File System

Reference: Chapter on Persistence from textbook

Two key OS abstractions

- Process – virtualization of the CPU
- Address Space – virtualization of memory

- TOGETHER, these two abstractions allow a program to run
  - In its own private, isolated world
  - As if it has its own processor(s)
  - As if it has its own memory
- This virtualization makes programming much easier

Third abstraction – persistent storage

- Devices
  - Hard disk drive
  - Solid-state storage devices
  - Optical devices (still in development)
- Critical problem
  - How to present to the user an abstraction so that this works on all forms of OS

About hard drives and SSD

- Hard drives typically last about 4 years
- SSD have a limited number of reads and writes.
    - If you write about 40 GB per day
    - Then SSD will probably last about 10 years or more depending on use
    - Same as Flash, it will need to erase old data and then write to a new area

About Flash drives

- The average flash drive will survive anywhere between 10,000 and 100,000 uses. What this means is that for every time you write or read something from a flash drive, it takes up a “use”. Typically flash drives will fail from manufacturing defects of quality issues before they fail from being used too much.
- Erase one block at a time. To do an overwrite on the same file, it will remove the file from the directory, and then write to another place. The old file will be erased later.

Where are we?

Concurrency

Build a simple memory manager - DONE

Maybe a Simple FTP

Build a virtual disk to simulate a simple file system - NOW

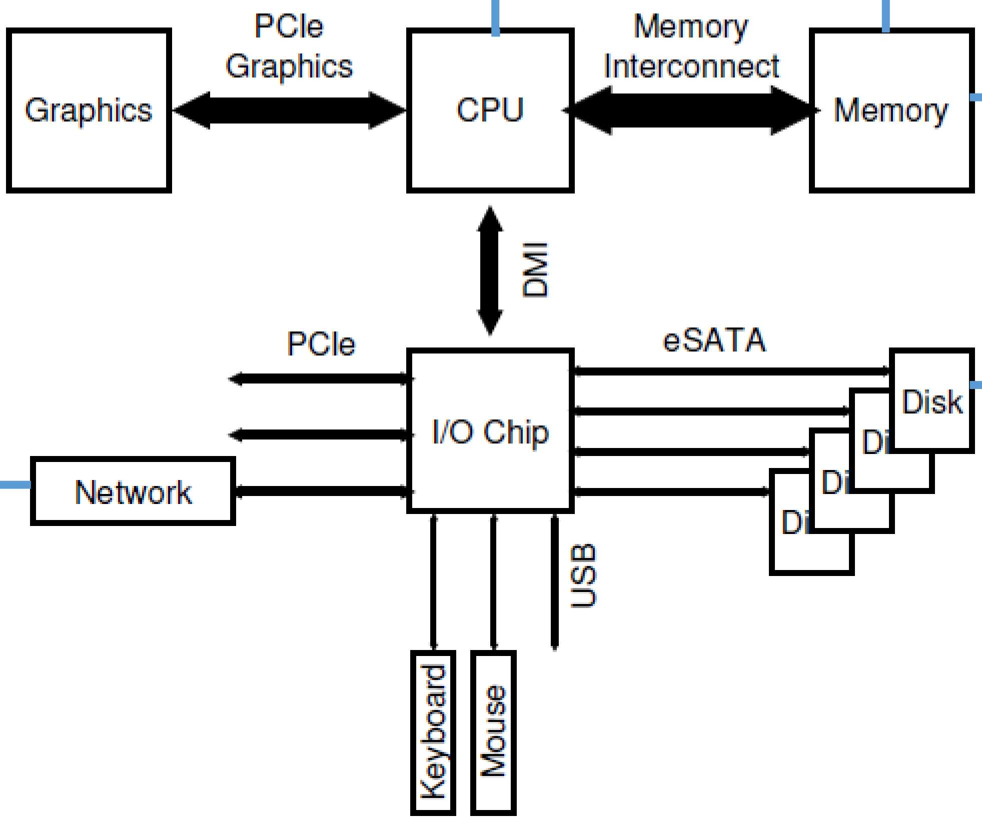


Figure 36.2: Modern System Architecture

Polling and Interrupts

- Polling involves waiting for device I/O to complete
  - Main disadvantage is that the cpu is kept busy just polling
- Interrupt is invented so one does not have to poll
  - Advantage: the cpu is free to allow another process to execute
  - Disadvantage: constant interrupting the CPU when a device has finished device also means the network.

- Recent findings show that it is not always better to have interrupts. Polling is making a strong comeback

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete
- The CPU has an *interrupt-request line* that is sensed after every instruction.
  - A device's controller *raises* an interrupt by asserting a signal on the interrupt request line.
  - The CPU then performs a state save, and transfers control to the *interrupt handler* routine at a fixed address in memory. (The CPU *catches* the interrupt and *dispatches*the interrupt handler.)
  - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a *return from interrupt* instruction to return control to the CPU. (The interrupt handler *clears* the interrupt by servicing the device.)

Two methods of device communication

- Explicit I/O instructions
  - A way for the OS to send data to specific device registers
  - Invented by IBM
  - Disadvantage: privileged instructions which mean the OS has to do it
- Memory Mapped I/O
  - Device registers are available as memory locations
  - Advantage: no new instructions are required as it is the same as reading and writing to memory locations

And then the device driver

- A specific and particular application
  - Communicates directly with the device either via device registers or memory-mapped locations
  - Provide an easy interface to users to access and communicate with devices
- Most of these are written in assembly language or C
- Newer languages have been developed
  - Rust
  - Julia

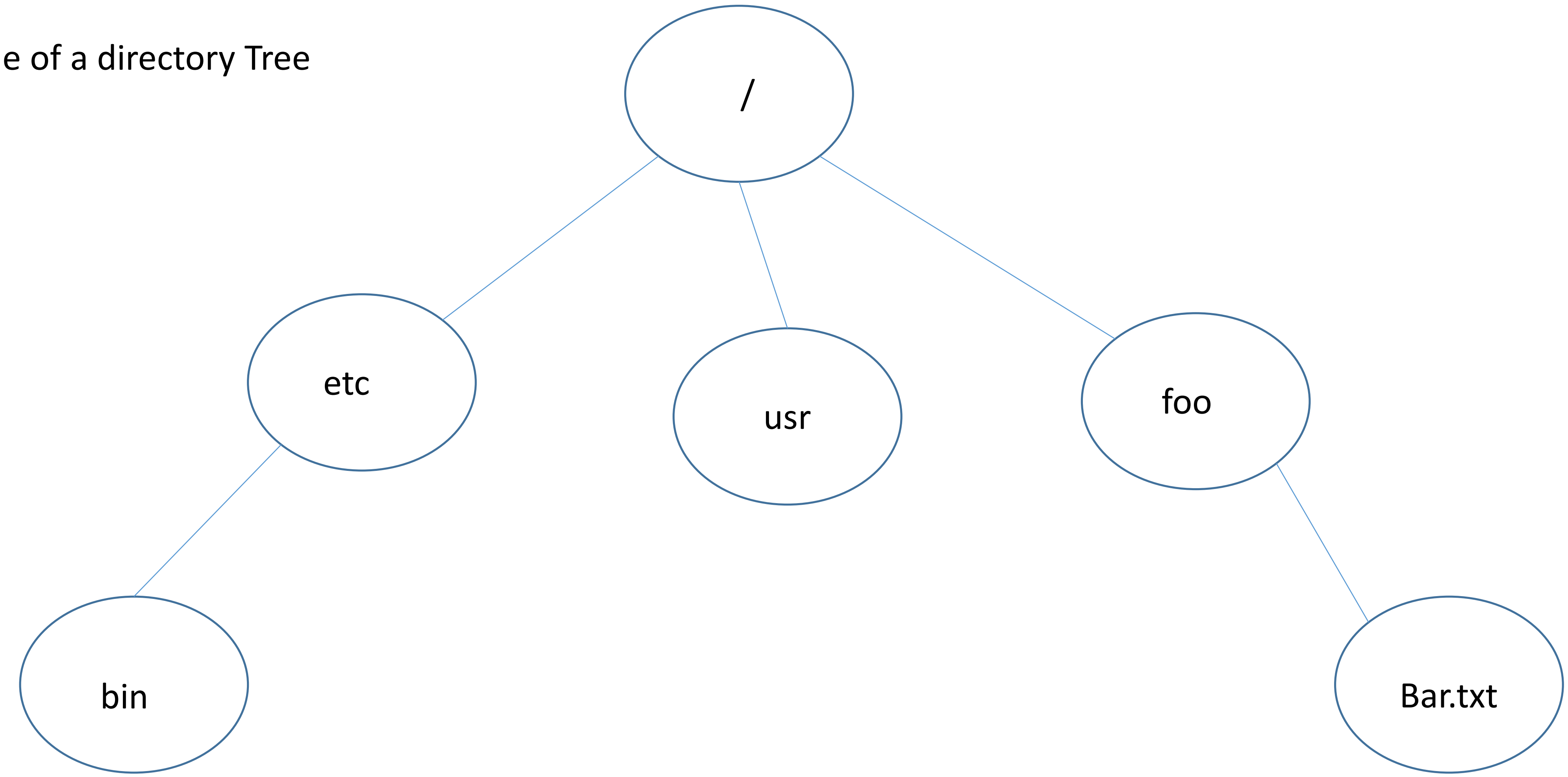
Abstraction of persistent storage

- Two key abstractions:
  - File
    - Linear array of bytes (RW)
    - Referencing – low level name or some number (inode number)
    - Most OS does not know about the structure of the file
      - Whether it is a picture, text file, encrypted file or python file
  - Directory
    - It is like a file with name and some number (inode number)
    - BUT the contents are specific
      - It contains a list (user-readable name, low-level number or number)

3 main functions of a file system

- Naming
  - Human readable file maps must be mapped to the associated disk blocks
- Organization
  - File systems normally have mechanisms that humans can use to organize information. A hierarchy is normal way and the directory tree structure is the main mechanism for organization. OBSERVE that directory is used instead of the word ‘folder’.
- Persistence
  - Data is stored in the file system and expected to remain there without being corrupted or deleted unless there is an explicit request to delete it. Support for allocation and deletion of files.

