Home

Extra credit 2: Expandable inodes for xv6

In this assignment you'll increase the maximum size of an xv6 file. Currently xv6 files are limited to 140 sectors, or 71,680 bytes. This limit comes from the fact that an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 128 more block numbers, for a total of 12+128=140.

In this homework you will change the xv6 file system code to support a "expandable inode" file addressing for files of infinite length (of course, in practice the file size will be limited by the size of your file system).

Preliminaries

Modify your Makefile's `CPUS` definition so that it reads:

```
CPUS := 1
```

This speeds up qemu when xv6 creates large files.

Since `mkfs` initializes the file system to have fewer than 1000 free data blocks, too few to show off the changes you'll make. Modify FSSIZE inside `param.h` to read 128*128*16 blocks (or 128MB)

```
#define FSSIZE       128*128*16
```

Finally, to make your code compile in `fs.h` change MAXFILE to

```
#define MAXFILE 128*2048
```

While we will support files of infinite length, in practice xv6 too much depends on the MAXFILE constant. So to make the build system happy, lets change it to some large value.

Download big.c into your xv6 directory, add them to the UPROGS list (same as you did for HW4), start up xv6, and run `big`. `big` writes 4 files: 1MB, 2MB, 4MB, and 8MB, and then reads them --- this will help you to test your solution. Big writes 64KB at a time, which is significantly faster (and yet it takes quite a bit of time). Feel free to edit `big.c` to adjust write size, and enable additional debug info.

Your Job

Modify bmap() so that it implements an expandable inode system. Initially your file should have only 1 direct block, i.e., the only element of ip->addrs[] (ip->addrs[0]) should point to the direct block of the file data. However the moment you add the second block your inode "expands", i.e., the 1 direct pointer turns into the pointer to an indirect block that can have 128 pointers to the data blocks. Again, nothing changes for a while until your file reaches the size of 128 blocks, and then your inode expands to have yet another layer of indirect blocks. See the examples below.

