Operating Systems
CSCI 5806

Spring Semester 2025 — CRN 22968

Term Project — Step 5 — File Access

Target completion date: Friday, April 4, 2025

Goals

- Provide functions to provide access to data within a file, given the file's inode.

Details

In this step, we provide access to a file's data, enabling read and write access to a file. Note that this includes directories, which are just files with a bit set in the file's inode to indicate it is a directory and not a data file.

Block-level file access

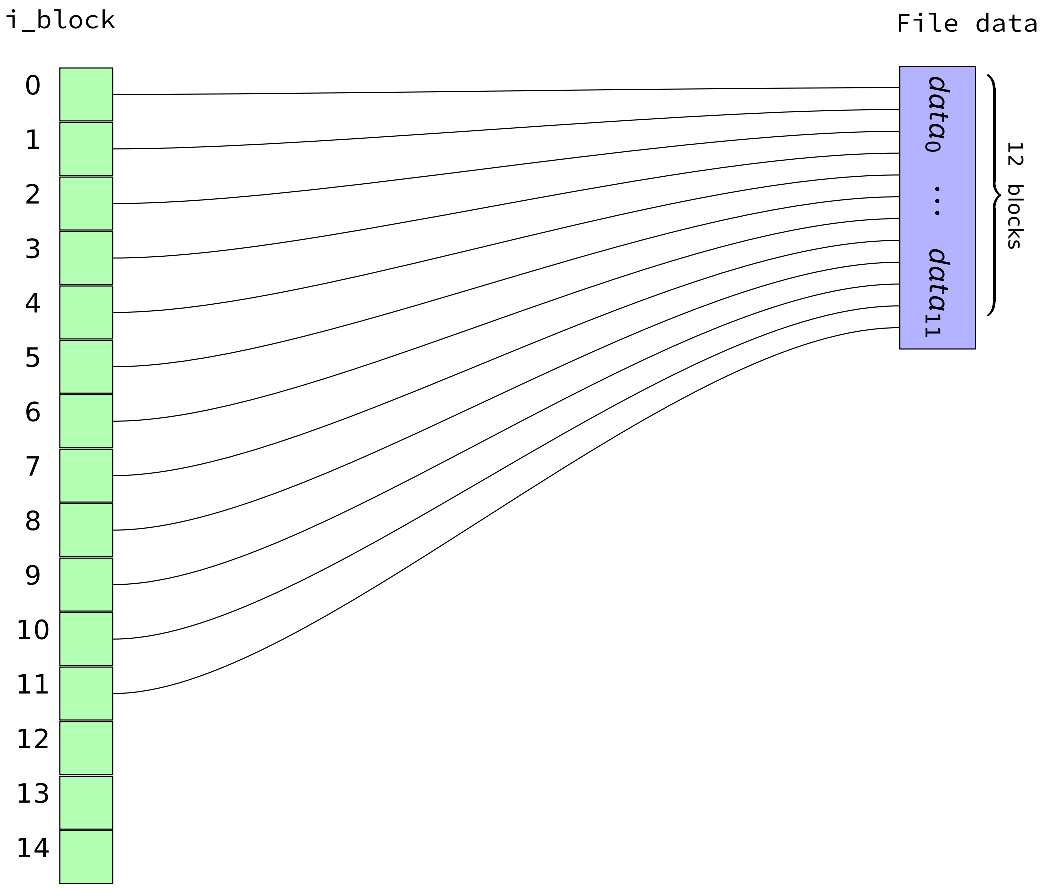
At the core of the file access functions is the ability to read or write one entire block in the file. This isn't as simple as using **`fetchBlock()`** or **`writeBlock()`**; the blocks are numbered sequentially, but the numbers reflect a position *within the file*, not their position in the filesystem. In other words, block 0 is the first block of the file's data, block 1 is the second block of file data, and so on.

To provide block-level access, we need two functions:

- **`int32_t fetchBlockFromFile(struct Inode *i,uint32_t bNum, void *buf)`**
  Read block **`bNum`** from the file, placing the data in the given buffer.
- **`int32_t writeBlockToFile(struct Inode *i,uint32_t bNum, void *buf)`**
  Write the given buffer into block **`bNum`** in the file. This may involve allocating one or more data blocks.