Example of a directory Tree



File operations with Python

- **fd = open('pressAkey.ipynb')**
- **type(fd)**
  - _io.TextIOWrapper
  - <_io.TextIOWrapper name='pressAkey.ipynb' mode='r' encoding='cp1252'>
- fd.fileno()
  - 4 { aka file descriptor or internally as an inode number)

File objects are built on top of C’s stdio package.

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

Open File table

- Each process maintains an array of file descriptors,
  - Each refers to an entry in the system-wide **Open File Table**
  - **Each entry tracks which relevant characteristics of the file – readable, writable, current offset, etc**

Open File table

System Call	Return Code	Current offset
fd = open(‘file’, ‘r’)	3	0
fd.read(100)	100	100
fd.read(100)	100	200
fd.read(100)	100	300
fd.close()	0	-

Open File table – two same entries

System Call	Return Code	OFT[10] Current offset	OFT[11] Current offset
fd1 = open(‘file’, ‘r’)	3	0	-
fd2 = open(‘file’, ‘r’)	4	0	0
fd1.read(100)	100	100	0
fd2.read(100)	100	100	100
fd1.close()	0	-	-
fd2.close()	0	-	-

Making and Mounting a File System

- Assemble a file system on persistent storage (say a drive)
  - Usually the command is ‘format’
- To make this file system accessible, we need to “mount” it
  - Command: mount
  - For windows, each file system is automount
  - Similar for Mac
  - On Linux, automount is done with manual override

Build a Simple File System

Essentially a virtual file system within a file

How to implement a Simple File System

- Questions:
  - What structures are needed on the disk?
    - Include abstractions
    - How do these structures need to track?
  - What kind of reference do we return to the user?
  - What kind of calls to provide the user to do operations like
    - Open
    - Read
    - Write
    - etc

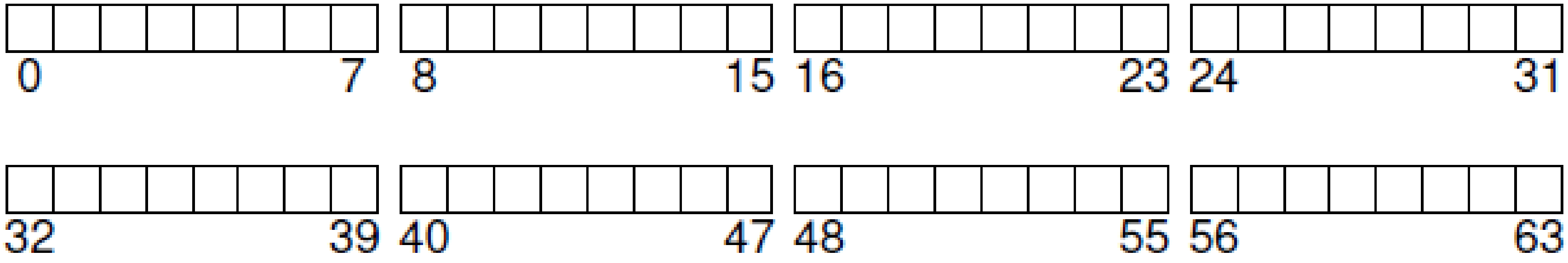
Data Structures of SFS (Simple File System)

- We are using the simple Unix way.
- Imagine (create a mental model) of a disk is partitioned into multiple blocks of data
- Each block is designated as
  - Inode
  - Data
  - Special (like the superblock)

An inode (index node) is a data structure in the traditional Unix-style file system

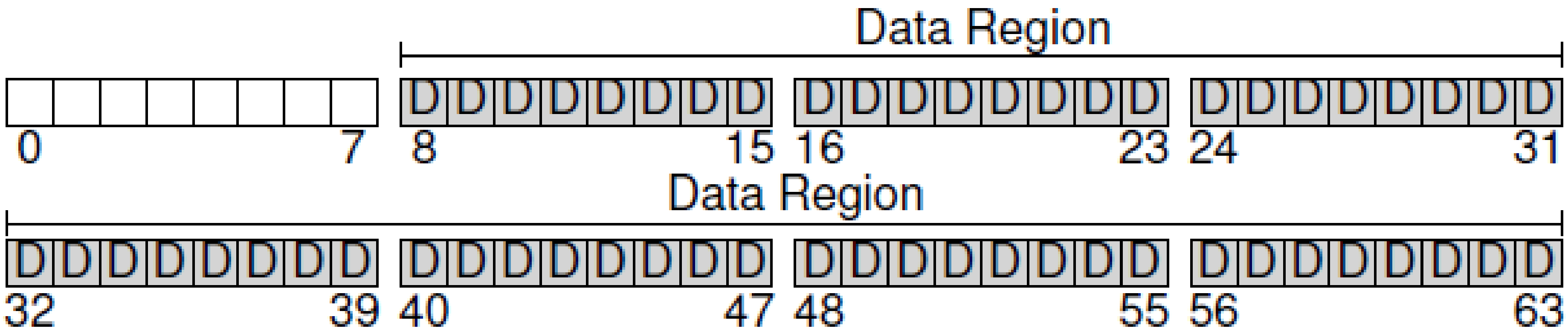
First Step: Divide the disk into blocks.
Each block is one fixed size.

I have done this for you.
With disk.py



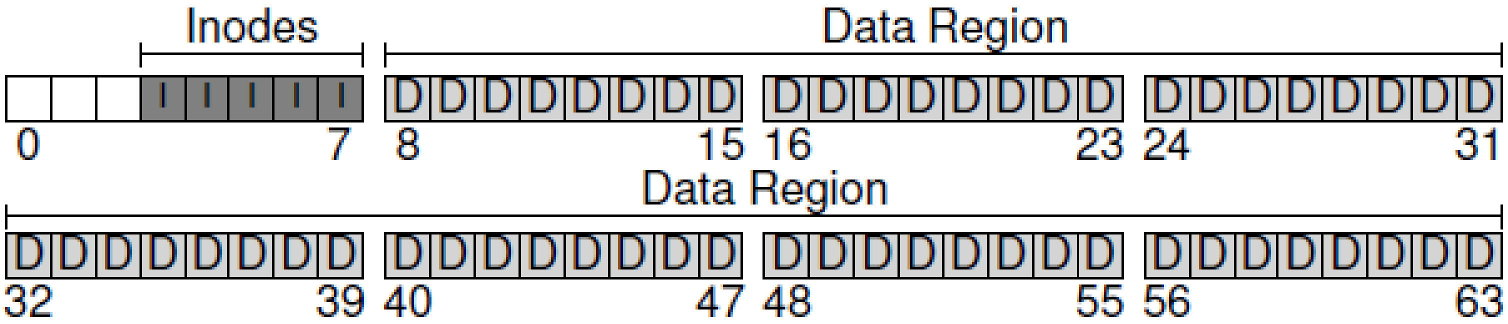
Most of the blocks will be filled with data from users.
We call these blocks to be data blocks.

We really don't care (at this stage) what kind of data.



The file system has to track information about each file.
This information (aka metadata in inode) consists of:

- **How many data blocks to a file**
- **Where are the data blocks**
- **Access rights, etc**



Most inodes are about 128 bytes or 256 bytes.

For a 4KB block, it can hold 16 inode if an inode is 256 byte.