An ASCII example for 0-layer

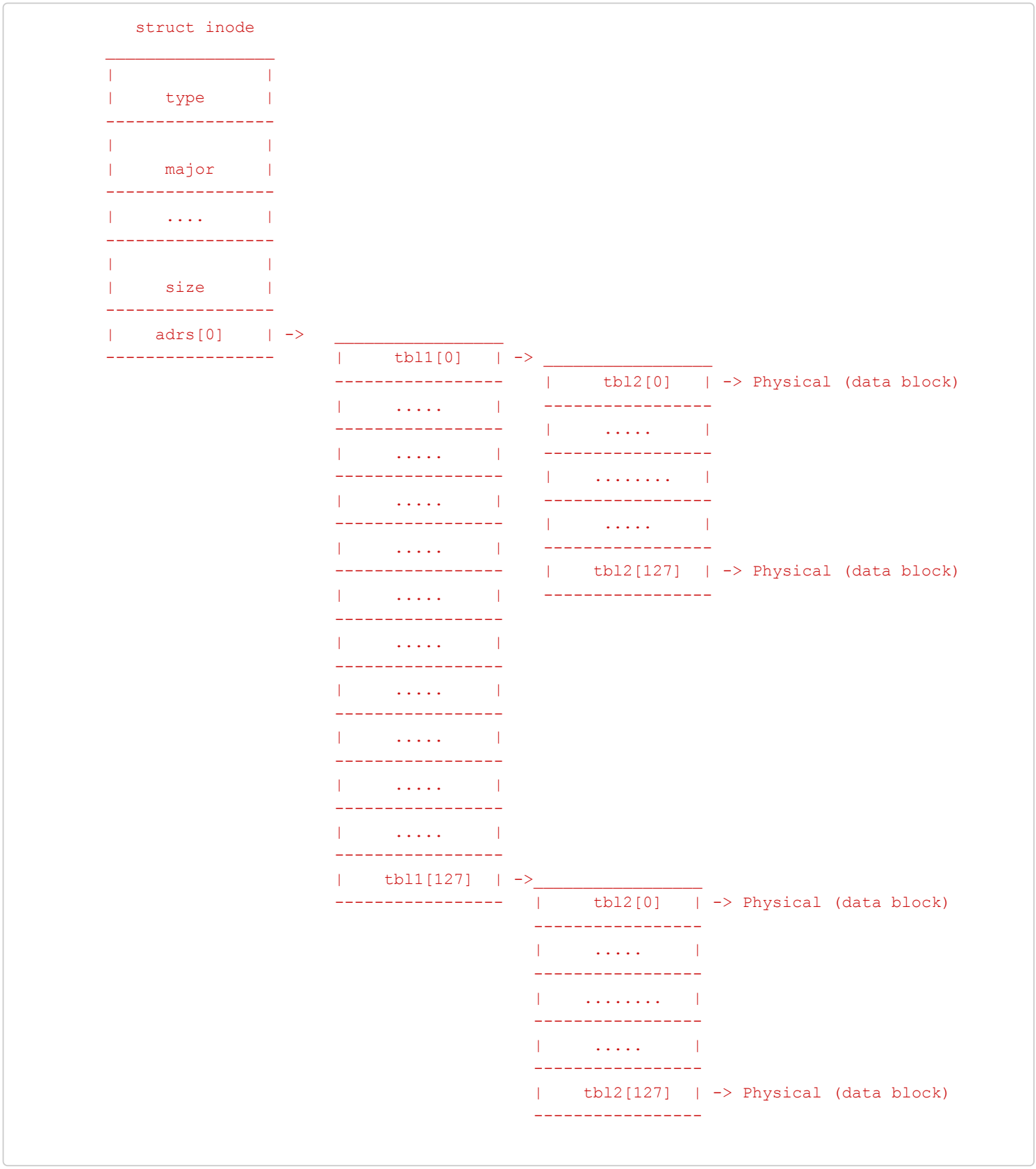
```
            struct inode
          ________________
```

```
          |                |
          |     type       |
          ------------------
          |                |
          |     major      |
          ------------------
          |     ....       |
          ------------------
          |                |
          |     size       |
          ------------------
          |    adrs[0]     | -> Physical (data block)
          ------------------
```

