

PoCSD In-memory file system

Code overview

Overview

- The in-memory file system is used as a basis for various assignments used in class
- The code implements, in memory, a file system inspired by the design in Chapter 2 of the textbook
- You will progressively add complexity to this design to capture, in practice, concepts that we discuss in class

Code overview

- The code is written in Python, and maintains data structures that store information and expose methods to perform various functions across the layers of file system
- The data structures are abstracted as classes with methods that implement the majority of the “heavy lifting code” necessary to manipulate them
- This presentation overviews the major data structures and methods, in a “bottom-up” fashion – from the “raw” block layer up

Overall code structure and Python files

- fsconfig.py – various file system configurations (you won't change)
- fsmain.py – command-line argument parsing and initialization
- shell.py – interactive command shell for file system operations
- fileoperations.py – file operations: create, write, read, ..
- absolutepath.py – absolute path layer; from path to inode number
- filename.py – Lookup() and helper functions (you won't change)
- inodenum.py – inode number layer (you won't change)
- inode.py – inode layer (you won't change)
- block.py – raw block layer

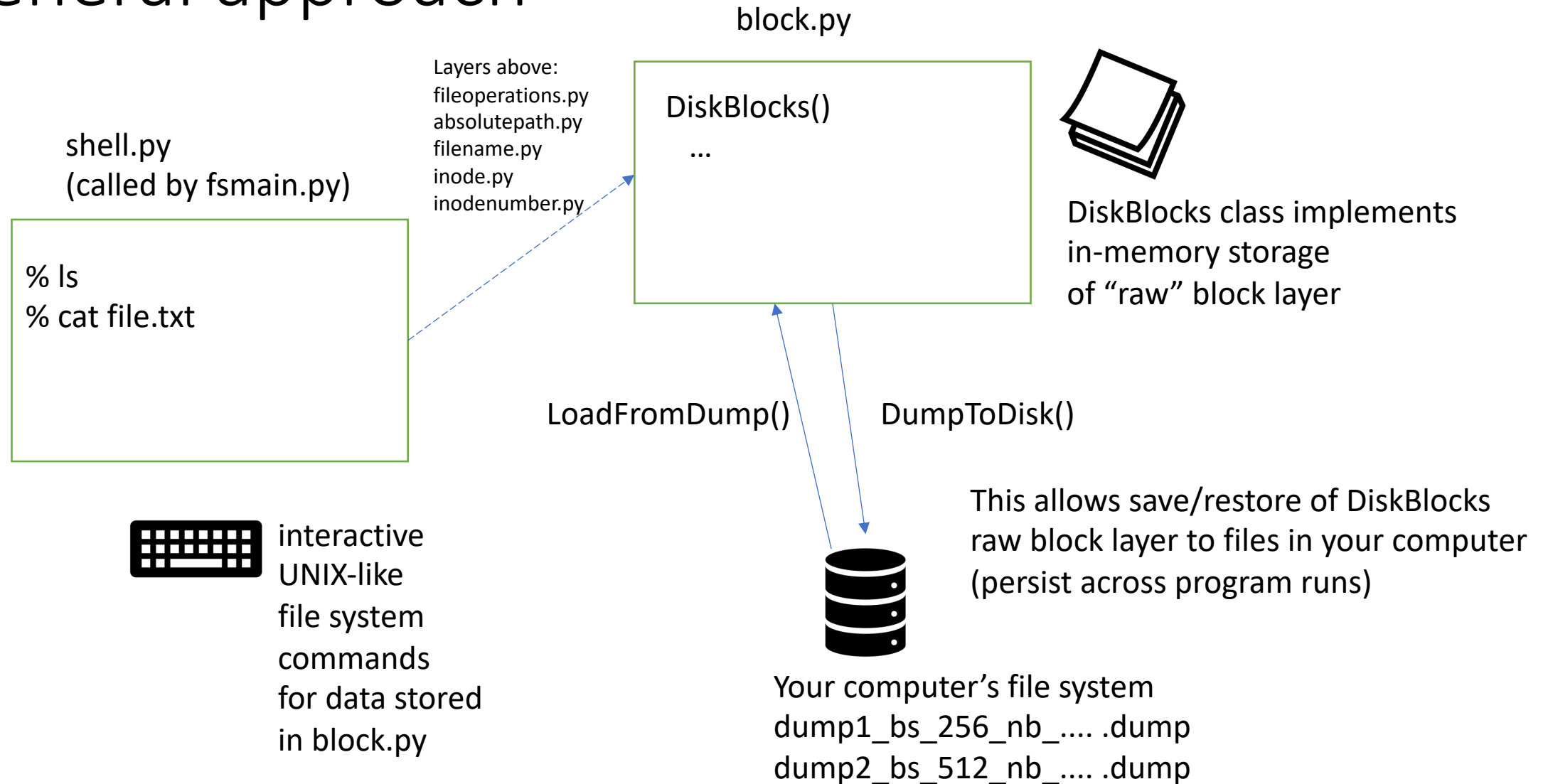
Layering on textbook's discussion

	Layer	Purpose	
	Symbolic link	Integrate multiple file systems	User-oriented names
	Absolute path name layer	Provide a root for naming hierarchies	
	Path name layer	Organize files into naming hierarchies	
	File name layer	Human-oriented names	User/machine Interface
	Inode number layer	Machine-oriented names	Machine-oriented names
	File layer	Organize blocks into files	
	Block layer	Identify disk blocks	