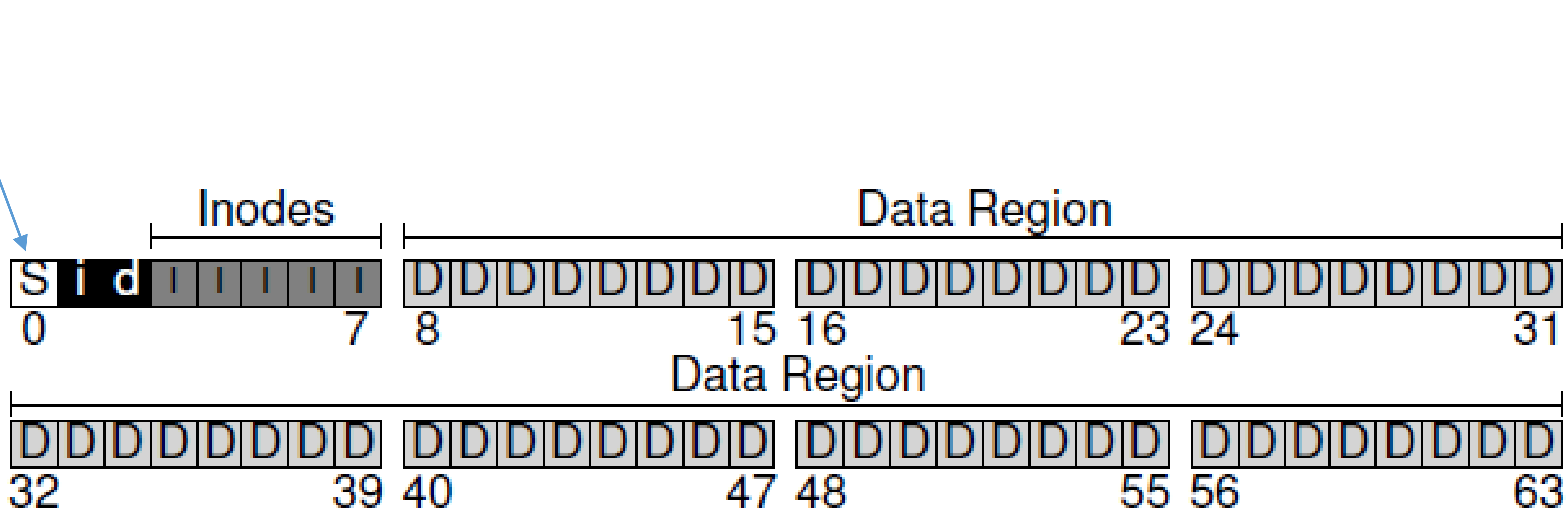
Need some way to track free data and inodes

- Use bitmap
  - One for the data region
  - One for the inode table (inode bitmap)

- A bitmap is an array of bits (or bytes) where each cell is either 0 (meaning free) or 1 (meaning in-use)

- And a block to track the file system - Superblock

Superblock

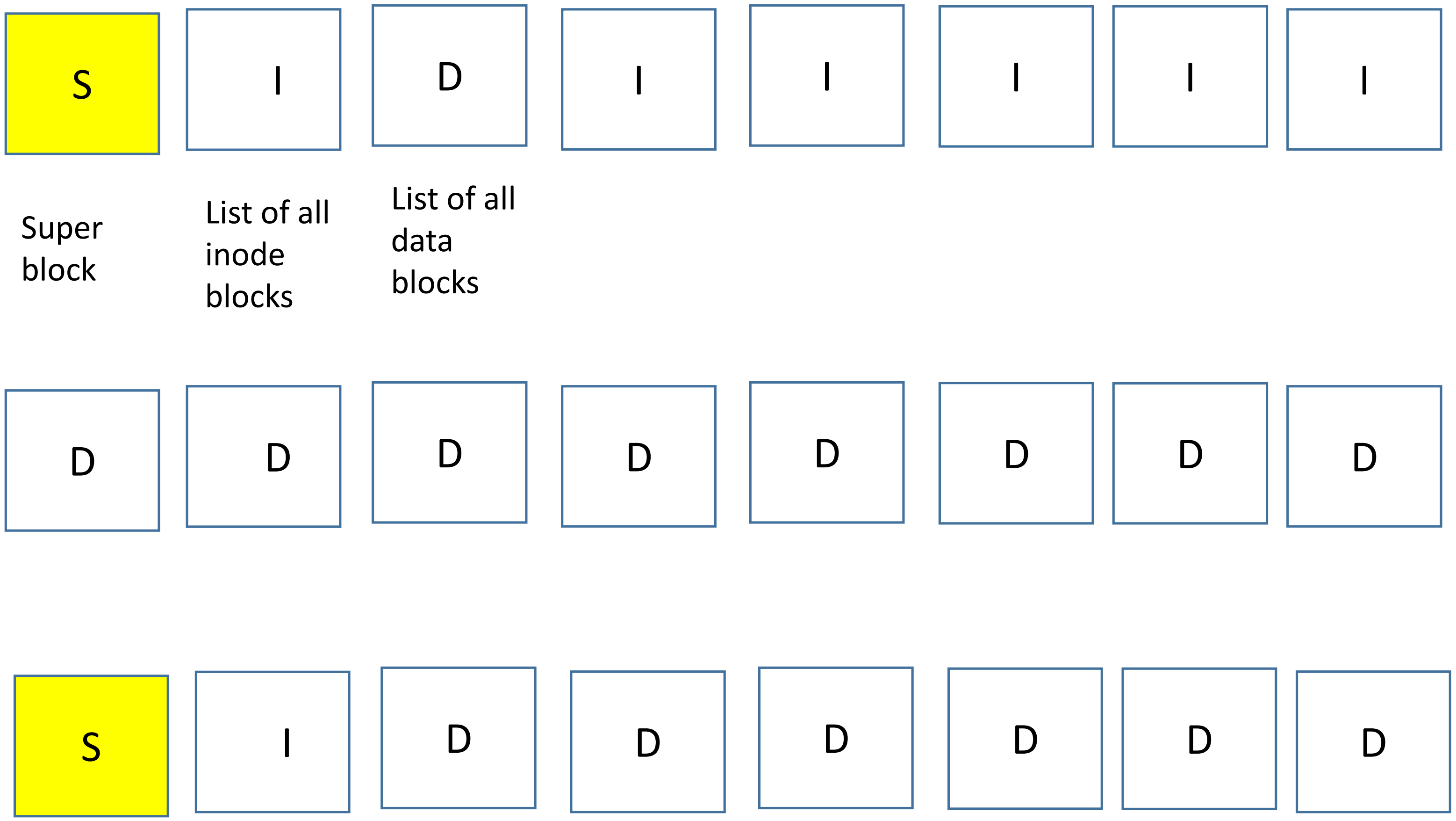


Superblock contains:

- Number of inodes
- Number of data blocks
- A magic number to identify the file system
- Various accounting features – like date of creation, etc

Logical Model : Block arrangement

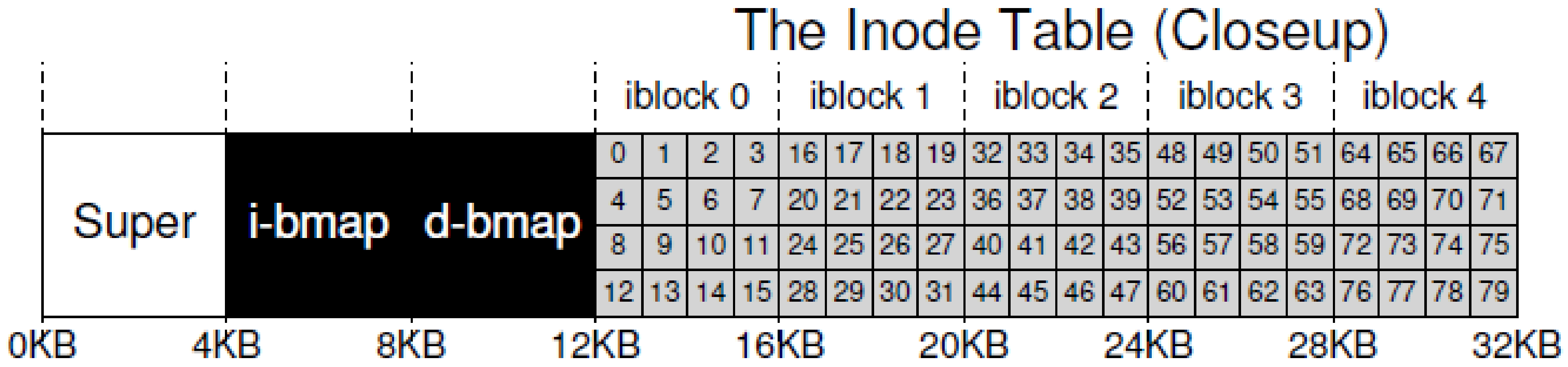
S – superblock
I – Inode block
D – data block



