COS 316
Naming in UNIX File Systems

We can view the UNIX file system as being composed of five *layers*[1] of abstraction, each with its own naming scheme:

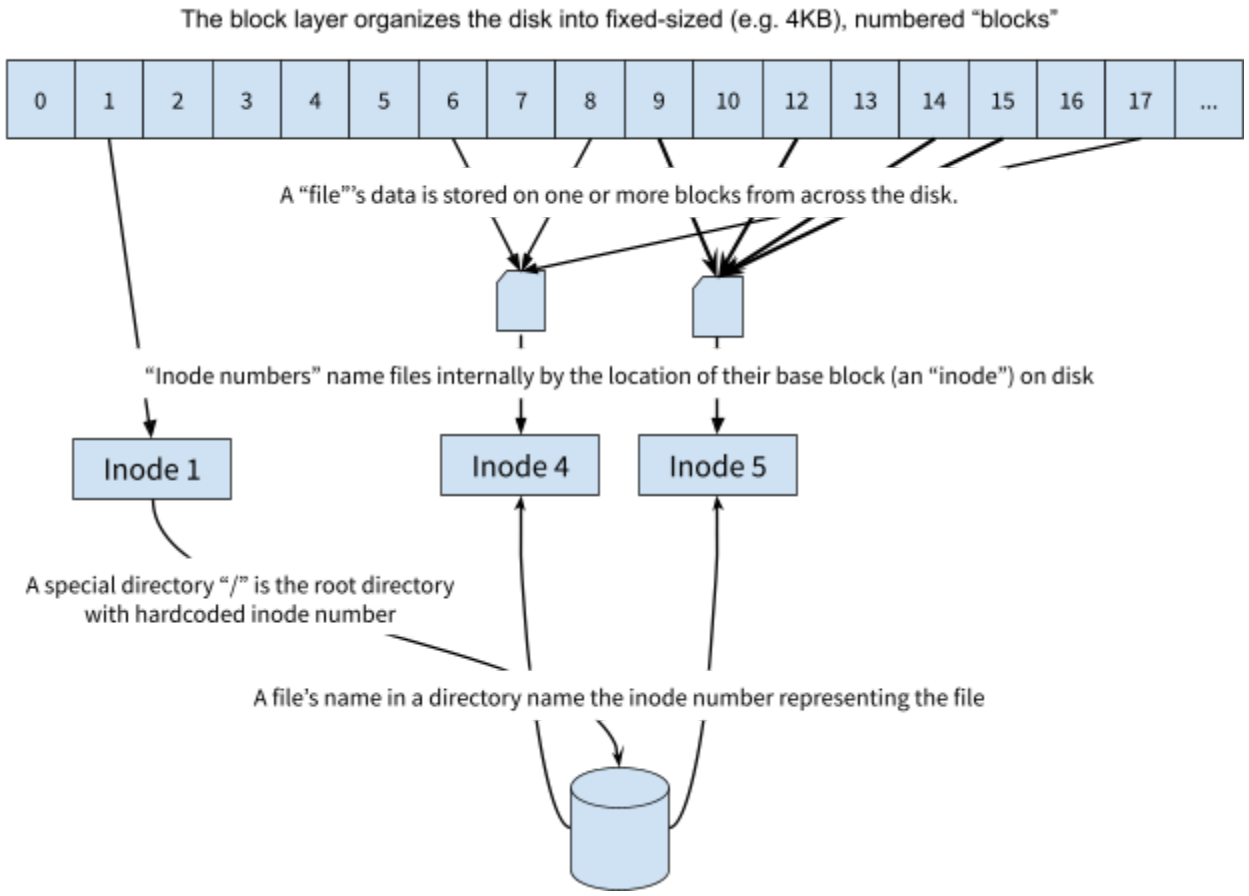
Layer	**Purpose**
Block layer	Organizes storage into fixed-sized blocks
File layer	Organizes blocks into arbitrary length-files
Inode number layer	Names files with unique numbers
Directory layer	Provides human-readable names for files in a directory
Absolute path name layer	Provides a global namespace

Each layer in this hierarchy uses a naming scheme to abstract details of the lower layers to the layer above it, providing portability across storage media, the ability to manage storage devices without user cooperation, and a balance between machine-useful naming and human readable names.