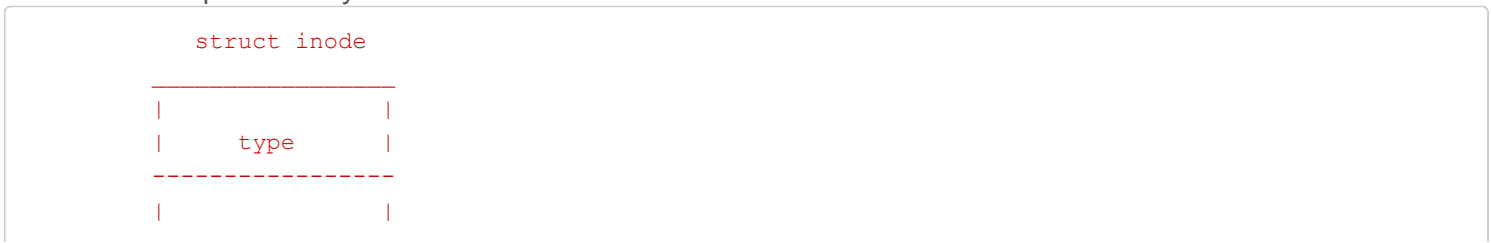
An ASCII example for 1-layer

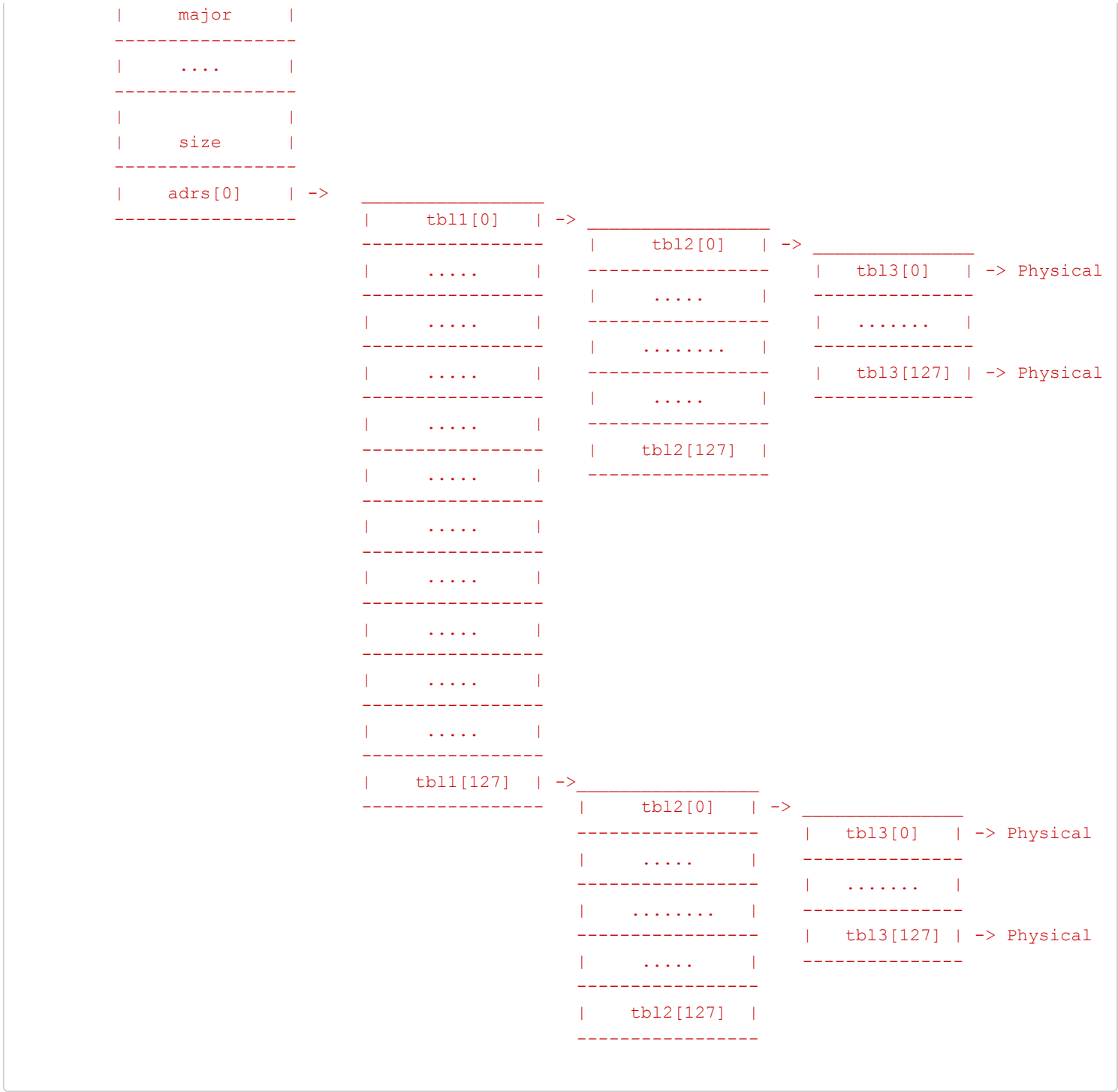
```
             struct inode
          ___________________
          |                |
          |     type       |
          ------------------
          |                |
          |     major      |
          ------------------
          |     ....       |
          ------------------
          |                |
          |     size       |
          ------------------
          |    adrs[0]     | ->   ___________________
          ------------------      |     tbl1[0]     | -> Physical (data block)
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |     .....       |
                                  -------------------
                                  |    tbl1[127]    | -> Physical (data block)
                                  -------------------
```

An ASCII example for 2-layers



An ASCII example for 3-layers





And so on... the system should support infinite files.