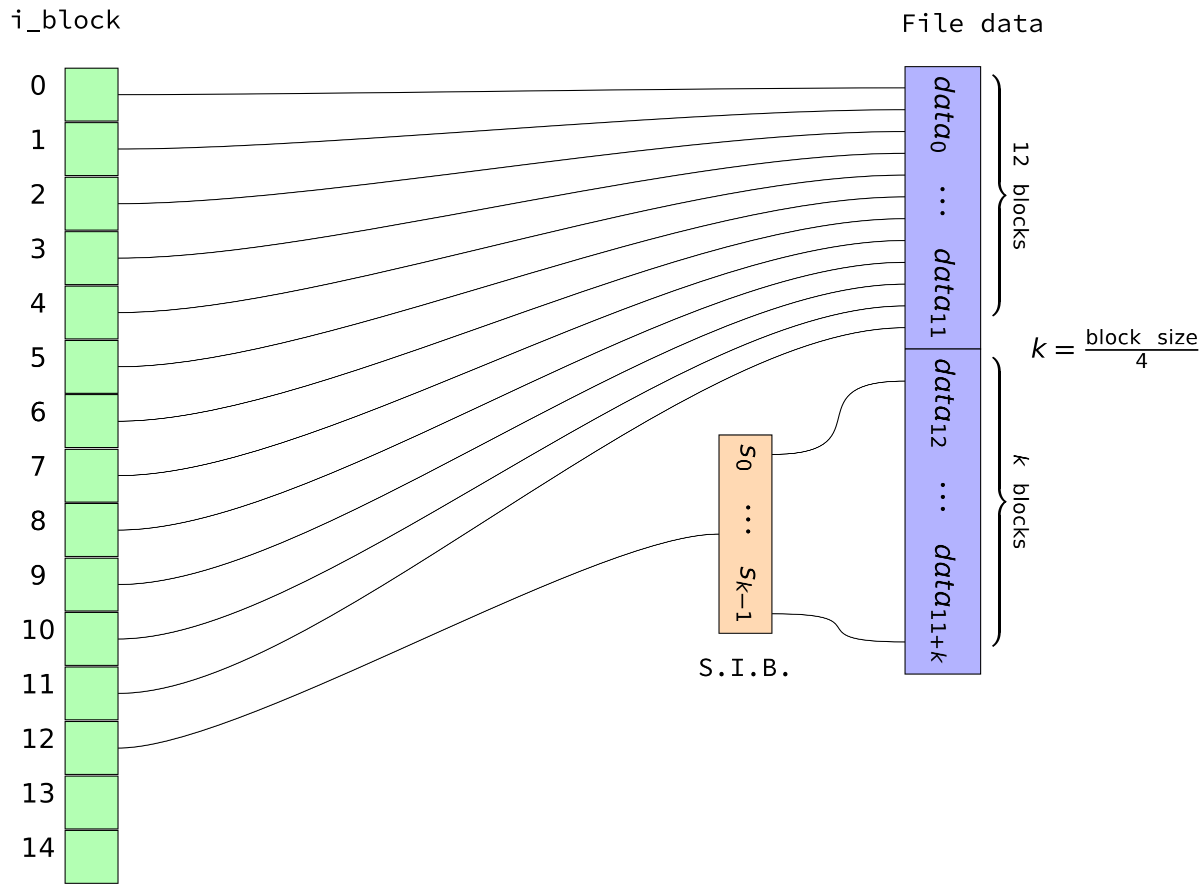
We must also understand how data is organized in a UNIX file. This begins with the **`i_block`** field in an inode. This is an array of `15` block numbers.

