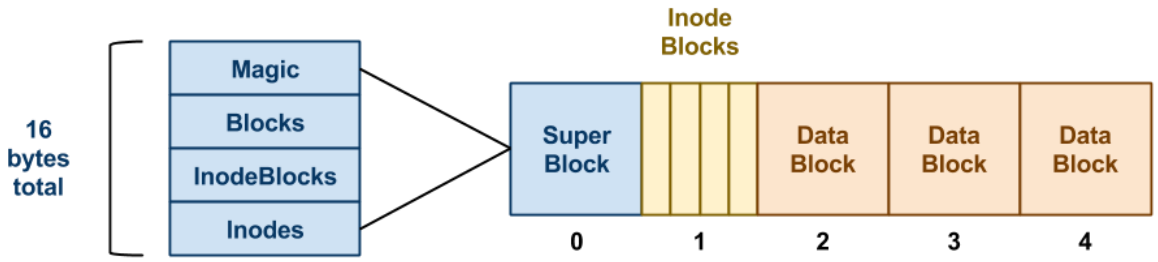
Simple File System Design

We will build SimpleFS file system that looks much like the Unix file system. Each "file" is identified by an integer called an inode number. The inode number is simply an index into the array of inode structures that starts in block 1. When a file is created, SimpleFS chooses the first available inode and returns the corresponding inode number to the user.

To implement the file system component, you will first need to understand the SimpleFS disk layout. We will assume that each disk block is 4KB in size. The first block of the disk is the superblock that describes the layout of the rest of the filesystem.

A certain number of blocks following the superblock contain “inode” data structures. Typically, 10% of the total number of disk blocks are used as inode blocks.

The remaining blocks in the filesystem are used as plain data blocks, and occasionally as indirect pointer blocks as shown in the example below:

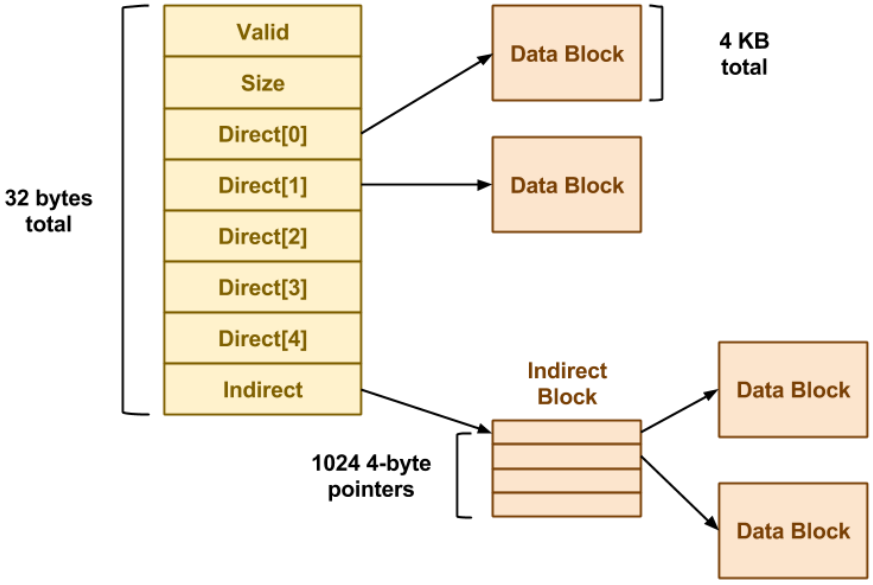


In this example, we have a **SimpleFS** disk image that begins with a **superblock**. This **superblock** consists of four fields:

1. **Magic**: The first field is always the `MAGIC_NUMBER` or `0xC0DEBABE`. The format routine places this number into the very first bytes of the **superblock** as a sort of filesystem "signature". When the filesystem is mounted, the OS looks for this magic number. If it is correct, then the disk is assumed to contain a valid filesystem. If some other number is present, then the mount fails, perhaps because the disk is not formatted or contains some other kind of data.
2. **Blocks**: The second field is the total number of blocks, which should be the same as the number of blocks on the disk.
3. **InodeBlocks**: The third field is the number of blocks set aside for storing **inodes**. The format routine is responsible for choosing this value, which should always be `10%` of the **Blocks**, rounding up.
4. **Inodes**: The fourth field is the total number of **inodes** in those **inode blocks**.

Note that the **superblock** data structure is quite small: only `16` bytes. The remainder of disk block zero is left unused.

Each **inode** in **SimpleFS** looks like the file:



Each field of the inode is a 4-byte (32-bit) integer. The Valid field is 1 if the inode is valid (i.e. has been created) and is 0 otherwise. The Size field contains the logical size of the inode data in bytes. There are 5 direct pointers to data blocks, and one pointer to an indirect data block. In this context, "pointer" simply means the number of a block where data may be found. A value of 0 may be used to indicate a null block pointer. Each inode occupies 32 bytes, so there are 128 inodes in each 4KB inode block.

Note that an indirect data block is just a big array of pointers to further data blocks. Each pointer is a 4-byte int, and each block is 4KB, so there are 1024 pointers per block. The data blocks are simply 4KB of raw data.

One thing missing in SimpleFS is the free block bitmap. As discussed in class, a real filesystem would keep a free block bitmap on disk, recording one bit for each block that was available or in use. This bitmap would be consulted and updated every time the filesystem needed to add or remove a data block from an inode.

Because SimpleFS does not store this on-disk, you are required to keep a free block bitmap in memory. That is, there must be an array of booleans, one for each block of the disk, noting whether the block is in use or available. When it is necessary to allocate a new block for a file, the system must scan through the array to locate an available block. When a block is freed, it likewise must be marked in the bitmap.

Suppose that the user makes some changes to a SimpleFS filesystem, and then reboots the system (ie. restarts the shell). Without a free block bitmap, SimpleFS cannot tell which blocks are in use and which are free. Fortunately, this information can be recovered by scanning the disk. Each time that an SimpleFS filesystem is mounted, the system must build a new free block bitmap from scratch by scanning through all of the inodes and recording which blocks are in use.

SimpleFS looks much like the Unix file system. Each "file" is identified by an integer called an inode number. The inode number is simply an index into the array of inode structures that starts in block 1. When a file is created, SimpleFS chooses the first available inode and returns the corresponding inode number to the user.