File System, Part 8: Disk blocks example

angrave edited this page on Dec 4, 2014 · 8 revisions

Edit New Page

Under construction

Please can you explain a simple model of how the file's content is stored in a simple i-node based filesystem?

Sure!To answer this question we'll build a virtual disk and then write some C code to access its contents. Our filesystem will divide the bytes available into space for inodes and a much larger space for disk blocks. Each disk block will be 4096 bytes-

```
// Disk size:
#define MAX_INODE (1024)
#define MAX_BLOCK (1024*1024)

// Each block is 4096 bytes:
typedef char[4096] block_t;

// A disk is an array of inodes and an array of disk blocks:
struct inode[MAX_INODE] inodes;
block[MAX_BLOCK] blocks;
```

Note for clarity we will not use 'unsigned' in this code example. Our fixed-sized inodes will contain the file's size in bytes, permission,user,group information, time meta-data. What is most relevant to the problem-at hand is that it will also include ten pointers to disk blocks that we will use to refer to the actual file's contents!

```
struct inode {
 int[10] directblocks; // indices for the block array i.e. where to the find the
 long size;
 // ... standard inode meta-data e.g.
 int mode, userid,groupid;
 time_t ctime,atime,mtime;
}
```

Now we can work out how to read a byte at offset `position` of our file:

```
char readbyte(inode*inode,long position) {
  if(position <0 || position >= inode->size) return -1; // invalid offset

  int  block_count = position / 4096,offset = position % 4096;

  // block count better be 0..9 !
  int physical_idx = lookup_physical_block_index(inode, block_count );
  // sanity check that the disk block index is reasonable...
  assert(physical_idx >=0 && physical_idx < MAX_BLOCK);


  // read the disk block from our virtual disk 'blocks' and return the specific b
  return blocks[physical_idx][offset];
}
```

Our initial version of lookup_physical_block is simple - we can use our table of 10 direct blocks!

```
int lookup_physical_block_index(inode*inode, int block_count) {
  assert(block_count>=0 && block_count < 10);

  return inode->directblocks[ block_count ]; // returns an index value between [0
}
```

This simple representation is reasonable provided we can represent all possible files with just ten blocks i.e. upto 40KB. What about larger files? We need the inode struct to always be the same size so just increasing the existing direct block array to 20 would roughly double the size of our inodes. If most of our files require less than 10 blocks, then our inode storage is now wasteful. To solve this problem we will use a disk block calledn the *indirect block* to extend the array of pointers at our disposal. We will only need this for files > 40KB

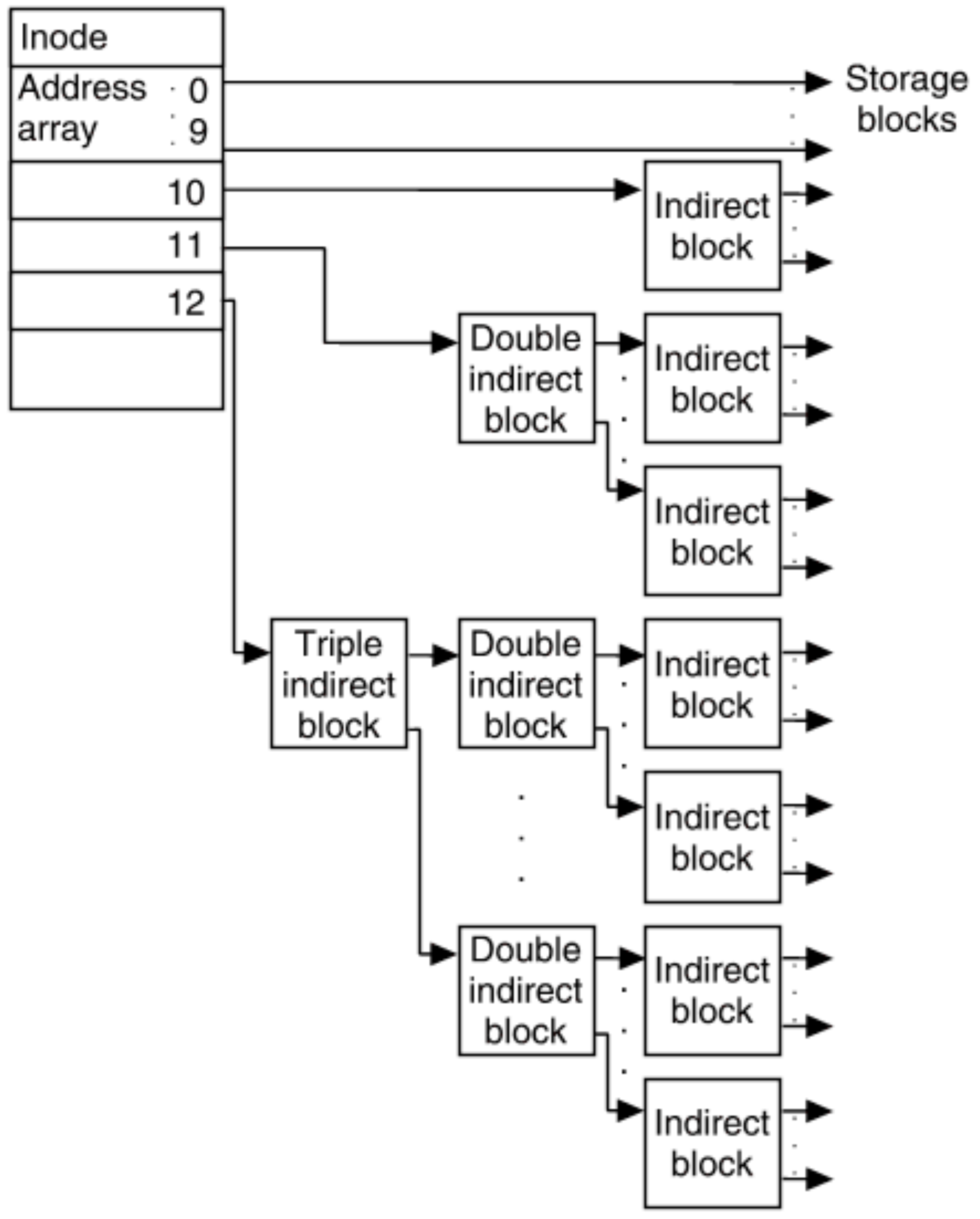
```
struct inode {
 int[10] directblocks; // if size<4KB then only the first one is valid
 int indirectblock; // valid value when size >= 40KB
 int size;
 ...
}
```