The first `12` entries of **`i_block`** contain indexes for the file's first twelve data blocks:



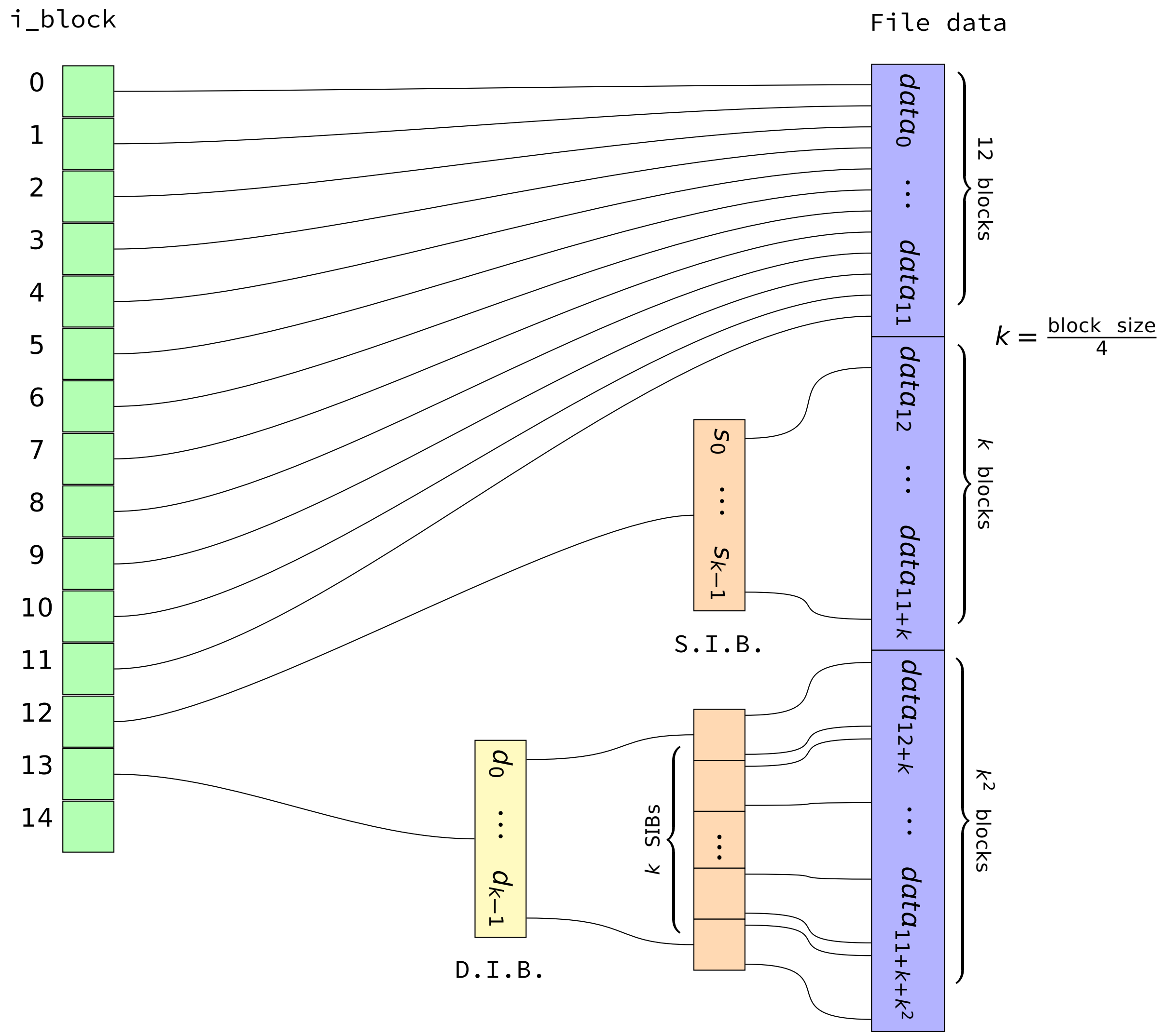
For a 1KB block system, that covers the first 12KB, which is about the average file size on a UNIX system.

The next entry in **`i_block`** — slot 12 — doesn't have the block number for the next data block. It tells you where to find a *single indirect block*, which is a data block that's been sliced into 4-byte chunks, each holding the block number of the next k data blocks, where k is the block size divided by 4. Using a 1KB system as an example, $k = 256$.