Diagram annotations:

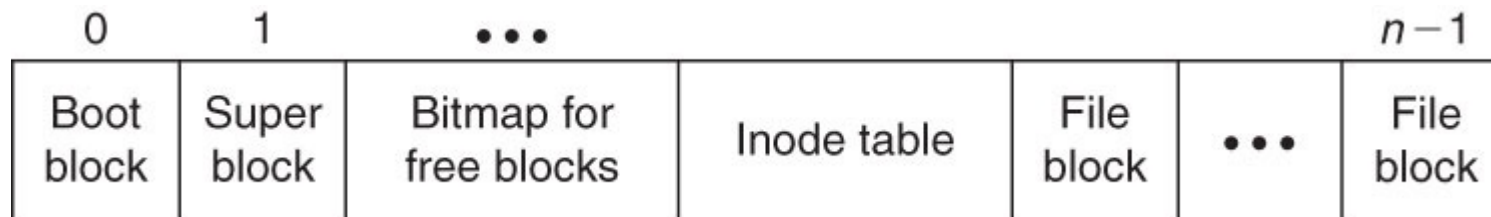
- HW3** (purple dashed arrow) points to the **Symbolic link** layer.
- absolutepathname.py** (red solid arrow) points to the **Absolute path name layer**.
- filename.py** (red solid arrow) points to the **File name layer**.
- inodenumnumber.py** (red solid arrow) points to the **Inode number layer**.
- inode.py** (red solid arrow) points to the **File layer**.
- block.py** (red solid arrow) points to the **Block layer**.

General approach



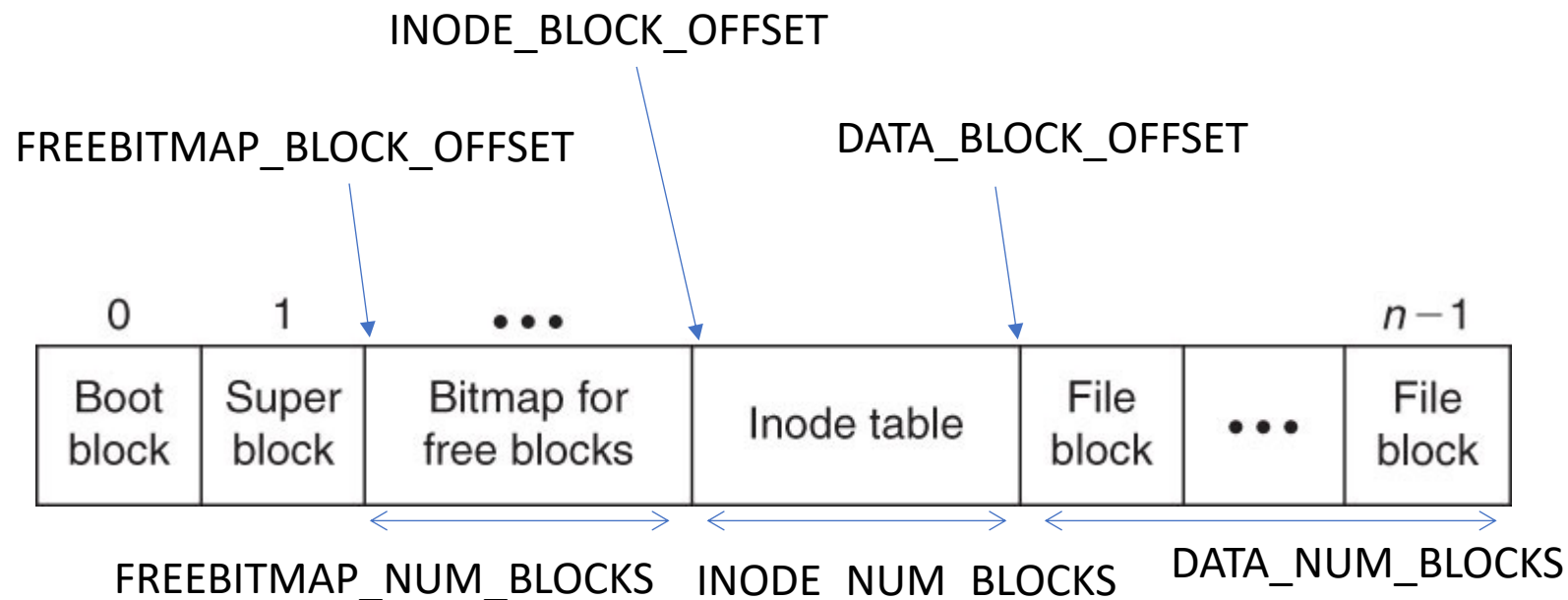
DiskBlocks()