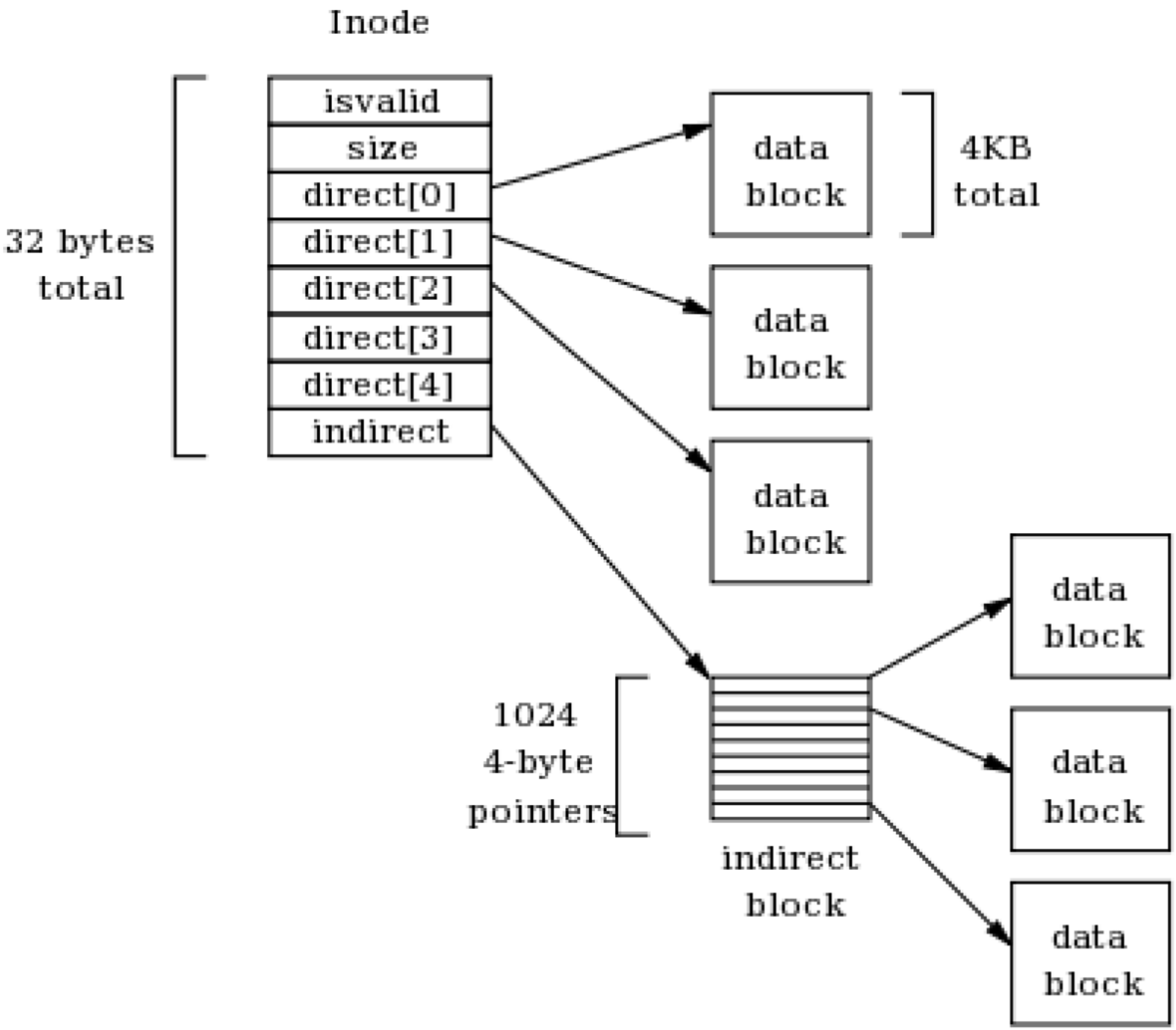
Once a disk is created,
The number of Inodes
And Data blocks
Are fixed.

Each inode is referred to by a number, i-number
or the low-level name of file,
or file descriptor.

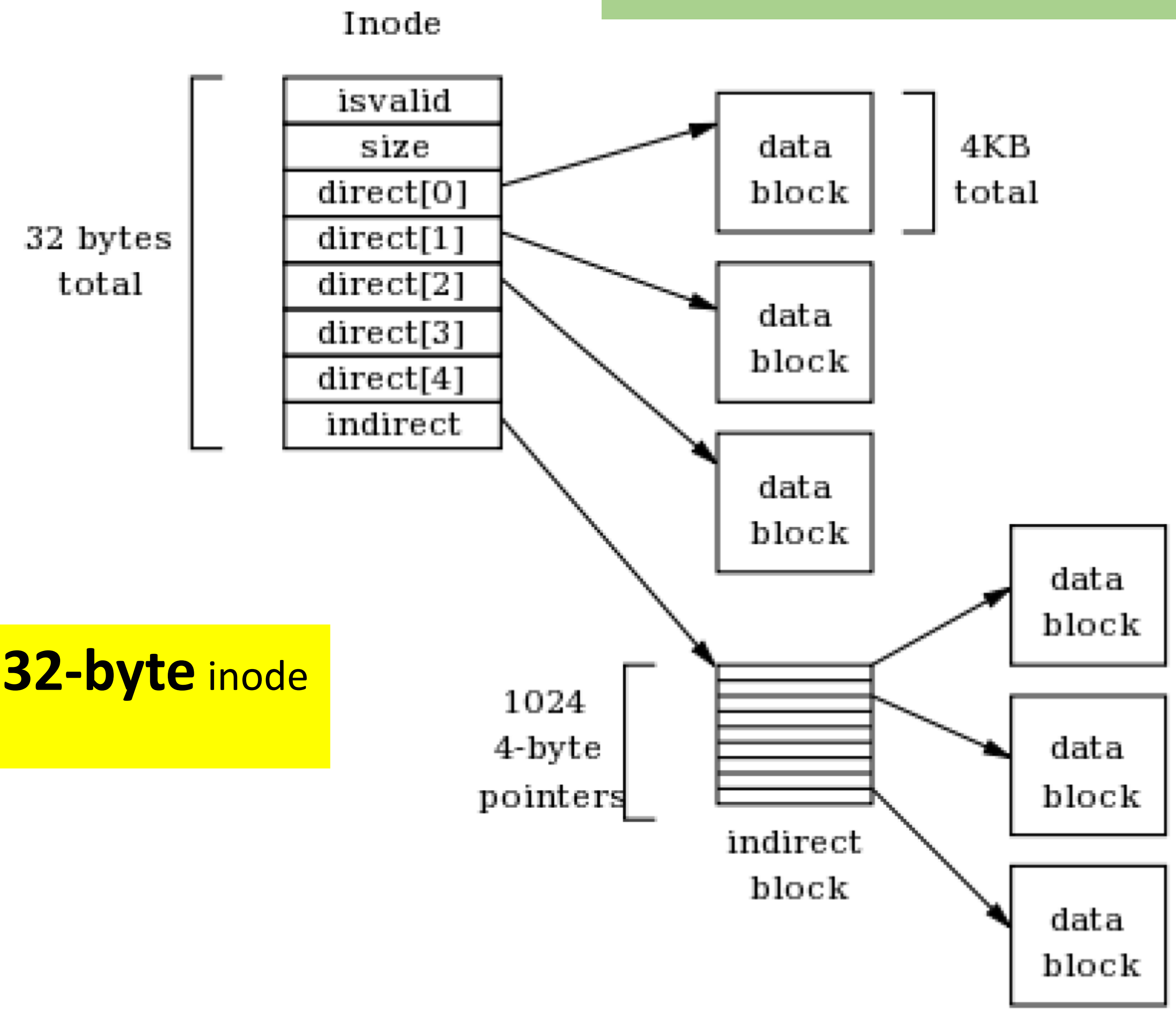
GIVEN an i-number, one is able to calculate Where on the disk the corresponding Inode is located.



Example: inode 33.
Inodeblock = 33 // nbr_Inodes_per_block = 33//16 = 2
Offset in block = 33 % 16 = 1 … index of 1
Actual block number on disk = Inodeblock + firstInodeBlock = 2 + 3 = 5


Inode
isvalid
size
direct[0]
direct[1]
direct[2]
direct[3]
direct[4]
indirect
32 bytes total
data block
4KB total
data block
data block
data block
data block
data block
1024 4-byte pointers
indirect block

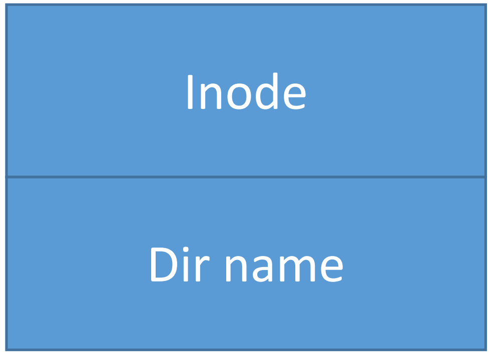
Index Node aka Information Node



isvalid is 1 when used
And 0 when not used
Size is the number of
Data blocks(for now)

