Short Assignment – File System

The file system used for this assignment is described on the following page.

Given the file system contents shown in the table below, answer the following questions:

1. Draw or otherwise fully list the directory tree. (to answer in text, a list of full pathnames would be acceptable)

2. Assume that we have read the superblock into memory (i.e. we can translate inode number to disk block without reading from the disk) but nothing else; given the following operation:

```
read(“/dir.1/file.2”, offset=2000, len=1000)
```

- What blocks will need to be read from disk for the following operation, and why?
- For each data block, what range of bytes will be copied into the output buffer?

```
block 0 (superblock):
  magic:       <correct>
  disk_size:   100   (1KB blocks)
  blk_map_len: 1
  in_map_len:  1
  inodes_len:  4     (64 inodes)

block 1: block bitmap
block 2: inode bitmap

mode 040*** = directory,
     100*** = regular file

blocks 3-6: inodes

inode 1:
    mode,size = 040777, 1024
    ptrs[0] = 7
inode 2:
    mode,size = 100666, 1000
    ptrs[0] = 8
inode 3:
    mode,size = 40777, 1024
    ptrs[0] = 9
inode 4:
    mode,size = 40777, 1024
    ptrs[0] = 10
inode 5:
    mode,size = 40777, 1024
    ptrs[0] = 11
inode 6:
    mode,size = 100666,6000
    ptrs[] = {12,13,14,15,16,17}
inode 7:
    mode,size = 100666,4444
    ptrs[] = {18,19,20,21,22}
inode 8:
    mode,size = 100666,2000
    ptrs[] = {23, 24]
```

```
inode 9:
    mode,size = 100666,100
    ptrs[] = 25
inode 10:
    mode,size = 100666,8000
    ptrs[] = {26,27,28,29,30,31}
    indir_1 = 32
inode 11:
    mode,size = 100666,500
    ptrs[] = {35}

block 7 (directory contents):
  "file.1", 2
  "dir.1",  3
  "dir.2",  4
  "dir.3",  5
  "file.6", 6

block 8 (file data)

block 9 (directory contents)
  "file.2", 7
  "file.3", 8

block 10 (directory contents)
  "file.4", 9
  "file.5", 10

block 11 (directory contents)
  "file.7", 11

blocks 12-32 (file data)

block 32 (indirect block)
  ptrs = {33, 34}

blocks 33-35 (file data)

remainder of blocks are unused
```

File system

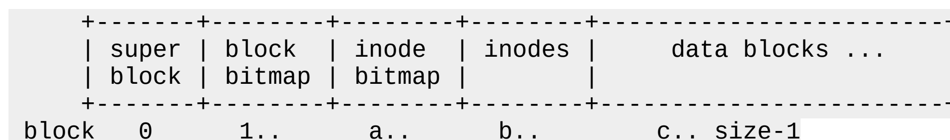
The homework file system is a simplfied version of the `ext2` Unix file system, described in chapter 6 of the testbook, with a simplified on-disk layout including fixed-length directory entries.

Data blocks are 1KB in size. Block numbers are stored as unsigned 32-bit integers, limiting disk size to 1TB. Inodes are 64 bytes, with 6 direct pointers, an indirect pointer, and a double-indirect pointer (see figure 6.4), for a maximum file size of 6 + 256 + 256*256 blocks or about 67MB.

Disk layout

The disk is divided into 5 regions:

1. the superblock, which tells you how big the regions are
2. the block bitmap, for allocating blocks
3. the inode bitmap
4. the inode region, containing 64-byte inodes packed 16 to a 1024-byte block
5. data blocks for directories, file data, and indirect blocks



```
     +-------+--------+--------+--------+-----------------------+
     | super | block  | inode  | inodes |     data blocks ...   |
     | block | bitmap | bitmap |        |                       |
     +-------+--------+--------+--------+-----------------------+
 block   0      1..      a..      b..        c.. size-1
```

Superblock: The superblock is the first block in the file system, and contains the information needed to find the rest of the file system structures.
The following C structure describes the superblock:

```
    struct fs_super {
        int32_t disk_size;         /* in 1024-byte blocks */
        int32_t blk_map_blks;      /* block map size, in blocks */
        int32_t in_map_blks;       /* inode map size, in blocks */
        int32_t inode_blks;        /* inode table len, in blocks */
        char pad[1005];            /* to make size = 1024 */
    };
```

Since inodes are 64 bytes, there are 16 of them in a 1024-byte block, and there are `16*inode_blks` inodes in total, numbered starting at zero. Note that the block and inode bitmaps must be multiples of a 1KB block, and thus may contain entries which don't correspond to valid blocks or inodes; by taking the disk size and total number of inodes into account we can ignore them.

Inodes: An inode looks like this:

```
struct fs_inode {
    int16_t uid;      /* file owner */
    int16_t gid;      /* group */
    int16_t mode;     /* type + permissions (see below) */
    int32_t mtime;    /* modification time */
    int32_t size;     /* size in bytes */
    int32_t ptrs[6];
    int32_t indir_1;
    int32_t indir_2;
    int16_t pad[9];   /* to make it 64 bytes */
};
```

"Mode": For historical reasons, Unix and Linux mash together the concept of object type (file/directory/device/symlink…) and permissions into a single 16-bit number. The result is called the file "mode", and looks like this:

```
        |<-- S_IFMT --->|           |<-- user ->|<- group ->|<- world ->|
        +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        | F | D |   |   |   |   |   | R | W | X | R | W | X | R | W | X |
        +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Since it has multiple 3-bit fields, it is commonly displayed in base 8 (octal) - e.g. permissions allowing RWX for everyone (rwxrwxrwx) are encoded as '777'. (note that in C, octal numbers are indicated by a leading "0", e.g. 0777, so the expression 0777 == 511 is true) The F and D bits correspond to octal numbers 0100000 and 040000.

Directories:
Directories are a multiple of one block in length, holding an array of directory entries:

```
    struct fs_dirent {
        uint32_t valid : 1;
        uint32_t inode : 31;
        char name[28];       /* with trailing NUL */
    };
```

Each "dirent" is 32 bytes, giving 4096/32 = 128 directory entries in each block. The directory size in the inode is always a multiple of 4096, and unused directory entries are indicated by setting the 'valid' flag to zero. The maximum name length is 27 bytes, allowing entries to always have a terminating 0 byte.

(what's that ": 1" and ": 31" thing? It combines two structure fields into a single integer - in this case, 1 bit for the valid flag and 31 bits for the inode number, both packed into a 32-bit integer)

Storage allocation:
Inodes and blocks are allocated by searching the respective bitmaps for a free entry, i.e. a bit set to 0. If bit i is in range and set to zero it can be allocated by setting it to 1. (i.e. setting the corresponding bit in memory, then writing the appropriate block back to the inode or block bitmap)