disk, since memory probably couldn't hold the entire fs.
Addressing Restoring at new mount
First of all, there are two very important events in the "life" of a file or inode and those are creation and deletion. This is because if the restore time desired lies after the deletion or before the creation time, it does not need to include that information in the new fork. If on the other hand the time desired lies after creation but before deletion (if there is a deletion time) it would need to find the correct version, and export that version. The new fork would create a "current version" of all the files, but maintain relation to where they came from in case you need to go back in time from the new fork. Then new changes would be recorded in the same way as before, but with the indication that these revision blocks are for a that specific fork of the file system.