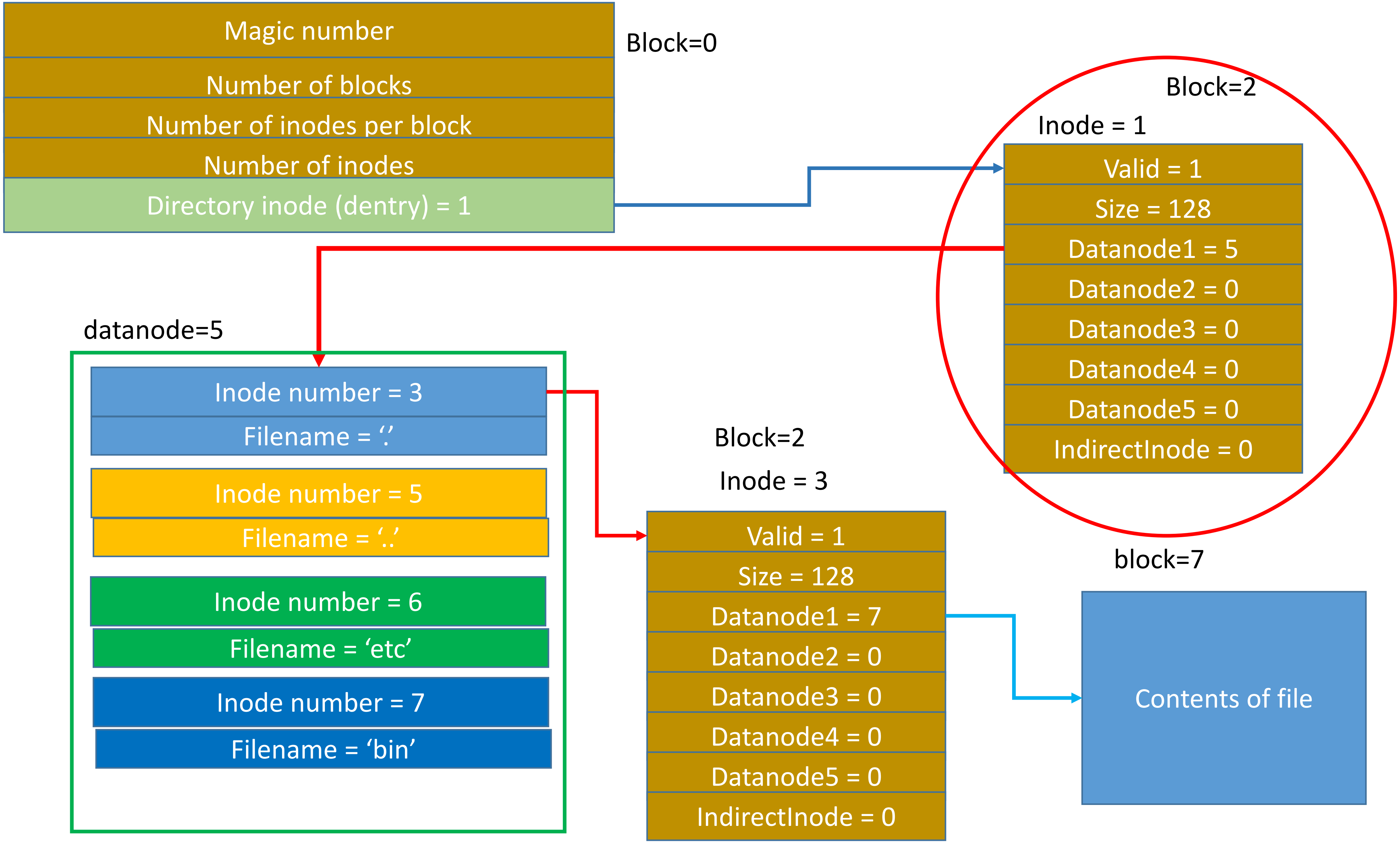
Write a list of **32-byte** inode
To a block

Directory Entry in a data block



Data block for /

Dir. entry	Field	Value
0	Inode	1
	Name	"."
1	Inode	1
	Name	".."
2	Inode	2
	Name	"etc"
3	Inode	3
	Name	"bin"
4	Inode	0
	Name	0

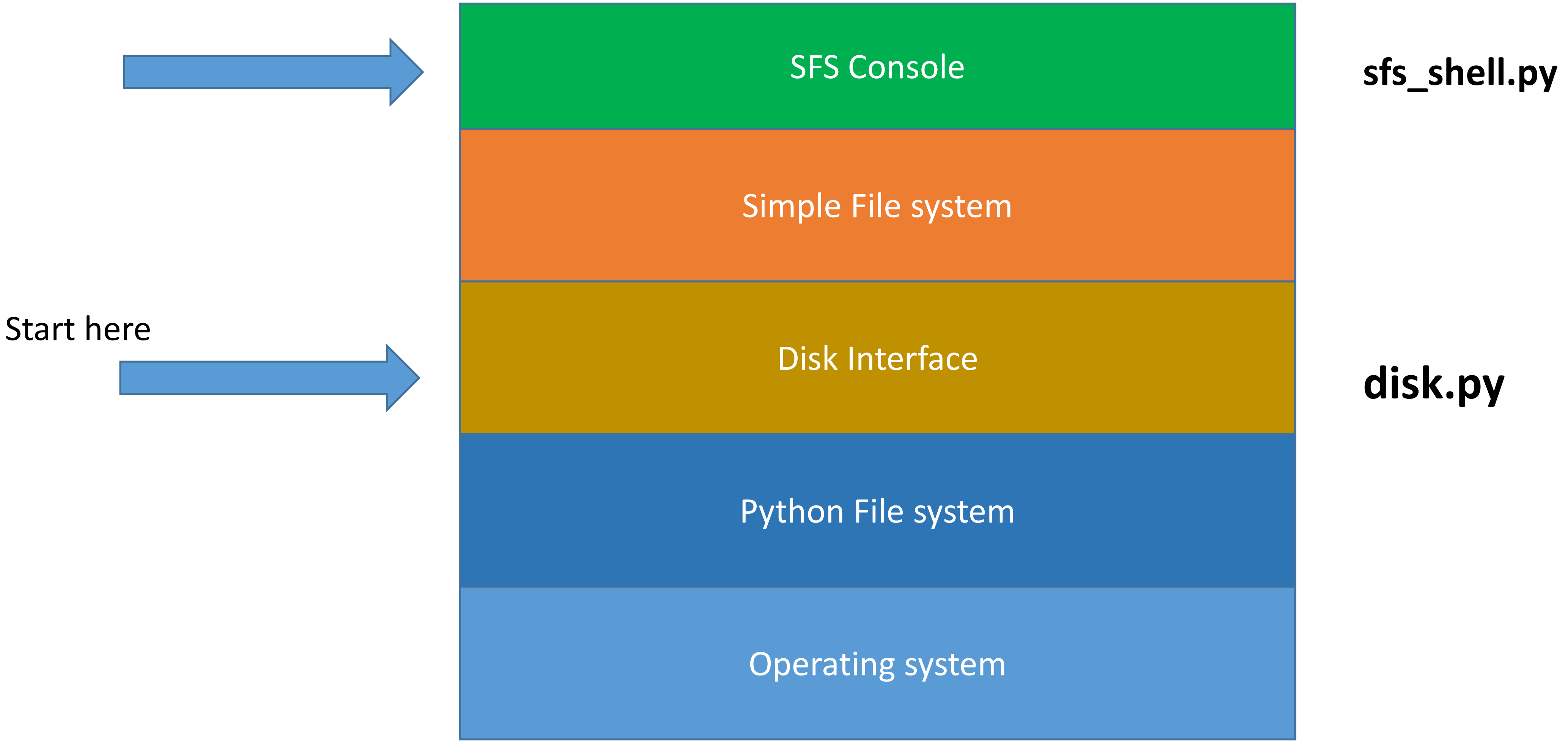


Magic number
Number of blocks
Number of inodes per block
Number of inodes
Directory inode (dentry) = 1
Block=0
Block=2
Inode = 1
Valid = 1
Size = 128
Datanode1 = 5
Datanode2 = 0
Datanode3 = 0
Datanode4 = 0
Datanode5 = 0
IndirectInode = 0
datanode=5
Inode number = 3
Filename = ‘.’
Inode number = 5
Filename = ‘..’
Inode number = 6
Filename = ‘etc’
Inode number = 7
Filename = ‘bin’
Block=2
Inode = 3
Valid = 1
Size = 128
Datanode1 = 7
Datanode2 = 0
Datanode3 = 0
Datanode4 = 0
Datanode5 = 0
IndirectInode = 0
block=7
Contents of file

Let’s start building SFS

And see how far we go

Overview – a layer diagram of what we intend to do



Create a directory for this project: sfs

- Put all files there
- sfs_shell.py (the console command shell)
  - https://code-maven.com/interactive-shell-with-cmd-in-python
- disk.py (the disk interface)

Disk Interface

- A disk can be rewritten in place
  - Read a block
  - Write a block
- Disk can be access directly
  - Access any file either sequentially or randomly

API for disk interface

- disk_open()   # open a file (hardwired) – perform a mount
- disk_read(blocknum)   # return a buffer of bytes of fixed size
- disk_write(blocknum, data) # write a block of bytes to given block
- disk_close()   # close the file – do an unmount
- disk_init()   # initialize a file with all the blocks

What is provided for you?