We can understand the naming scheme for each layer using the properties of naming schemes we discussed in previous lectures:

1. *Names*: what is the format of names and what space are they drawn from
2. *Values*: what values to names represent
3. *Allocation*: how do we allocate new names and values
4. *Lookup*: how do we find a value given its name

[1] The Saltzer book describes two additional layers: the directory layer is described as two layers (the “File name layer” and “Path name layer,” and an additional layer called the “Symbolic link layer,” which we just elide here).



The Block Layer

Storage media use a variety of different mechanisms to organize storage. Hard disks use spinning *platters* and moving arms to address data, which is organized into *cylinders* on each *platter*. Tape organizes data as a single contiguous magnetic stripe. SSDs are formed from arrays of NAND flash that organize data into *planes* for parallelism, and further into *dies*, *blocks* and *pages*.

The role of the block layer is to expose these diverse media through a single abstraction that preserves important performance characteristics common to those media.

In particular, the block layer divides the storage device into fixed-sized "blocks" of contiguous memory. Block sizes vary by device, but 4KB blocks are typical today. Each block is numbered with a "block number" starting at 0.

This enshrines two common properties of many storage devices. First, that storage may not be completely contiguous (e.g., hard disks are organized into cylinders and SSDs may have internal sections of NAND flash which is invalid). Second, while storage devices generally allow for random access, devices generally perform best when accessed in relatively large contiguous chunks.

Block numbers allow higher layers to reference chunks of storage without knowledge of their specific location or their physical organization. The relatively large size of blocks ensures higher layers use the underlying hardware in such a way that takes advantage of typical performance tradeoffs.

1. Names

The block layer uses positive integers, starting from 0, to name blocks. The largest block number is the size of the total size of the device in bytes, divided by the blocksize. For example, a hard disk with a capacity of 4GB and a 4KB block size will have 4GB / 4KB = 1048576 blocks.

2. Values

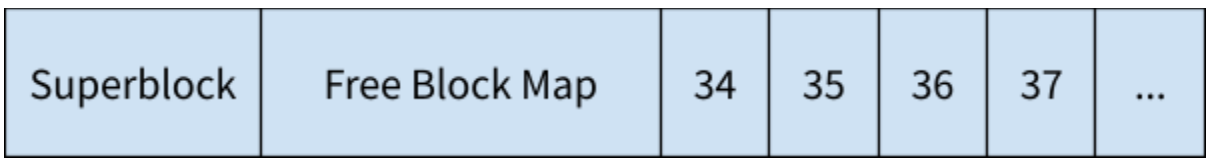
Each block is a block-sized contiguous array of persistent memory, often 4KB.

3. Allocation

Each potential block has a pre-assigned block number based on its position in the storage device. But in order to allocate a new block and supply its number we need to keep track of free blocks.

UNIX file systems use a special block (usually block number 0) called a “super block” to store metadata, which includes a data structure to keep track of free blocks.

Early file-system implementations used a linked list of free blocks with a head pointer stored in the super block. However, this can make it hard to find contiguous blocks (which is desirable for large files). So many file systems keep a bitmap marking free blocks in the super block or some number of subsequent blocks.



To find a free block, we simply search the free block map for a zero-bit. The index of the zero-bit corresponds to the free block's number. We mark this block occupied (by flipping the bit) and return the number.

4. Lookup

Mapping a block number to a block is device specific, and depends on how we lay out storage on the device. However, given a block name *i* we use the mapping of blocks to return the *i*th block.

The File Layer

Blocks give us a way to name fixed-sized chunks of persistent storage. But files have arbitrary size, include metadata (permissions, creation and modification times, owner, etc), and may grow or shrink over time.

In particular, files often contain data that spans multiple, potentially non-sequential, blocks. We need some way to find all the blocks associated with a particular file and order them.

The file layer accomplishes this using an object called an "inode" (probably short for "index" node, but the origin is not entirely clear). An inode includes file metadata (size, type, permissions, ownership, etc) as well as data structure storing an ordered list of block numbers for blocks that store the file contents.

