EXHIBIT 34

XP 000610668

A Cross-Platform Binary Diff

p.D 05L95
p. 32+35-36 (3)

Seeing how one binary file differs from another

Kris Coppieters

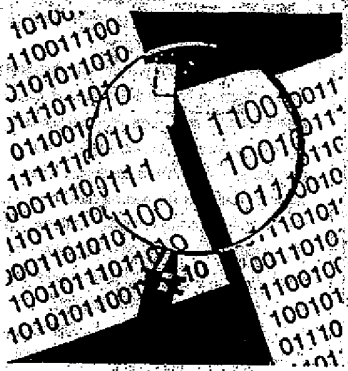
Binary file comparison is useful for many applications. One example is sending updates of large files over a communications line. Instead of sending a complete update each time, you could send the complete file once, then create a diff file containing the differences between the original file and the updated file. At the receiving side, this diff file could be used to update the original file. Creating a diff file is processor and memory intensive. Under DOS, such a process can easily exceed the 640K limit. On the other hand, using a diff file to update the file is very lightweight and fast. In such cases, it may be desirable to create the diff file on another platform and use the resulting file under DOS.

Such were the requirements when I created BinDiff, a utility that intelligently compares two versions of a binary file and creates a diff file. The algorithm in BinDiff tries to find equal chunks in the two files being compared. BinDiff then uses an indexing algorithm to find matching chunks so equal chunks need not be in the same sequence in both files. BinDiff is built from a single C source file that compiles on UNIX, OS/2, DOS, and the Macintosh. A command-line user interface is used on the first three platforms, while a point-and-click interface is used on the Mac. Because BinDiff is insensitive to Endianness, diff files created on one platform can be used on another.

Kris is manager of the service and support department of Logic, an AppleCentre/IBM R/6000 VAR/Novell Partner. He can be reached on CompuServe at 100025,2724.

Chunking the Binary File

Many diff utilities only work with pure text files because they depend heavily on the concept of a "line" denoted by some kind of delimiter. Dividing files into lines lets you index the lines in one file, then use the index to find matches with lines in another file. A binary file, however, has no such delimiter, making it harder to index. I chose an artificial line delimiter (a single byte code between 0 and 255) to divide a binary file into fake lines, which can be indexed and processed just like lines in a text file.



The algorithm to choose a suitable delimiter involves simple statistics. For each possible byte value, I calculate the mean length and standard deviation of the lines in the file. From the byte values with a mean block length between 20 and 80 bytes, I choose the one with the lowest standard deviation. If there is no such byte, I gradually loosen the limits on the block length, trying values from 20 to 130, 20 to 180, and so on. In the rare cases that this does not help, 0 is used as the byte value. Choosing the lowest standard deviation yields the byte that most evenly divides the file; the blocks are more or less the same length. I tested this algorithm on a number of text files, and in many cases, the most suitable delimiter coincided with the actual line delimiter (CR or LF).

Indexing and Matching

Once both files are divided into chunks, BinDiff builds an index and begins matching chunks from both files. I've arbitrarily chosen to index the original file (*<file.old>*). I read the file and create an unbalanced binary tree of chunks. Each tree node contains a left and right pointer. In each node, I also store some bytes from the file to speed up comparisons. The first few bytes to compare do not have to be read from disk. I also add sequences of at least ten equal bytes to the chunk-index tree, regardless of delimiters.

At this point, BinDiff can match the updated file (*<file.new>*) against the index tree. The updated file is read and chunked using the same delimiter. Each chunk or run of at least ten equal bytes is looked up in the tree. Every match of sufficient length is expanded as much as possible and stored in a linked list of matches. A match can be smaller than a complete chunk. If six or more bytes match, the match is stored. Due to the file format used, matches of less than six bytes increase the size of the diff file instead of reducing it. I try to expand a match both forward and backward, because the two files may already match prior to the location where the match is found, and they might continue to match after the delimiter that ends the chunk; see Figure 1.

Distance, Expanding, and Enclosing

The "distance" of a match is the difference between the file positions in the original and updated files. If a match is at the same position in both files, for example, its distance is 0. A match is an "expanded" form of a smaller match if the starting position of the smaller match is within the larger match and if the distance of both matches is the same. A smaller match is "enclosed" by a larger match if the cor-

(continued on page 35)

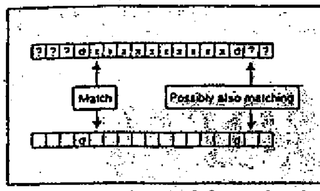


Figure 1: Matching both forward and backward.

(continued from page 32)
responding smaller chunk in the updated file is enclosed in the larger chunk. One match can enclose another without being expanded: Enclosing matches have unequal distances; expanded matches have equal distances.

The linked list of matches is kept sorted on starting position in the updated file. Before expanding the match, the linked list is checked. If the match is already present in an expanded form, it is not expanded or added; if no expanded form of the match is present, it is expanded and inserted into the linked list. While inserting the new match, all matches enclosed by it are removed from the list. The new match is better because it is bigger. After the match list is built, the complete match list is reviewed. Overlapping matches are cut off, and if a match drops below six bytes, it is removed from the list.