- disk.py – an implementation of read and write to a file
- sfs_shell.py – a command interactive shell to test out the disk commands

What you need to do in class NOW

- Create the Superblock
- And write it to disk
- And read it back to check
- Create command in sfs_shell to test these commands

Steps in building the SFS

Suggested filenames	Layer	
	SFS Console	sfs_shell.py (provided, but Expect lots of changes)
createfile.py	Create file	Functions for You to do
format.py	Format function	
mkdir.py	Create Directory (mkdir)	
datablockbm.py	Datablock bitmap	
inodebm.py	Inode bitmap	
inode.py	Inode	
superblock.py	Superblock	
	Disk Interface	disk.py (provided)
	Python File system	
	Operating system	

You can make changes to both provided files

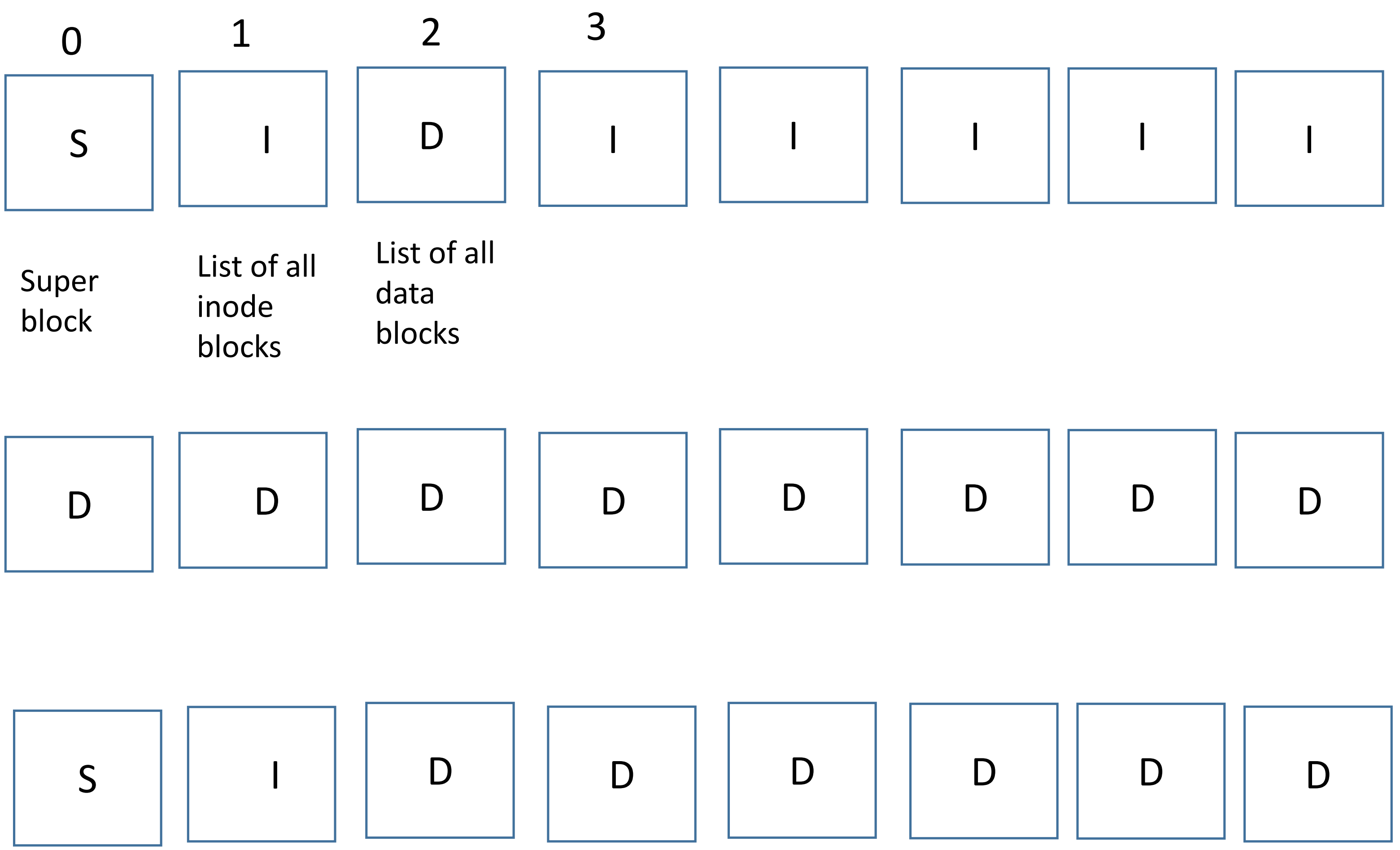
Superblock contents

Each entry is 4 bytes

Magic number
Number of blocks
Number of inodes per block
Number of inodes
Directory inode

Block nbr of Data bitmap
Block nbr of Inode bitmap

Block arrangement



S – superblock
I – Inode block
D – data block

Once a disk is created,
The number of Inodes
And Data blocks
Are fixed.

Suggestion: create a file superblock.py

https://lectures.quantecon.org/py/numpy.html

Use numpy array – more functionality

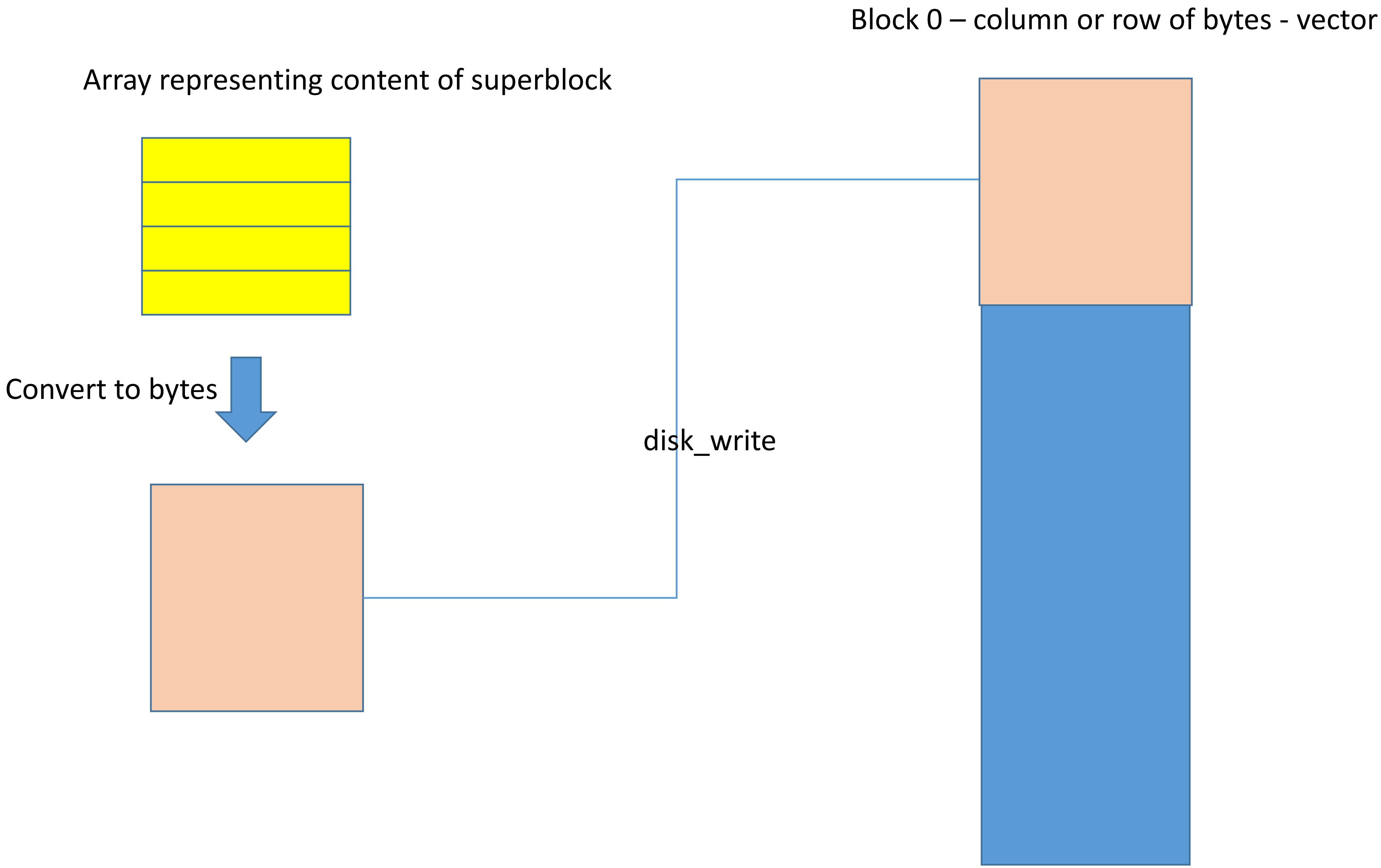
```
number_cells = disk.BLOCK_SIZE // disk.CELL_SIZE
cell_type = 'int32' # which is 4 bytes
```

```
sb_array = np.zeros(shape=(number_cells, 1), dtype=cell_type)
```