- This class stores the raw block array
- The blocks are used to store file system data and metadata
 - Boot block – normally this would store boot code; it's unused in our class
 - Superblock() – stores file system constants
 - Bitmap for free blocks: stores an array of bytes, one byte per block, which determines if the block is free (0) or used (1)
 - Inode table: stores a table of inodes
 - File blocks: store file system object data (file contents, directory entries)



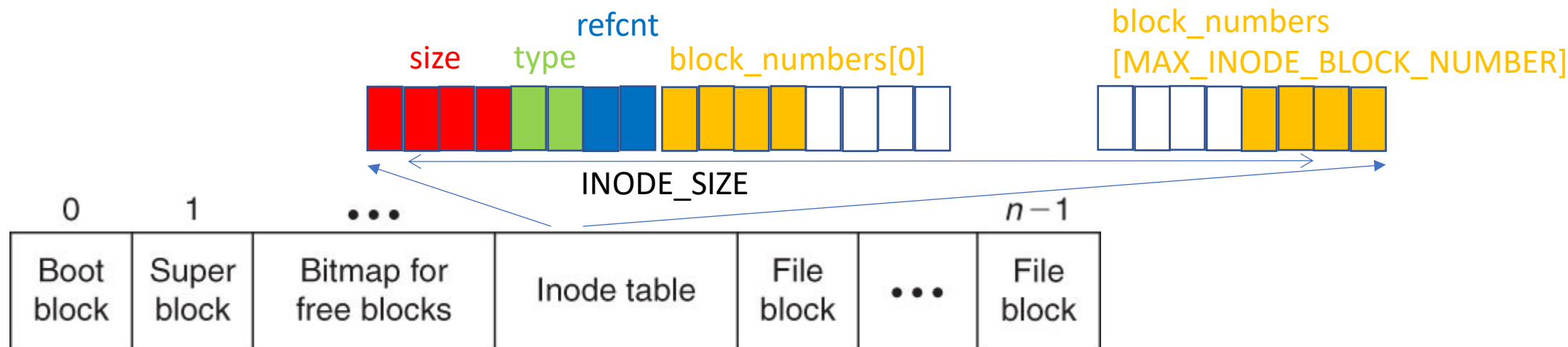
DiskBlocks()

- File system constants are declared at the beginning of memoryfs.py
 - BLOCK_SIZE (Bytes), TOTAL_NUM_BLOCKS, MAX_NUM_INODES, and INODE_SIZE (the size of each inode, in Bytes)
 - Additional constants used in the code are derived from them, such as:



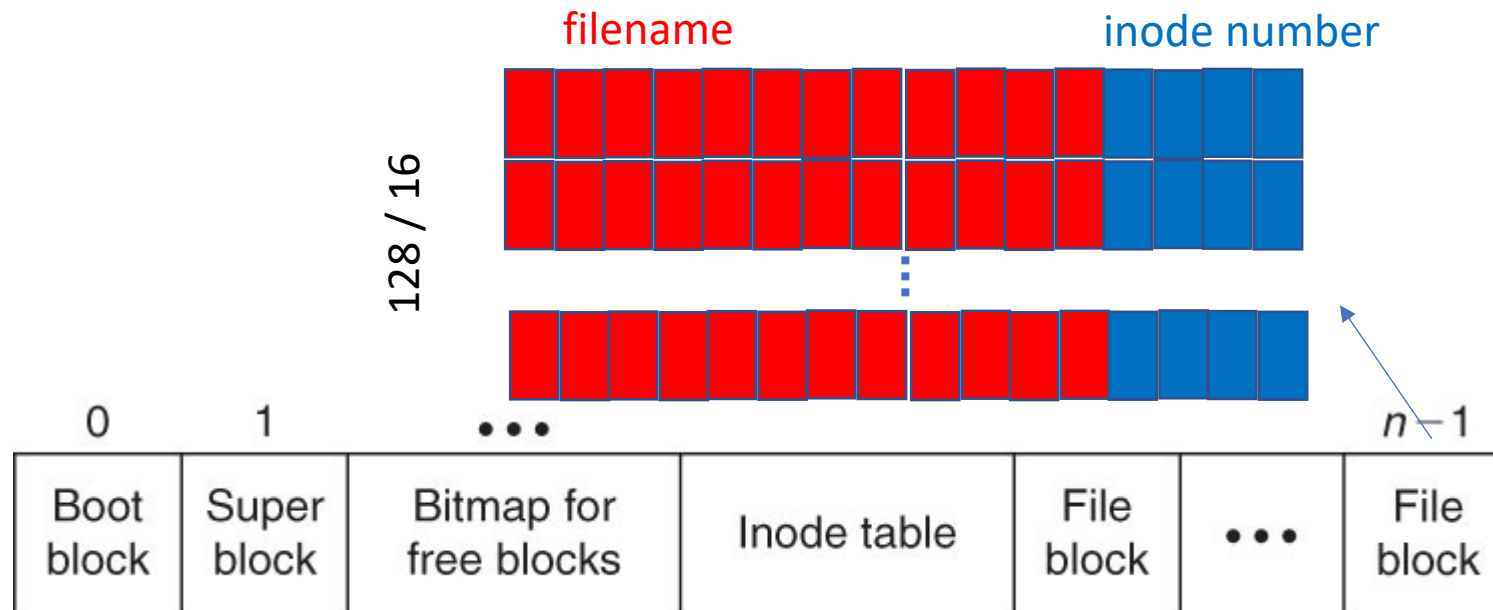
DiskBlocks() – encoding inodes

- Note: to simplify programming and readability, the following constants used for the inode data structure cannot be changed:
 - 4 Bytes are used to store **size** (hence maximum file size cannot exceed 4GB)
 - 2 Bytes are used to store **type** (plenty; we only have a handful of types)
 - 2 Bytes for **refcnt** (hence a file has at most 65535 references)
 - 4 Bytes for **each index in block_numbers** (hence at most 2^{32} blocks)



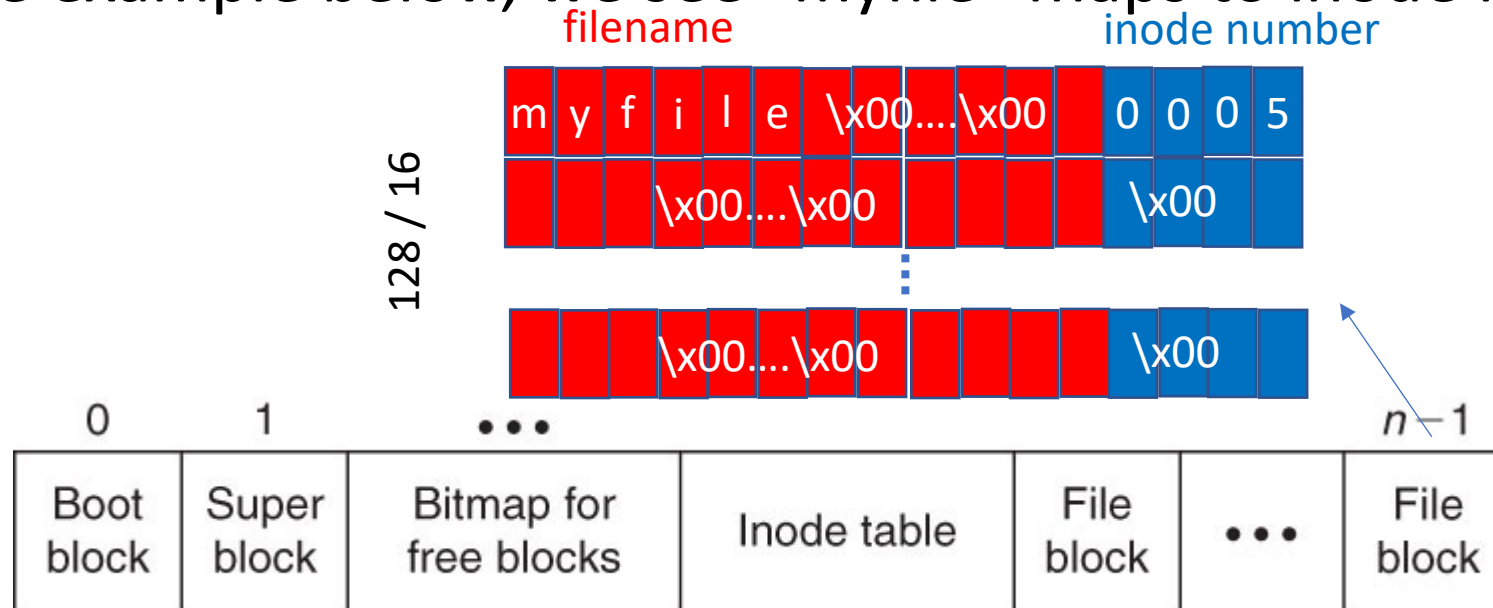
DiskBlocks() – encoding directory entries

- Each data block storing directory entries is structured as follows:
 - MAX_FILENAME determines the max **size** (characters) of an object's **name**
 - INODE_NUMBER_DIRENTRY_SIZE determines the **size** of an inode **number**
 - By default, MAX_FILENAME=12 and INODE_NUMBER_DIRENTRY_SIZE=4
 - Hence, each (name,inode) mapping uses 16 Bytes
 - With default block size 128 Bytes, there are up to 8 directory entries per datablock



DiskBlocks() – encoding directory entries

- The file name can have any characters, except “/”
 - The code doesn't enforce for encoding
- Unused file name bytes are filled with zeroes ('\x00' in hex)
 - A file name that is all zeroes means that the entry is free
- Inode numbers are encoded as big-endian
- In the example below, we see “myfile” maps to inode number 5



DiskBlocks()