The indirect block is just a regular disk block of 4096 bytes but we will use it to hold pointers to disk blocks. Our pointers in this case are just integers, so we need to cast the pointer to an integer pointer:

```
int lookup_physical_block_index(inode*inode, int block_count) {
  assert(sizeof(int)==4); // Warning this code assumes an index is 4 bytes!
  assert(block_count>=0 && block_count < 1024 + 10); // 0 <= block_count< 1034

  if( block_count < 10)
     return inode->directblocks[ block_count ];

  // read the indirect block from disk:
  block_t* oneblock = & blocks[ inode->indirectblock ];

  // Treat the 4KB as an array of 1024 pointers to other disk blocks
  int* table = (int*) oneblock;

 // Look up the correct entry in the table
 // Offset by 10 because the first 10 blocks of data are already
 // accounted for
  return table[ block_count - 10 ];
}
```

For a typical filesystem our index values are 32 bits i.e. 4bytes. Thus in 4096 bytes we can store 4096 / 4 = 1024 entries This means our indirect block can refer to 1024 * 4KB = 4MB of data. With the first ten direct blocks we can therefore accommodate files up to 40KB + 1024 * 4KB= 4136KB . Some of the later table entries can be invalid for files that are smaller than this.

For even larger files we could use two indirect blocks. However there's a better alternative, that will allow us to efficiently scale up to huge files. We will include a double-indirect pointer and if that's not enough a triple indirect pointer. The double indirect pointer means we have a table of 1024 entries to disk blocks that are used as 1024 entries. This means we can refer to 1024*1024 disk blocks of data.



(source: http://uw714doc.sco.com/en/FS_admin/graphics/s5chain.gif)

```
int lookup_physical_block_index(inode*inode, int block_count) {
  if( block_count < 10)
     return inode->directblocks[ block_count ];

  // Use indirect block for the next 1024 blocks:
  // Assumes 1024 ints can fit inside each block!
  if( block_count < 1024 + 10) {
      int* table = (int*) & blocks[ inode->indirectblock ];
      return table[ block_count - 10 ];
  }
  // For huge files we will use a table of tables
  int i = (block_count - 1034) / 1024 , j = (block_count - 1034) % 1024;
  assert(i<1024); // triple-indirect is not implemented here!

  int* table1 = (int*) & blocks[ inode->doubleindirectblock ];
   // The first table tells us where to read the second table ...
  int* table2 = (int*) & blocks[   table1[i]   ];
  return table2[j];
```

```
    // For gigantic files we will need to implement triple-indirect (table of tabl
}
```

Notice that reading a byte using double indirect requires 3 disk block reads (two tables and the actual data block).