To use disk_write()

you need to convert sb_array to an array of bytes

```
sb_array.tobytes()
```


Block 0 – column or row of bytes - vector
Array representing content of superblock
Convert to bytes
disk_write

Superblock – do now

Essential functions:

1. Create superblock
2. Read superblock
3. Write superblock

There are two components:
1. Memory version of superblock
2. Disk version of superblock

For sfs_shell.py

Add new cmds to do the three essential functions

For inodes

- Essential Functions
  1. Init all the inode blocks
  2. Read an inode block
  3. Write an inode block
- Two versions:
  - Memory version of that inode block
  - Disk version

For sfs_shell.py

Add new cmds to do the three essential functions

Valid = 0
Size of file = 0
Datanode1 = 0
Datanode2 = 0
Datanode3 = 0
Datanode4 = 0
Datanode5 = 0
IndirectInode = 0

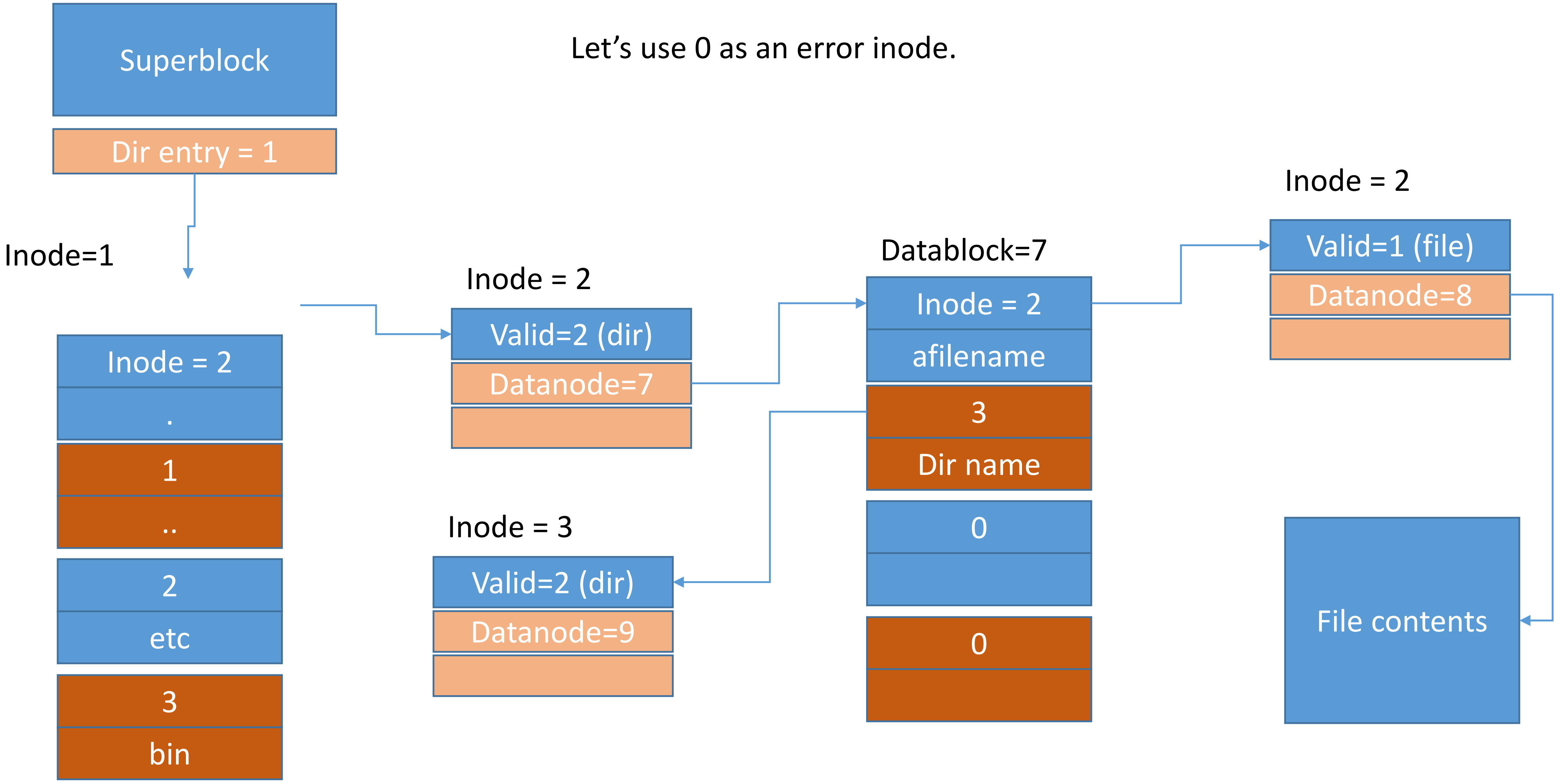
Where to put the “type” information?

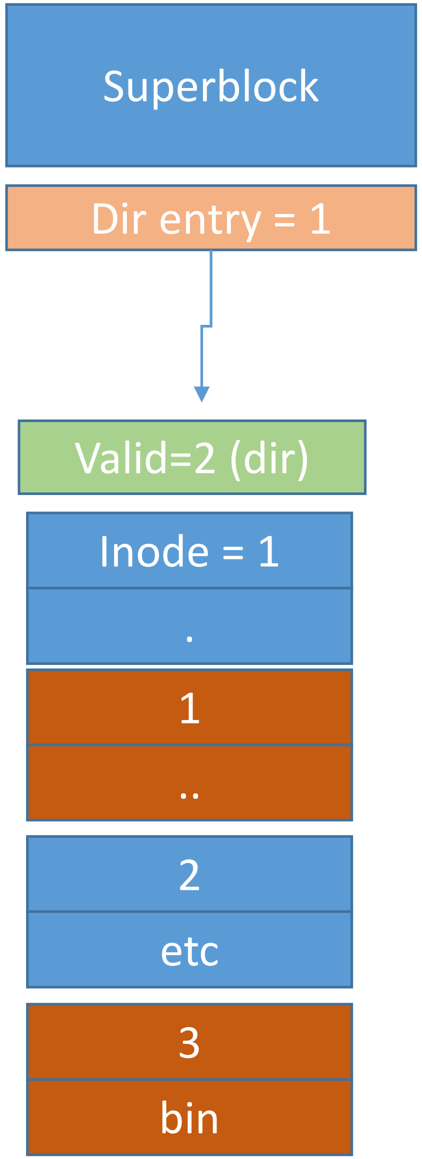
‘Valid’ has 4 bytes. Can’t we put the type information there?

FREE = 0
FILE = 1
DIR = 2
BAD = 99

Will this work?

Let’s use 0 as an error inode.





We still have a problem.

When you do a “dir”…

You are listing the contents of datablock=1

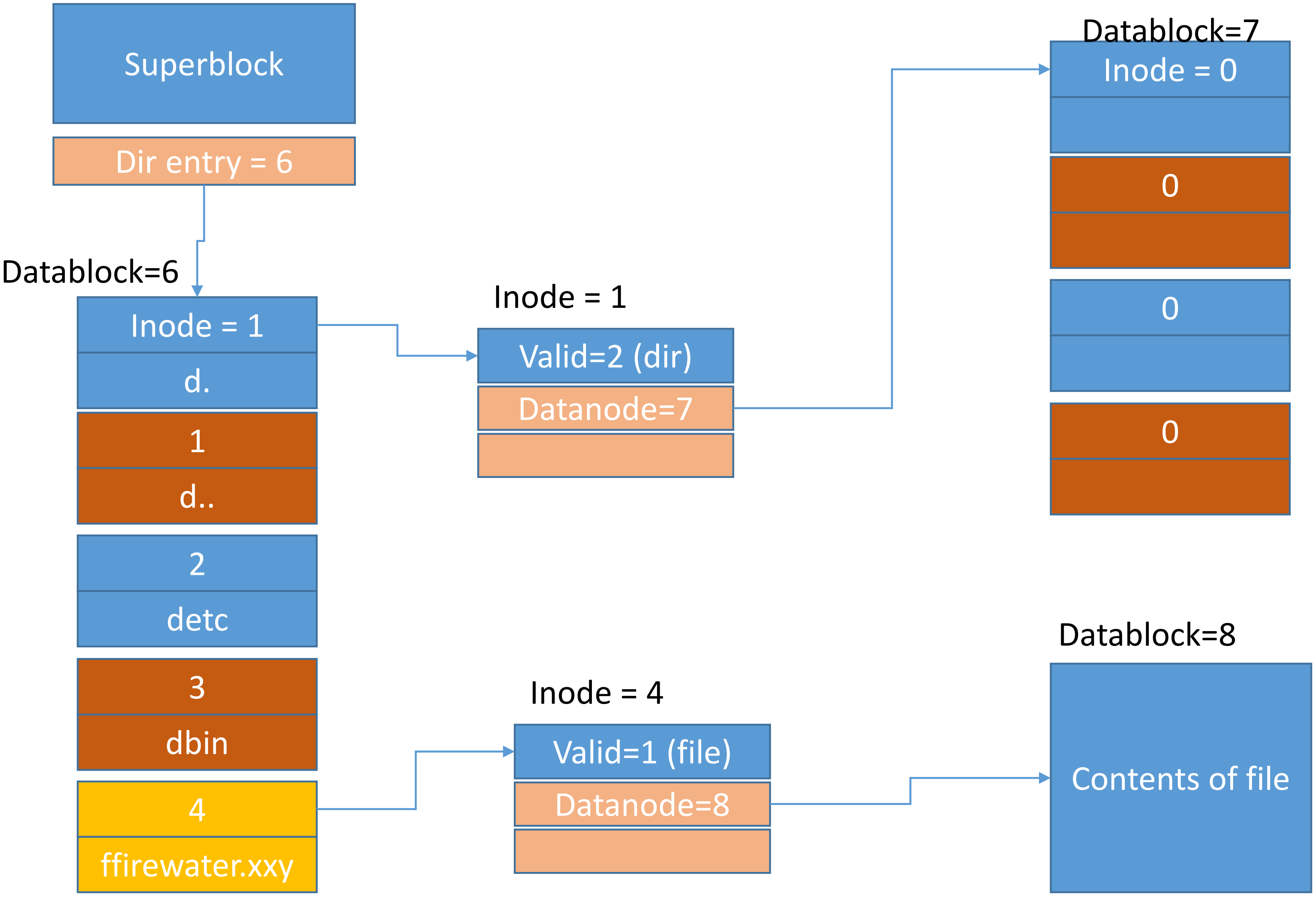
HOWEVER, one cannot differentiate between a directory or a file in that list.

