File System Reconstruction Lab

Jared Culp (jjc4fb), Bryan Walsh (bpw7xx)

Unused blocks:

We were unable to compile a list of unused blocks in time. :(

Perpetrator:

We believe the perpetrator is Professor Soroush.

Questions:

1. *Describe your algorithm for solving this problem.*

   To solve this problem, we used the mmap() system call to map the filesystem we were given into virtual memory.

   We perform an initial pass over the entire filesystem, looking for a UID and GID matching 10578 and 1231 respectively. Since we know that the inode is laid out in memory as follows:

```
typedef struct inode {
    int foo;      /* Unknown field */
    int nlink;    /* Number of links to this file */
    int uid;      /* Owner's user ID */
    int gid;      /* Owner's group ID */
    int size;     /* Number of bytes in file */
    int ctime;    /* Time field */
    int mtime;    /* Time field */
    int atime;    /* Time field */
    int dblocks[N_DBLOCKS]; /* Pointers to data blocks */
    int iblocks[N_IBLOCKS]; /* Pointers to indirect blocks */
    int i2block; /* Pointer to doubly indirect block */
    int i3block; /* Pointer to triply indirect block */
} inode_t;
```

   As a result, whenever we find a matching UID and GID, we create an inode and point it to the found location in memory. Since the inodes on the filesystem and the inodes defined in our program are laid out in memory the same way, all of the fields fall into place because of contiguous memory. We keep track of a dynamically growing array of inode pointers, so that we can use them later in file reconstruction.

For each inode that we find, we then reconstruct the file by first grabbing the information pointed to by each of the 12 direct blocks. We also need to multiply this value by the block size, in this case 1024, to get the proper address. We then write this to the output file. For the 3 indirect blocks, there is one extra memory address to “follow” to the data, as the initial memory address is a mapping to another memory address, where the actual data lies. We have to do this for each of the memory pointers that fit in the one block. For the doubly and triply indirect blocks, we simply need to follow one more level of abstraction.

When reading the first direct memory block, we take advantage of the fact that binary files typically begin with a magic number corresponding to a file extension. Those relationships are defined below:

```
char ps[]       = {0x25, 0x21};
char pdf[]      = {0x25, 0x50, 0x44, 0x46};
char gif[]      = {0x47, 0x49, 0x46, 0x38};
char gif1[]     = {0x37, 0x61};
char gif2[]     = {0x39, 0x61};
char tif[]      = {0x49, 0x49, 0x2A, 0x00};
char tiff[]     = {0x4D, 0x4D, 0x00, 0x2A};
char png1[]     = {0x89, 0x50, 0x4E, 0x47};
char png2[]     = {0x0D, 0x0A, 0x1A, 0x0A};
char exe[]      = {0x4D, 0x5A};
char ascii[]    = {0xFF, 0xD8};
char html[]     = {0x25, 0x21};
```

2. *What is the complexity of your algorithm (e.g.,O(n)) in terms of the number of inodes? Number of data blocks?*
   The algorithm needs to initially search through the entire binary file system presented to us. If we define b as the number of bytes in that file system, then the preprocessing is O(b).

   Once we find the inodes, we perform reconstruction on each of the n inodes. Each operation involves the direct, indirect, doubly indirect, and triply indirect memory accesses. The direct memory accesses are O(12); the indirect are O(3), the doubly and triply indirect are O(1). We can regard all of these operations as a constant time operation O(1). Each file then needs to be written, which is proportional to the number of bytes (as an I/O operation), which we can regard as some constant factor since the input sizes are relatively small. As a result, the total operation is O(n), regardless of how many of the memory blocks are used.

To construct a list of unused blocks is a O(b) operation, where b is the number of blocks in the original file system.

3. *What files did you find? Provide a brief description of each (file format and, if known, what the contents represent).*
   We found 4 files that matched the UID and GID given.
   - A PDF file
   - A GIF file
   - A TIFF file
   - A PS file

4. *Which files, if any, use the indirect block? Doubly indirect? Triply indirect?*
   All of the files used direct and indirect blocks. Only the and the files used doubly indirect blocks, since they were the larger files. None of the files used triply indirect blocks.

5. *How many inodes fit in one disk block?*
   A disk block is defined as 1024 bytes. When we run the sizeof() command on the inode struct defined above, we get 100 bytes. Therefore, at most 10 inodes can fit on a single disk block.

6. *If the inodes were not included in the data file, could these files still be reconstructed? Why or why not? If the inodes existed somewhere but the uid and gid were not known, could these files still be reconstructed? Why or why not?*
   If the inodes were not included in the data file, it would be extremely difficult to reconstruct the file. The inode holds valuable information with how the information of the file is laid out in memory. If we have no way to access these addresses, we would essentially have to try to brute force the pieces of the file together. If the uid and gid were not known, we could reconstruct the files, but we would attempt to reconstruct a significantly more amount. By knowing the uid and gid in the original problem, we were able to narrow our search down to only 4 inodes for reconstruction.

7. *How much time did you spend on this assignment, was it fun, and how could it be improved?*
   The concept of this assignment was really interesting; however, the assignment was extremely difficult and we encountered a significant amount of trouble with reading in the hexadecimal data. This assignment was extremely low level (we were working with individual bytes!), but was nevertheless more frustrating than it probably needed to be. Very little direction was given, which seemed to be the source of most of the problem.