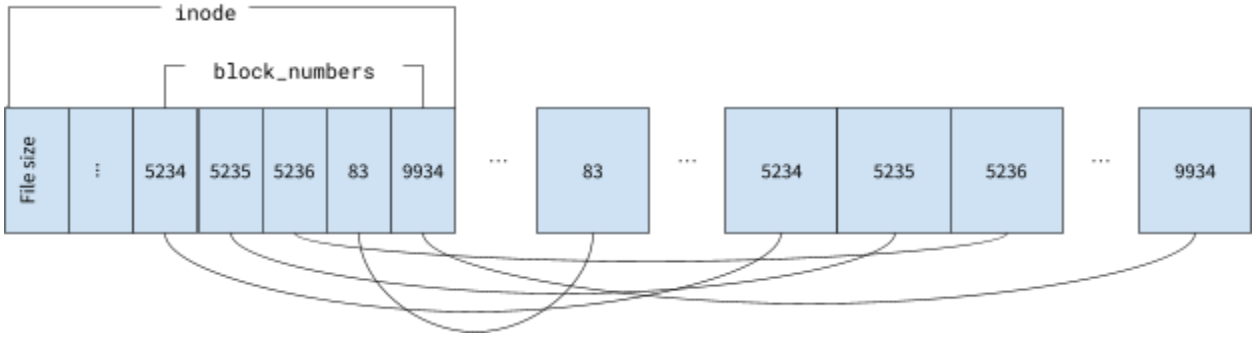
1. Names

We name files using an inode struct. Different file systems use different structures for the inode, but a reasonable model looks as follows:

```
struct inode {
  int32_t filesize
  … other metadata…
  int32_t block_numbers[N]
}
```

The inode structs above includes metadata and stores an array of block numbers inline, up to a fixed maximum *N* limited by the maximum size of the inode itself (since the array needs to fit inside the inode struct).

For example, it's common to store one inode in one block, such that the inode struct is at most the size of a block. In our example representation, with 4KB blocks, this would limit the total size of a file to under 4MB: an inode itself can be no larger than 4KB, meaning the block_numbers array is at most 4KB (actually smaller since there must be room for metadata as well. Since each element of the array is 4 bytes (a 32-bit integer), we can have at most 4096 / 4 = 1024 entries. Each entry, in turn, points to a block, which is 4KB. So each file can have at most 1024 * 4096 bytes = 4MB.



Exercise: How might we encode files larger than 4MB? How can we avoid a limit on the file size?

2. Values

Values in the file layer are… you guessed it... files: a linear array of bytes, along with file metadata.

3. Allocation

Since we use blocks to store inodes, a reasonable allocation scheme would be to reuse the block allocation mechanism from the block layer.

Some file systems tailored for specific kinds of storage media and workloads may have other allocation techniques. For example, it's common to cluster inodes together in order to avoid disk seeks when reading metadata of files from a single directory (e.g., when running `ls -l`).

4. Lookup

To lookup file data from an inode, we traverse the block_numbers data structure to find the block associated with the file offset we want to read or write:

```
def (inode *inode) offset_to_block(int offset) returns block:
  block_idx = offset / BLOCKSIZE
  block_num = inode.block_numbers[block_idx]
  return device.block_number_to_block[block_num]
```

The Inode Number Layer

A file isn't particularly useful if we can't get to it. The lowest level of naming for a file in UNIX uses an "inode number", simply a non-negative integer that uniquely identifies the inode struct for the file. Inode numbers are primarily used by the directory layer (see next section), but you can view your files' inode numbers using `ls -i`.

1. Names

   Inode numbers are non-negative integers (strikingly similar to block numbers) that uniquely identify an inode.

2. Values

   Values in this layer are the inode structs from the previous section.

3. Allocation

   We can reuse block allocation from the block layer to get blocks for new inodes and simply use the block number as the inode number.

   As discussed in the previous section, some file systems use special allocation schemes for inodes in order to cluster related inodes together on disk.

4. Lookup

   If we're reusing the block allocation scheme for inode allocation, lookup is similar: we use the inode number as a block number to get the block on which the inode is stored, and read the block as an inode struct.

The Directory Layer

Numbered inodes are great for software and hardware to reference individual files, but they are not particularly useful for humans. Moreover, naming specific files doesn't help users organize or discover files.