- The key methods exposed by DiskBlocks() are:
 - Put(block_number, block_data)
 - Writes raw data block indexed by block number
 - Data block must be a byte array of up to BLOCK_SIZE length
 - If it is smaller than BLOCK_SIZE, Put() pads it with zeroes up to BLOCK_SIZE
 - Number must be between 0..TOTAL_NUM_BLOCKS-1
 - bBlock_data = Get(block_number)
 - Reads from raw block storage indexed by block number
 - block_data is a bytearray; block_number an integer
- DiskBlocks() is an in-memory class; two methods are exposed to “dump” the entire raw block storage to disk, and to read from disk
 - This is useful to save/restore the file system state between invocations of memory.py
 - DumpToDisk() and LoadFromDump() use Python’s “pickle” to serialize the object

DiskBlocks()

- DiskBlocks() constructor:
 - This method initializes the raw block storage in memory with zeroes
- LoadFromDump(self, filename):
 - This method initializes the raw block storage in memory from a file
 - You will be given examples of “dump files” in HW#1
 - You can create your own “dump files” to checkpoint block storage to disk – see below
- DumpToDisk(self, filename):
 - This method creates a dump file from the raw block contents in memory

Visualizing data in DiskBlocks()

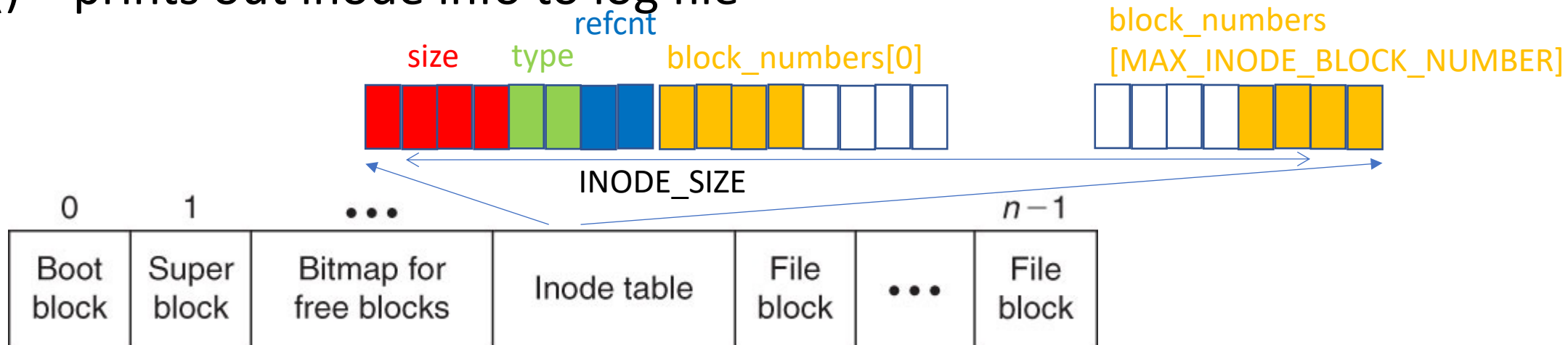
- The file system shell has debugging commands to help you visualize raw block data (and inode info), and get yourself acquainted with how data is stored in blocks:
 - **load** dumpfile, save dumpfile : load or save dumpfile in disk
 - **showfsconfig** : displays FS configuration, number of blocks, block size, number of inodes, inode size
 - **showblock** blockno : displays the contents of raw block number "blockno"
 - **showblockslice** blockno start end : displays a slice of the contents of raw block number "blockno", in hexadecimal format, from byte "start" to byte "end", including byte "end"
 - **showinode** inodeno : displays the inode data structure for inode number "inodeno"

Inode()

- This class represents an inode in memory
 - Its type, size, reference count, and array of block_numbers[]
 - The types supported in the version you start from are:
 - INODE_TYPE_INVALID
 - INODE_TYPE_FILE
 - INODE_TYPE_DIR
- Inodes are initialized as INVALID, size 0, refcnt 0, block_numbers[] as an array of zeroes

Inode()

- Helper functions extract/insert inode class from raw block storage
 - InodeFromByteArray(b)
 - This method loads an inode's type, size, refcnt, block_numbers[] from a byte array with INODE_SIZE bytes
 - InodeToBytearray()
 - This method returns a byte array encoding the inode's information
- Print() – prints out inode info to log file