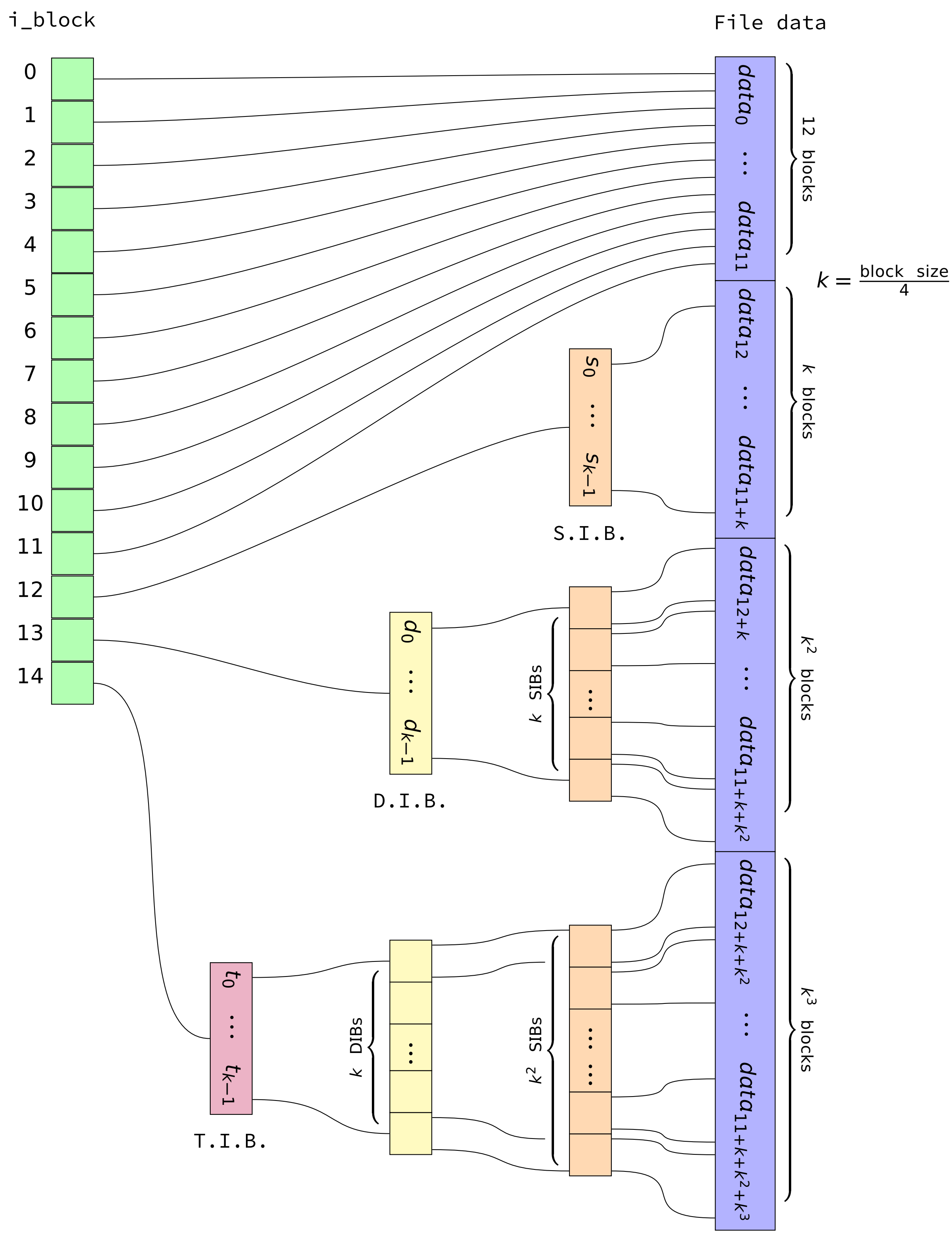
The single indirect block is itself an array of k block numbers, each holding the location of the next data block. Between the 12 direct blocks in the **i_block** array and the single indirect block, we can access the first $12 + k$ blocks of file data. In a 1KB system, that provides access to the first 268KB of data.

If we need to access more data, slot 13 in the **i_block** array holds the index of a *double indirect block*, which contains the indexes of k single indirect blocks, each of which holds k indexes of data blocks, giving access to k^2 additional data blocks. In a 1KB file system, this yields an additional 65 536KB of data.