After you have modified `bmap()` you need to go into `mkfs.c` and modify `iappend()`. This c file gets run during the make process and builds the file system image which gets run with QEMU. The code and logic will be extremely similar to the code you had written in bmap().

If you have a hard time understanding what iappend() is doing, here is the annotated code

```
void
iappend(uint inum, void *xp, int n)
{
  char *p = (char*)xp;
  uint fbn, off, n1;
  struct dinode din;
```

```
  char buf[BSIZE];
  uint indirect[NINDIRECT];
  uint x;

  // Read inode number inum into &din
  rinode(inum, &din);
  // inode might already have some data, offset is the last byte it has
  off = xint(din.size);
  // While we have bytes to write into the inode, loop
  while(n > 0){
    // Get the block number of the last block from the offset
    fbn = off / BSIZE;
    assert(fbn < MAXFILE);
    // if block number is still inside direct blocks ...
    if(fbn < NDIRECT){
      // is block allocated?
      if(xint(din.addrs[fbn]) == 0){
        // no allocate it by incrementing freeblock pointer
        din.addrs[fbn] = xint(freeblock++);
      }
      // allocated ... get the block
      x = xint(din.addrs[fbn]);
    } else {
      // oh no... it's an indirect block
      if(xint(din.addrs[NDIRECT]) == 0){
        // was first level of indirection allocated?
        din.addrs[NDIRECT] = xint(freeblock++);
      }
      // read the sector that contains the 1 level indirect table
      // into indirect
      rsect(xint(din.addrs[NDIRECT]), (char*)indirect);
      // check if the entry already allocated in the table
      if(indirect[fbn - NDIRECT] == 0){
        // not allocated, allocate a new block and update
        // the first level indirect table
        indirect[fbn - NDIRECT] = xint(freeblock++);
        // write first level table back to disk
        wsect(xint(din.addrs[NDIRECT]), (char*)indirect);
      }
      // get the sector number
      x = xint(indirect[fbn-NDIRECT]);
    }
    n1 = min(n, (fbn + 1) * BSIZE - off);
    // read sector
    rsect(x, buf);
    //copy data into buf
    bcopy(p, buf + off - (fbn * BSIZE), n1);
    // write back the sector
    wsect(x, buf);
    n -= n1;
    off += n1;
    p += n1;
  }
  din.size = xint(off);
  // write back the inode
  winode(inum, &din);
```

```
}
```

You don't have to modify xv6 to handle deletion of files with linked-list blocks.

What to Look At

The format of an on-disk inode is defined by `struct dinode` in `fs.h`. You're particularly interested in `NDIRECT`, and the `addrs[]` element of `struct dinode`. Look here for a diagram of the standard xv6 inode.

The code that finds a file's data on disk is in `bmap()` in `fs.c`. Have a look at it and make sure you understand what it's doing. `bmap()` is called both when reading and writing a file. When writing, `bmap()` allocates new blocks as needed to hold file content, as well as allocating an indirect block if needed to hold block addresses.

`bmap()` deals with two kinds of block numbers. The `bn` argument is a "logical block" -- a block number relative to the start of the file. The block numbers in `ip->addrs[]`, and the argument to `bread()`, are disk block numbers. You can view `bmap()` as mapping a file's logical block numbers into disk block numbers.

Hints

Make sure you understand `bmap()`. Write out a diagram of the relationships between `ip->addrs[]`, the direct blocks, the list blocks. Make sure you understand why adding linked list addressing block allows you to have files of infinite size.

If your file system gets into a bad state, perhaps by crashing, you may need to delete `fs.img` (do this from Unix, not xv6).

Don't forget to `brelse()` each block that you `bread()`.

You should allocate linked list blocks as needed, like the original `bmap()`.

Our solution was done in around 50 lines of code.

Submit your solution through Gradescope Gradescope CS143A Operating Systems as a compressed tar file of your xv6 source tree (after running make clean). You can use the following command to create a compressed tar file.

```
openlab$ cd xv6-public
openlab$ make clean
openlab$ cd ..
openlab$ tar -czvf extra2.tgz xv6-public
```

Updated: December, 2020