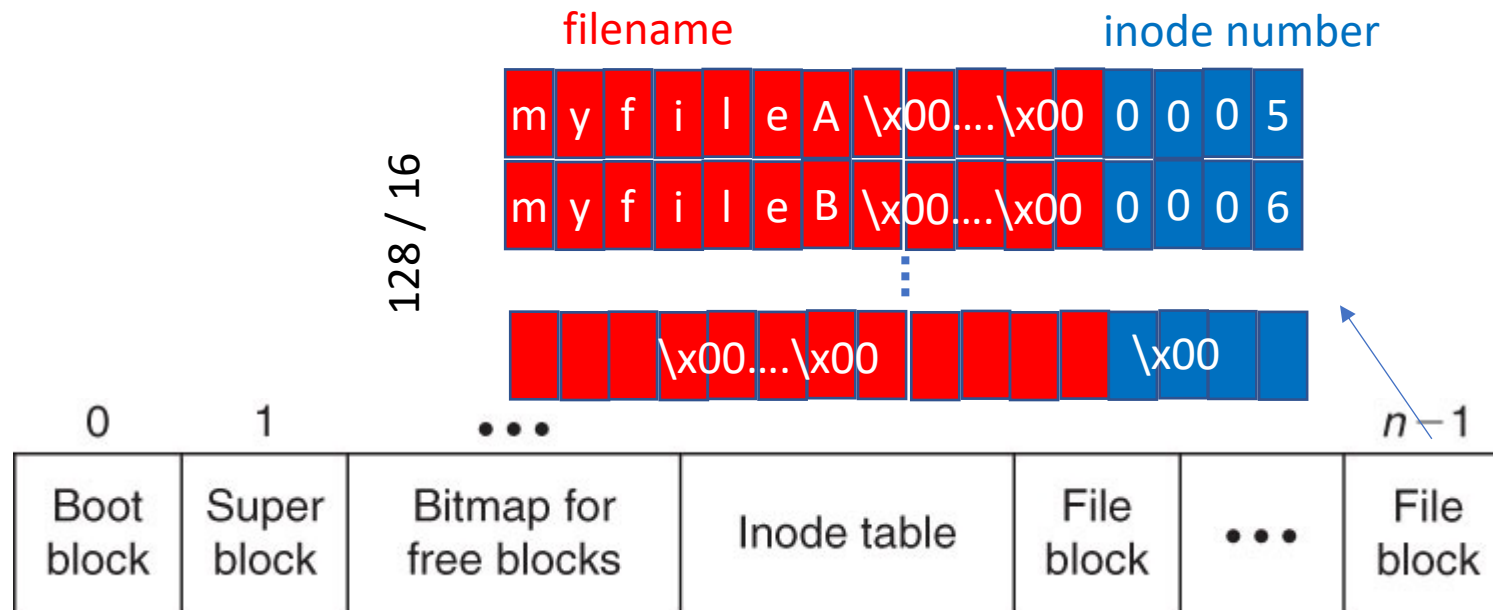


InodeNumber()

- This class exposes methods that use numbers to identify an inode
 - It holds an Inode() object, the inode's number
 - You need to initialize it with a number, and the object holding disk blocks
- InodeNumberToInode()
 - Load inode structure from raw block storage, given its number
- StoreInode()
 - Store inode structure in raw block storage, given its number
- InodeNumberToBlock(offset)
 - Returns a block of data from raw storage, given its offset; equivalent to textbook's INODE_NUMBER_TO_BLOCK

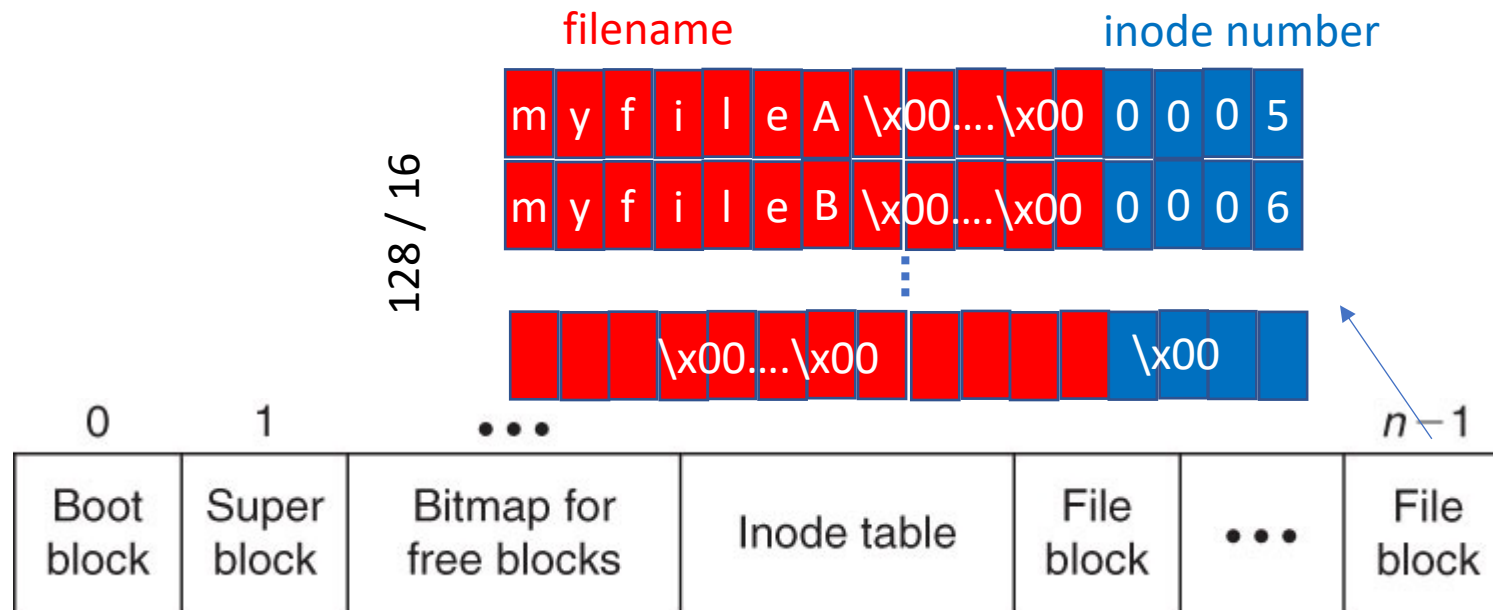
FileName()