If more space is needed — most of you have worked with files larger than 66MB — there is still one more level of indirection available. The final slot in **i_block** holds the index of a *triple indirect block*. That block holds the indexes of k double indirect blocks, which each hold the indexes of k single indirect blocks, which each hold the indexes of k data blocks. This provides access to a final k^3 data blocks; with a 1KB block size, this provides access to an additional 16 777 216KB of data.

If more space is needed, a larger block size is used; there is no further level of indirection.



This looks intimidating, but it is actually rather easy to navigate, and quite efficient — any byte can be accessed in at most five disk accesses (inode, triple, double, single, data blocks).

Here are some observations about the file structure:

- The index of every data block is held in an array. That array might be the **`i_block`** array or it might be an array held in a single indirect block.
- The same can be said about the indexes of the single, double and triple indirect blocks.
- The trees headed by the single, double and triple indirect blocks all have a regular structure, with each node having exactly k children, where k is the block size divided by 4.
- Every single indirect block accesses k data blocks; every double indirect block accesses k single indirect blocks and k^2 data blocks.

The approach to take is to first determine which tree, if any, we need to descend to find our data block, and determine which block number *within that tree* we want. Then, we can descend the tree in a simple, methodical way.

Algorithm 1 Fetching a block from a file, part 1 of 2

```
k ← block size / 4
if b < 12 then                              ▷ index is in the i_block array
    blockList ← i_block                     ▷ Set up the array to read from

    goto direct
else if b < 12 + k then                     ▷ index is in first single indirect block
    if i_block[12] = 0 then
        return false
    end if
    FetchBlock(i_block[12], buf)            ▷ fetch SIB

    blockList ← buf                         ▷ Set up the array to read from
    b ← b − 12                              ▷ adjust b for nodes skipped over

    goto direct
else if b < 12 + k + k^2 then               ▷ index is under first double indirect block
    if i_block[13] = 0 then
        return false
    end if
    FetchBlock(i_block[13], buf)            ▷ fetch DIB

    blockList ← buf                         ▷ Set up the array to read from
    b ← b − 12 − k                          ▷ adjust b for nodes skipped over

    goto single
else                                        ▷ index is under triple indirect block
    if i_block[14] = 0 then
        return false
    end if
    FetchBlock(i_block[14], buf)            ▷ fetch TIB

    blockList ← buf                         ▷ Set up the array to read from
    b ← b − 12 − k − k^2                    ▷ adjust b for nodes skipped over
end if
```

Algorithm 2 Fetching a block from a file, part 2 of 2

$index \leftarrow b/(k^2)$ ▷ Determine which DIB to fetch
$b \leftarrow b \bmod (k^2)$ ▷ Determine which block under that DIB we want

if $blockList[index] = 0$ **then**
    return false
end if
FetchBlock($blockList[index], buf$) ▷ Fetch the DIB and point to it
$blockList \leftarrow buf$

single: ▷ Given a DIB, fetch proper SIB

$index \leftarrow b/k$ ▷ Determine which SIB to fetch
$b \leftarrow b \bmod k$ ▷ Determine which block under that SIB we want

if $blockList[index] = 0$ **then**
    return false
end if
FetchBlock($blockList[index], buf$) ▷ Fetch the SIB and point to it
$blockList \leftarrow buf$

direct: ▷ Given an array of data block indexes, fetch block
▷ Array can be SIB or i_block

if $blockList[b] = 0$ **then**
    return false
end if
FetchBlock($blockList[b], buf$) ▷ Fetch the data block
return true

The two parts, taken together, form a subroutine for fetching block *b* from a file. It returns true if the read succeeds, false if it fails, which it would if the block hasn't been allocated.

Writing to a block is only slightly more complicated. The additional complexity is due to allocation of blocks when necessary, including indirect blocks, and determining which additional blocks need to be written due to updating indexes after allocation. However, it does follow the general pattern of fetching a block.

When reading, indirect blocks can be read into the same buffer than eventually holds the data, since the data read is the last fetch. However, when writing, a second block-sized temporary buffer is needed to hold indirect blocks. Since the data block is written as the last I/O operation in the writing process, its buffer can't be used to hold indirect blocks.