Writing the Diff File

The diff file consists of a header and a sequence of tagged entries. The header contains a signature, file sizes, checksums, and data specifically for Macintosh files (the type and the creator of the updated file). Some tags are used as headers to data included in the diff file. Other tags encode references to data present in the original file.

Finally, the diff file is written. The tags are four bits in size and are contained in the lower four bits of a byte. The upper four bits, together with zero, one, or two extra bytes, contain a chunk size. There is also a block of bytes or two, three, or four extra bytes to encode a chunk file position within the original file.

A 4-bit field is used to encode sizes 1–16 bytes (size 0 is never used); a 12-bit field for sizes 17–4112 (4096+16); and a 20-bit field for sizes 4113–1,052,688. Larger chunks are encoded by using more than one tag. Depending on the size and location of a chunk, I use either a short or a long code: References to small chunks near the start of the file are encoded in three bytes (4-bit tag and 4-bit size, two bytes for file position). The largest reference to a big chunk near the end of the file can be seven bytes (4-bit tag, 20-bit size, 4 bytes for file position), but most will not exceed six.

The diff file contains two separate, sequential diff files. On DOS, UNIX, and OS/2, the second diff file is always empty. On the Macintosh, the file contains the diff information for the resource forks (if present).

Platform Specifics

Macintosh files differ in that they are composed of the data fork and the resource fork. The data fork corresponds to a file on the other platforms. On a low level, the resource fork can be seen as just another file. On a higher level, it is used by the Macintosh OS for maintaining a database-like structure of resources. In BinDiff, the resource fork is viewed as a second file. To keep the code as portable as possible, I created new versions of standard file I/O functions like *fopen*, *fseek*, *fread*, and *fwrite*. My version of *fopen* has an extra option that lets you specify which fork to open. Macintosh diff files from the data fork are usable on other platforms; those from the resource fork are not.

On many UNIX platforms, a K&R C compiler is available. BinDiff uses double headers: Each function has an ANSI header and a K&R header. If an ANSI compiler is available on a particular UNIX platform, it can be activated with one of the conditional compilation switches. Note that the UNIX version has no progress bar—BinDiff simply displays a message after startup.

RedBend0008247

Putting it Together

BinDiff's complete C source code and project files are available electronically; see "Availability," page 3. I've tested BinDiff with Symantec C++ 7.0 and MetroWerks C++ 1.0 on the Macintosh, Borland C++ 3.1 under DOS, Borland C++ 1.0 on OS/2, and a K&R C compiler on A/UX. For the Macintosh version, you'll also need the file BinDiff.rsrc, which contains the resources for a dialog-window layout. The Symantec project file should contain: BinDiff.c, ANSI++, CPlusLib, and MacTraps. I put the library files in a separate segment. I use *BIdF* as creator and .DIF as file type. Because the standard file functions such as *fopen* and *fseek* are already present in ANSI++, I define the corresponding functions FOPEN, FSEEK, and so on, and use macros to convert lowercase functions to their uppercase variants (for example, *=define fclose(x) FCLOSE(x)*). This prevents linking problems.

Using Borland C++ under DOS and OS/2, you must create a project that includes BinDiff.c, and change the settings so that BinDiff.c is compiled in C++ mode. On most UNIX systems, you can compile with the following command line: *cc bindiff.c -o bindiff -lm*.

The source file contains multiple versions of the program; you can change compilation variables according to the version being compiled. By setting BDEXTR to-1 instead of 0, you compile a reduced version of BinDiff, called "BdExtr," that contains only the code for applying a diff file to an original file. Setting one of the values *BorlandC*, *MacC*, or *StdUnixC* to "1" identifies the compilation platform. The routine *ScanFile* scans through a file and calculates the mean block size and standard deviation for each byte value 0–255 if this byte were to be used as a delimiter.

In BinDiff, the resource fork is viewed as a second file

FindDelimiter checks the tables built by *ScanFile* and chooses a suitable delimiter.

BuildTree scans a file and builds the chunk index tree. *ExtendMatch* extends a match forward and backward, until the first nonmatching bytes or the file limits. *MatchFiles* matches the second file to the first file's index tree. *DumpDiff* writes the tagged diff file. Depending on the platform, part of the routine is executed once (DOS/OS/2/UNIX) or twice (for both forks of a Macintosh file). *SubtractFiles* is the highest-level routine to create a diff from an original and an updated file. *AddFiles* is the highest-level routine to apply a diff to an original file in order to create an updated file.

Conclusion

You can optimize BinDiff in several ways. For instance, the unbalanced tree can become a balanced tree. This yields better performance with already-sorted files (such as sorted text files). Next, consider the possibility of using the zero delimiter when no good delimiter is found. The zero default is probably one of the worst choices, but it is very rarely used, normally occurring only on very small files.

Also, more data could be read into memory. Currently, only a very small part of the file is read into the nodes of the tree, making the algorithm rather dependent on disk I/O performance. When a lot of memory is available, more of the file should be read into the tree. Another optimization is to use CRC instead of the simple checksums used for checking the files. CRCs give more security against using a diff on a nonmatching original file, and against diff file corruption. Finally, the diff file could be compressed.

DDJ