- This class exposes methods at the file layer
 - You need to initialize it with the object holding disk blocks
- HelperGetFilenameString(block, index)
 - Extract MAX_FILENAME-size string from bytearray block, using index
 - E.g. with block below, index 0 returns “myfileA”, index 1 returns “myfileB”
 - Padded to the right with “\x00”



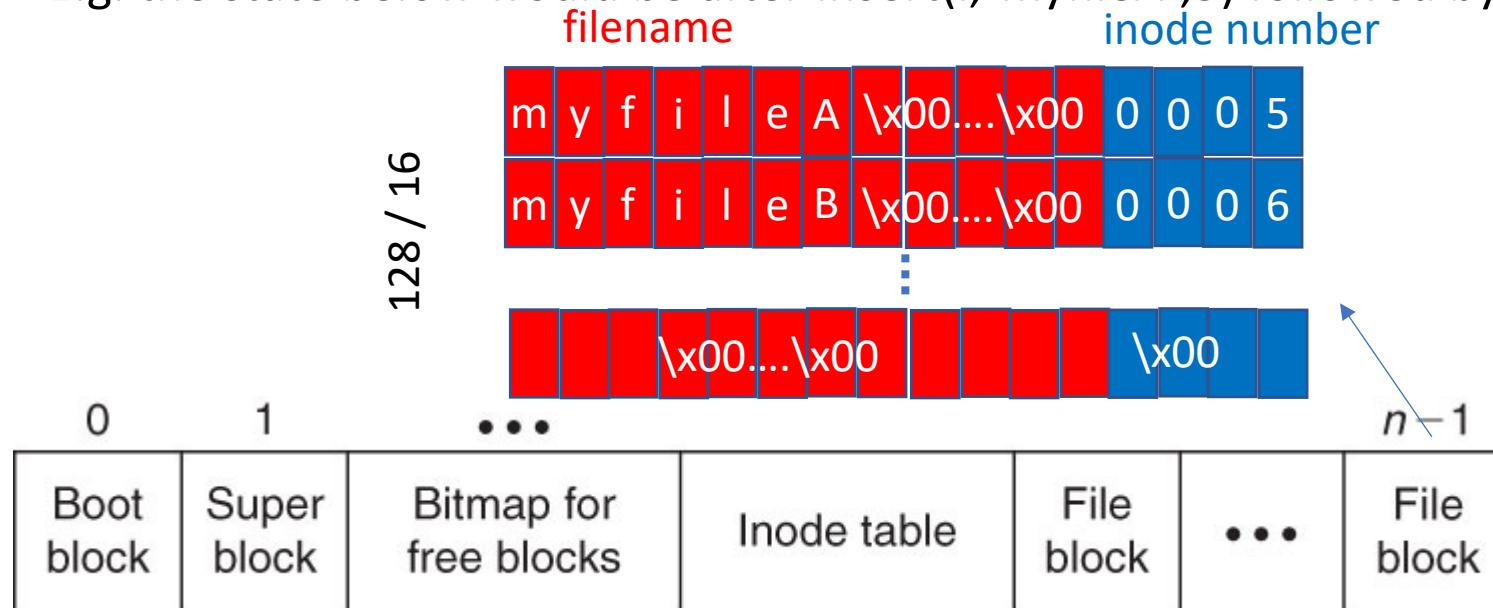
FileName()

- HelperGetFilenameInodeNumber(block, index)
 - Extract INODE_NUMBER_DIRENTRY_SIZE int from block, using index
 - E.g. with block below, index 0 returns 5, index 1 returns 6



FileName()

- InsertFileNameInodeNumber(insert_to, filename, inodenumber)
 - Inserts a (filename,inodenumber) entry to a directory's data block
 - insert_to is an InodeNumber() object - the inode number of the directory that we're inserting this entry
 - This adds an entry to the end of the table (starting from inode.size)
 - Allocates a new block and add to block_numbers[] if needed
 - E.g. the state below would be after Insert(i,"myfileA",5) followed by Insert(i,"myfileB",6)