The ALLOCATE**()** function allocates an unused block and returns the block number. It handles marking the block as used and updating the counts in the superblock and group descriptor table and updates those structures and the block bitmap on disk.

If fetching a block returns false, the buffer should be set to all zeroes.

If the data block has to be allocated, you need to adjust the **`i_blocks`** field (not the same as the array **`i_block`**) in the inode; this counts the number of `512`-byte chunks used by the file's data.

Algorithm 3 Writing a block to a file, part 1 of 2

```
k ← block size /4
if b < 12 then                                   ▷ index is in the i_block array
    if i_block[b] == 0 then                      ▷ If block not there, allocate it
        i_block[b] ← ALLOCATE()
        WRITEINODE(iNum, iNode)
    end if
    blockList ← i_block                          ▷ Set up the array to read from

    goto direct
else if b < 12 + k then                          ▷ index is in first single indirect block
    if i_block[12] == 0 then                     ▷ If block not there, allocate it
        i_block[12] ← ALLOCATE()
        WRITEINODE(iNum, iNode)
    end if
    FETCHBLOCK(i_block[12], tmp)                 ▷ fetch SIB

    ibNum ← i_block[12]
    blockList ← tmp                              ▷ Set up the array to read from
    b ← b − 12                                   ▷ adjust b for nodes skipped over

    goto direct
else if b < 12 + k + k^2 then                    ▷ index is under first double indirect block
    if i_block[13] = 0 then
        i_block[13] ← ALLOCATE()
        WRITEINODE(iNum, iNode)
    end if
    FETCHBLOCK(i_block[13], tmp)                 ▷ fetch DIB

    ibNum ← i_block[13]
    blockList ← tmp                              ▷ Set up the array to read from
    b ← b − 12 − k                               ▷ adjust b for nodes skipped over

    goto single
else                                             ▷ index is under triple indirect block
    if i_block[14] = 0 then
        i_block[14] ← ALLOCATE()
        WRITEINODE(iNum, iNode)
    end if
    FETCHBLOCK(i_block[14], tmp)                 ▷ fetch TIB

    ibNum ← i_block[14]
    blockList ← tmp                              ▷ Set up the array to read from
    b ← b − 12 − k − k^2                         ▷ adjust b for nodes skipped over
end if
```

Algorithm 4 Fetching a block from a file, part 2 of 2

```
index ← b/(k^2)                                   ▷ Determine which DIB to fetch
b ← b mod (k^2)                                   ▷ Determine which block under that DIB we want

if blockList[index] = 0 then
    blockList[index] ← ALLOCATE()
    WRITEBLOCK(ibNum, blockList)
end if
ibNum ← blockList[index]
FETCHBLOCK(blockList[index], tmp)                 ▷ Fetch the DIB and point to it
blockList ← tmp

single:                                           ▷ Given a DIB, fetch proper SIB

index ← b/k                                       ▷ Determine which SIB to fetch
b ← b mod k                                       ▷ Determine which block under that SIB we want

if blockList[index] = 0 then
    blockList[index] ← ALLOCATE()
    WRITEBLOCK(ibNum, blockList)
end if
ibNum ← blockList[index]
FETCHBLOCK(blockList[index], tmp)                 ▷ Fetch the SIB and point to it
blockList ← tmp

direct:                                           ▷ Given an array of data block indexes, write block
                                                  ▷ Array can be SIB or i_block

if blockList[b] = 0 then
    blockList[b] ← ALLOCATE()
    WRITEBLOCK(ibNum, blockList)
end if
WRITEBLOCK(blockList[b], buf)                     ▷ Write the data block
```

That's all you'll need for the project. If you wish to generalize the code a little more, you can write the five file functions — open, close, read, write and seek — to work at this level as well.

Example

This is the output from my step 5 program, on the fixed VDI file with 1KB blocks. It shows the root directory — inode 2 — and the file system's **`lost+found`** directory — inode 11 — in readable form. It also shows the contents of the data for each "file."