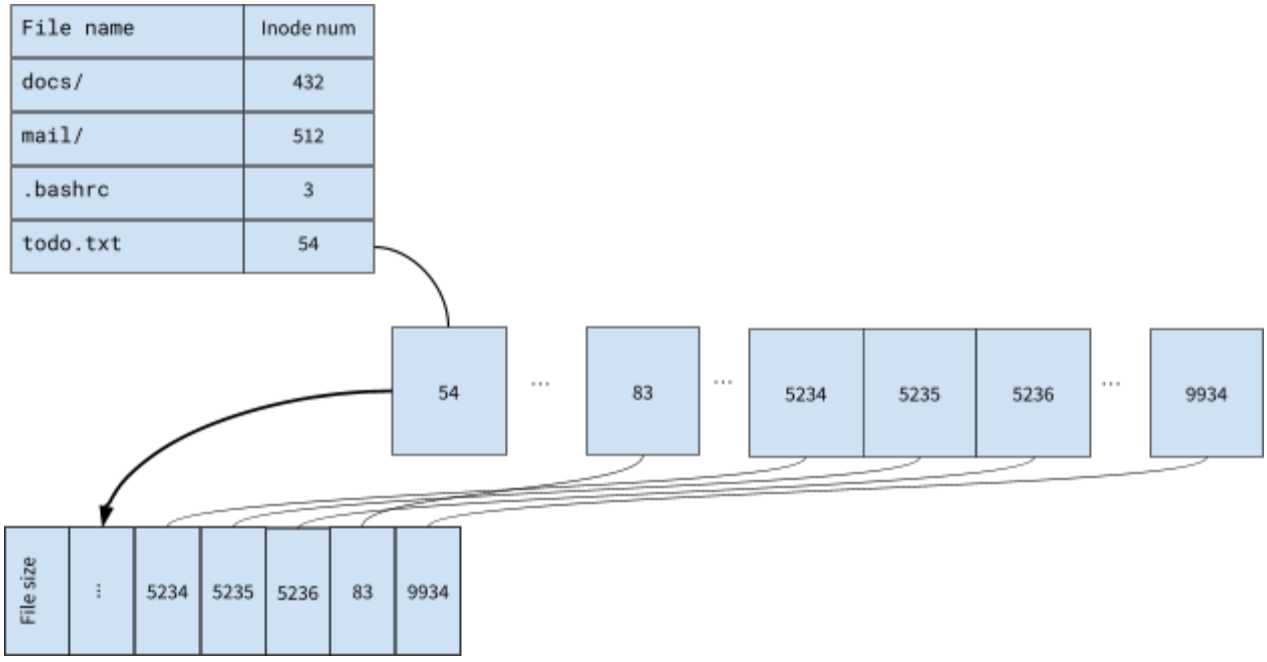
The directory layer helps us solve both of these problems by grouping files into collections called “directories” and giving them human readable names. Each directory maps human readable file-names---more or less arbitrary ASCII strings---into inode numbers corresponding to files.

Abstractly, a directory is just a list of directory entries, each with a filename to an inode number:

```
struct dirent {
  int32_t inode_num
  char    filename[MAX_FILENAME_LEN]
}

typedef directory dirent[MAX_DIR_SIZE]
```

UNIX file systems store these lists of directory entries as special file types, reusing the inode and file layers. One consequence of this is that filenames inside directories can themselves map to sub-directories, giving us a familiar hierarchy of directories for paths of the form: `path/to/file.txt`



1. Names

In principle, a file name can be any ASCII string. In practice, file systems often limit the length of the filename or the characters it may use. In other cases, such as Apple's HFS+, file names are case-insensitive, so "FILE.txt" and "file.txt" are considered the same. Finally, many modern file systems support unicode file names, rather than just ASCII.

2. Values

The directory layer maps file names to inode numbers, described in the previous layer.

3. Allocation

UNIX file systems store directories as a special file type. As a result, we create a new directory by allocating an inode and inode number, as well as blocks to store the directory entries.

To allocate a new mapping between a filename in the directory to an inode number, we simply add an entry in the directory's list. Of course, the file system must check for duplicates that already exist in the list.

4. Lookup

Given a filename inside a directory, we search the list of entries for one with a matching filename field and return the corresponding inode number field.

Give a relative path, we can use this algorithm recursively to resolve the entire path, traversing subdirectories along the way.