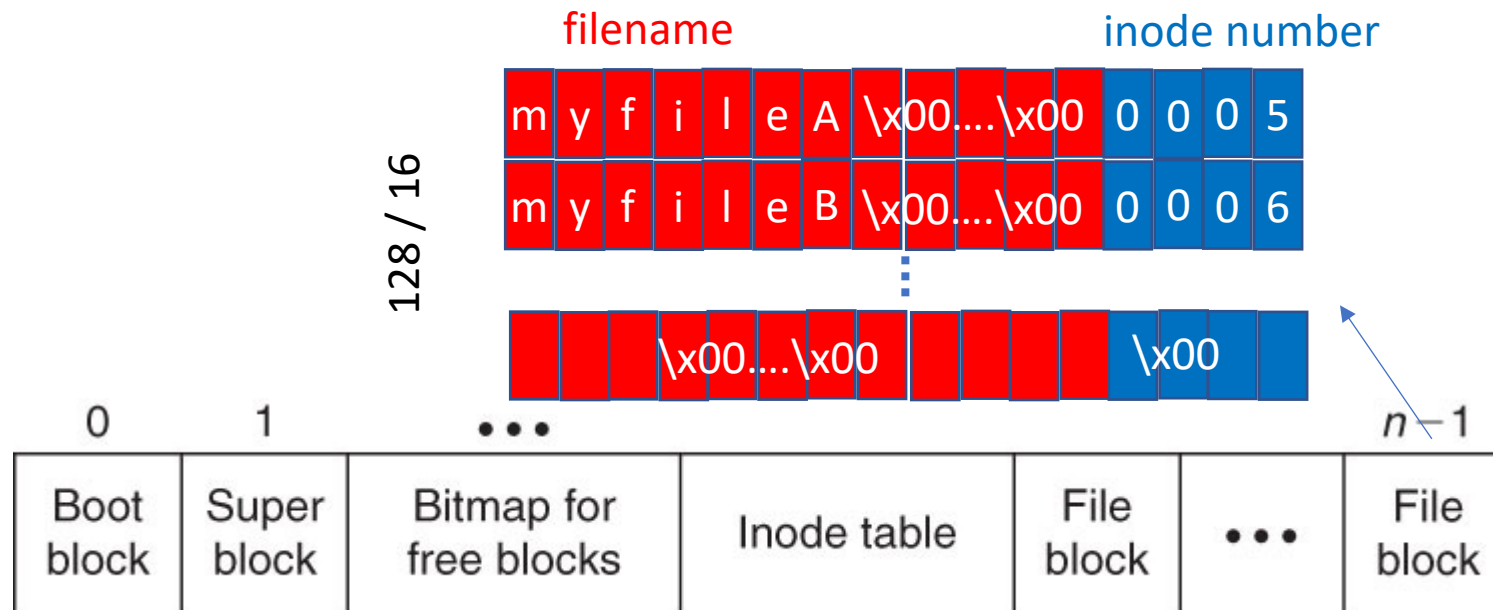


FileName()

- FindAvailableInode()
 - Scans inode table to find a free inode
 - Used when allocating a new inode as part of Create()
 - An inode is free if its type is INODE_TYPE_INVALID
- FindAvailableFileEntry(dir)
 - Finds next available block in a directory's block_numbers[] array
 - If inode reached maximum size, returns error
 - Used when allocating a new data block for a directory as part of Create()
- AllocateDataBlock()
 - Allocates a new data block from free list; if there is a free block available, update free bitmap, and returns block number
- InsertFileNameInodeNumber():
 - Insert (name,inode) binding to directory table

FileName()

- Lookup(filename, dir)
 - Looks up a file object in a directory – as in the book's LOOKUP
 - dir is an inode number, filename is a string
 - Example, Lookup("myfileA",0) would return 5 for the example below, assuming this is one of the data blocks indexed in the root directory (inode number 0)



FileName()

- InitRootNode()
 - Initializes root inode (inode number 0), and writes to block storage
 - Also inserts “.” directory entry
- Create(dir, name, type)
 - Create a file system object of given type, in given directory, with given name
 - Allocates a new inode for the object
 - Adds (name,inode_number) to dir’s entry, allocating new block if needed
 - If type is INODE_TYPE_DIR
 - Allocate data block for new directory, and insert “.” and “..” entries
 - If type is INODE_TYPE_FILE
 - Create empty file, with size 0 – no data blocks are allocated yet
 - Returns the new object’s inode number

FileOperations()

- Create(dir, name, type)
 - Create a file system object of given type, in given directory, with given name
 - Allocates a new inode for the object
 - Adds (name,inode_number) to dir's entry, allocating new block if needed
 - If type is INODE_TYPE_DIR
 - Allocate data block for new directory, and insert "." and ".." entries
 - If type is INODE_TYPE_FILE
 - Create empty file, with size 0 – no data blocks are allocated yet
 - Returns the new object's inode number

FileOperations()

- Write(file_inode_number, offset, data)
 - Writes bytearray “data” into file given its inode number, starting at offset
 - file_inode_number must index an inode of type INODE_TYPE_FILE
 - offset can point to anywhere within the file, and data can be smaller or larger than BLOCK_SIZE, but:
 - offset cannot be larger than file’s size
 - offset + len(data) cannot exceed MAX_FILE_SIZE
 - This method allocates data blocks as needed to accommodate data
 - Partial writes (smaller than a block) only change the bytes to be written
 - The block is first read from raw storage
 - Returns the number of bytes written