As a bonus, inode 12 — corresponding to the Arduino tarball — is included; copying that is the big test of file copying, as it requires access to all levels of indirection, using the 1KB file.

```
Inode 2:
Offset: 0x0
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
  +------------------------------------------------+  +----------------+
00|ed 41 e8 03 00 04 00 00 5f e7 a9 58 a8 bf ba 56|00| A       _  X   V|
10|a8 bf ba 56 00 00 00 00 e8 03 04 00 02 00 00 00|10|   V            |
20|00 00 00 00 03 00 00 00 03 02 00 00 00 00 00 00|20|                |
30|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|30|                |
40|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|40|                |
50|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|50|                |
60|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|60|                |
70|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|70|                |
80|                                                |80|                |
90|                                                |90|                |
a0|                                                |a0|                |
b0|                                                |b0|                |
c0|                                                |c0|                |
d0|                                                |d0|                |
e0|                                                |e0|                |
f0|                                                |f0|                |
  +------------------------------------------------+  +----------------+

                 Mode: 40755 -d-----rwxr-xr-x
                 Size: 1024
               Blocks: 2
            UID / GID: 1000 / 1000
                Links: 4
              Created: Tue Feb  9 23:42:16 2016
          Last access: Sun Feb 19 13:43:43 2017
    Last modification: Tue Feb  9 23:42:16 2016
              Deleted: Wed Dec 31 19:00:00 1969
                Flags: 00000000
         File version: 0
            ACL block: 0
        Direct blocks:
                  0-3:          515           0           0           0
                  4-7:            0           0           0           0
                 8-11:            0           0           0           0
Single indirect block: 0
Double indirect block: 0
Triple indirect block: 0

Offset: 0x0
```

```
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
  +-----------------------------------------------+  +----------------+
00|02 00 00 00 0c 00 01 02 2e 00 00 00 02 00 00 00|00|        .       |
10|0c 00 02 02 2e 2e 00 00 0b 00 00 00 14 00 0a 02|10|    ..          |
20|6c 6f 73 74 2b 66 6f 75 6e 64 00 00 0c 00 00 00|20|lost+found      |
30|24 00 1c 01 61 72 64 75 69 6e 6f 2d 31 2e 36 2e|30|$   arduino-1.6.|
40|37 2d 6c 69 6e 75 78 36 34 2e 74 61 72 2e 78 7a|40|7-linux64.tar.xz|
50|11 77 00 00 10 00 08 02 65 78 61 6d 70 6c 65 73|50| w      examples|
60|18 00 00 00 a0 03 0f 01 61 72 64 75 69 6e 6f 2d|60|        arduino-|
70|62 75 69 6c 64 65 72 00 00 00 00 00 00 00 00 00|70|builder         |
80|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|80|                |
90|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|90|                |
a0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|a0|                |
b0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|b0|                |
c0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|c0|                |
d0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|d0|                |
e0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|e0|                |
f0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|f0|                |
  +-----------------------------------------------+  +----------------+

(intervening blocks are all zeroes and not shown here)

Offset: 0x300
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
  +-----------------------------------------------+  +----------------+
00|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|00|                |
10|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|10|                |
20|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|20|                |
30|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|30|                |
40|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|40|                |
50|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|50|                |
60|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|60|                |
70|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|70|                |
80|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|80|                |
90|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|90|                |
a0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|a0|                |
b0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|b0|                |
c0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|c0|                |
d0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|d0|                |
e0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|e0|                |
f0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|f0|                |
  +-----------------------------------------------+  +----------------+

Inode 11:
Offset: 0x0
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
  +-----------------------------------------------+  +----------------+
00|c0 41 00 00 00 30 00 00 88 bb ba 56 88 bb ba 56|00| A   0      V   V|
10|88 bb ba 56 00 00 00 00 00 00 02 00 18 00 00 00|10|   V            |
20|00 00 00 00 00 00 00 00 04 02 00 00 05 02 00 00|20|                |
30|06 02 00 00 07 02 00 00 08 02 00 00 09 02 00 00|30|                |
40|0a 02 00 00 0b 02 00 00 0c 02 00 00 0d 02 00 00|40|                |
50|0e 02 00 00 0f 02 00 00 00 00 00 00 00 00 00 00|50|                |
60|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|60|                |
```

```
70|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|70|                |
80|                                               |80|                |
90|                                               |90|                |
a0|                                               |a0|                |
b0|                                               |b0|                |
c0|                                               |c0|                |
d0|                                               |d0|                |
e0|                                               |e0|                |
f0|                                               |f0|                |
  +-----------------------------------------------+  +----------------+

                 Mode: 40700 -d-----rwx------
                 Size: 12288
               Blocks: 24
            UID / GID: 0 / 0
                Links: 2
              Created: Tue Feb  9 23:24:40 2016
          Last access: Tue Feb  9 23:24:40 2016
    Last modification: Tue Feb  9 23:24:40 2016
              Deleted: Wed Dec 31 19:00:00 1969
                Flags: 00000000
         File version: 0
            ACL block: 0
        Direct blocks:
                  0-3:          516          517          518          519
                  4-7:          520          521          522          523
                 8-11:          524          525          526          527
Single indirect block: 0
Double indirect block: 0
Triple indirect block: 0

Offset: 0x0
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
  +-----------------------------------------------+  +----------------+
00|0b 00 00 00 0c 00 01 02 2e 00 00 00 02 00 00 00|00|        .       |
10|f4 03 02 02 2e 2e 00 00 00 00 00 00 00 00 00 00|10|    ..          |
20|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|20|                |
30|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|30|                |
40|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|40|                |
50|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|50|                |
60|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|60|                |
70|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|70|                |
80|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|80|                |
90|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|90|                |
a0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|a0|                |
b0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|b0|                |
c0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|c0|                |
d0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|d0|                |
e0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|e0|                |
f0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|f0|                |
  +-----------------------------------------------+  +----------------+

(intervening blocks are all zeroes and not shown here)
```

```
Offset: 0x2f00
  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
  +-----------------------------------------------+  +----------------+
00|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|00|                |
10|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|10|                |
20|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|20|                |
30|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|30|                |
40|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|40|                |
50|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|50|                |
60|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|60|                |
70|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|70|                |
80|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|80|                |
90|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|90|                |
a0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|a0|                |
b0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|b0|                |
c0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|c0|                |
d0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|d0|                |
e0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|e0|                |
f0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|f0|                |
  +-----------------------------------------------+  +----------------+

Inode 12:
Offset: 0x0
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
  +-----------------------------------------------+  +----------------+
00|b4 81 e8 03 84 ed af 05 49 bf ba 56 49 bf ba 56|00|        I   VI  V|
10|85 be ba 56 00 00 00 00 e8 03 01 00 d6 da 02 00|10|   V            |
20|00 00 00 00 01 00 00 00 01 04 00 00 02 04 00 00|20|                |
30|03 04 00 00 04 04 00 00 05 04 00 00 06 04 00 00|30|                |
40|07 04 00 00 08 04 00 00 09 04 00 00 0a 04 00 00|40|                |
50|0b 04 00 00 0c 04 00 00 11 02 00 00 12 02 00 00|50|                |
60|b3 07 00 00 ad 7b 45 5c 00 00 00 00 00 00 00 00|60|     {E\        |
70|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|70|                |
80|                                               |80|                |
90|                                               |90|                |
a0|                                               |a0|                |
b0|                                               |b0|                |
c0|                                               |c0|                |
d0|                                               |d0|                |
e0|                                               |e0|                |
f0|                                               |f0|                |
  +-----------------------------------------------+  +----------------+

                 Mode: 100664 f------rw-rw-r--
                 Size: 95415684
               Blocks: 187094
            UID / GID: 1000 / 1000
                Links: 1
              Created: Tue Feb  9 23:40:41 2016
          Last access: Tue Feb  9 23:40:41 2016
    Last modification: Tue Feb  9 23:37:25 2016
              Deleted: Wed Dec 31 19:00:00 1969
                Flags: 00000000
         File version: 1548057517
```

```
           ACL block: 0
       Direct blocks:
                0-3:         1025        1026        1027        1028
                4-7:         1029        1030        1031        1032
               8-11:         1033        1034        1035        1036
Single indirect block: 529
Double indirect block: 530
Triple indirect block: 1971
```