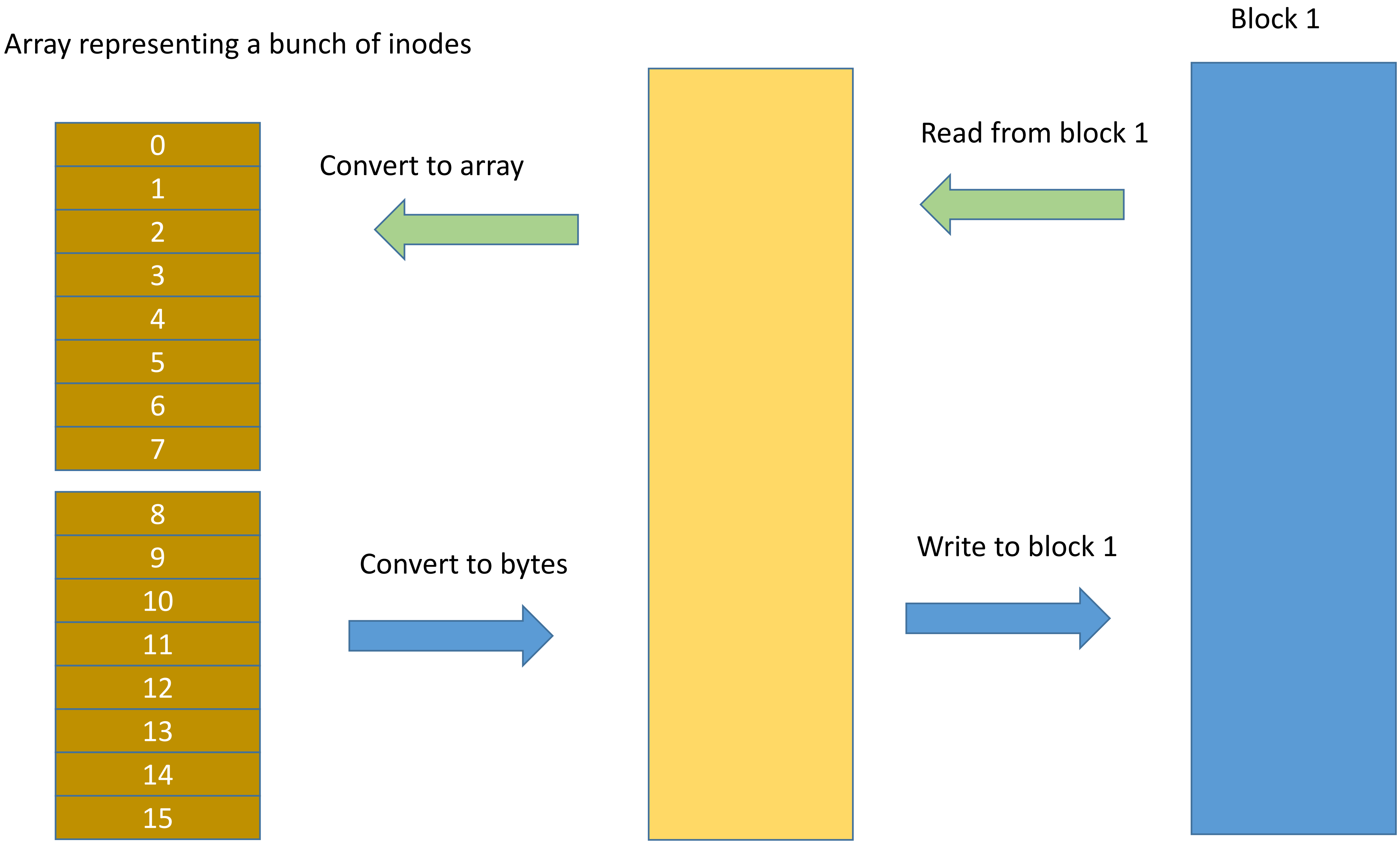
One simple way is to add a character in front of the name

Example:
Instead of ‘.’ it will be ‘d.’
Instead of ‘etc’ it will be ‘detc’.

For a file entry instead of ‘firewater.xxy’,
It will be ‘ffirewater.xxy’.

This saves you
Another tip to
another inode


Superblock
Dir entry = 6
Datablock=6
Inode = 1
d.
1
d..
2
detc
3
dbin
4
ffirewater.xxy
Inode = 1
Valid=2 (dir)
Datanode=7
Inode = 4
Valid=1 (file)
Datanode=8
Datablock=7
Inode = 0
0
0
0
Datablock=8
Contents of file



Array representing a bunch of inodes
Block 1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Convert to array
Read from block 1
Convert to bytes
Write to block 1

For sfs_shell.py

Add new cmds to do the essential functions

For the DataBlock and Inode bitmaps

```
blockBitmap = np.zeros(shape=(arraysize, 1), dtype='int8')
```

EXAMPLE:

Blocksize = 512 bytes

arraysize = 512

BUT, your disk has only 64 blocks, therefore this arraysize is too large.
BUT, you need to write a block back to disk.

SO, have the 512 size array, but set the rest beyond 64 as 'BAD'.

Four states:
FREE = 0
USED = 1
DIR = 2
BAD = 99

Essential Functions:

- Initialize bitmaps for data and inode blocks
- Read and write bitmaps

Usually the datablock bit or just block bitmap,
Is stored in memory.

BUT everytime it is updated, one need to store it
On disk.

FIRST, we need to build out this block bitmap

We need to scan every inode from all the inode blocks,

And mark the block bitmap accordingly to reflect what we find in the inodes.

REMEMBER we assume that the inodes are all good.

Conceptualize the disk
Structures and their use

Block	
0	Superblock
1	Datablock bitmap
2	Inode bitmap
3	Inode block
4	Inode block
5	Inode block
6	Data block
7	Data block
8	Data block

Superblock template

4 bytes

- Magic number = 123456
- Number of blocks = 64
- Number of inodes per block = 32
- Number of inodes = 3

- Directory inode (dentry) = 0
- DataBlock bitmap = 1
- Inode bitmap = 2
- First datablock = 6
- First inode block = 3

May not be Used depending On your implementation

Create a directory (mkdir)

- Function to create a directory
  - mkdir – create one directory

- createInitialDirectoryStructure
  - Create the initial directory structure
    - .
    - ..
    - etc
    - bin

def createInitialDirectoryStructure():

1. Get a free datablock page
2. make this datablock page into a directory entry page
3. Update superblock with the new directory entry block
4. Create a directory entry for '.'
5. Create a directory entry for '..'
6. Create a directory entry for 'etc'
7. Create a directory entry for 'bin'

Format – initialize the whole disk

- Two things are required now
  1. Build the superblock
  2. Initialize all the inodes
- Later three more:
  - Build the data block bitmap
  - Build the inode bitmap
  - Build the directory map
- For a 512-byte block, and an inode of size 32 bytes, there will be 512/32 = 16 inodes per block
- REMEMBER to add “format” in your shell program

You have to write it to disk
No disk writes – point deduction

Inode

- The number of inodes is decided in the format phase
- Usually, the number of inodes imply max number of files
- Heurestic: One inode for every three data blocks
- Inode contains the following information:
  - File access permissions
  - File types – text, binary
  - Encoding type – asci,utf-8,utf-16,emac, etc
  - Time accounting – creation and access
  - Access accounting – userid
  - Filesize
  - etc

Format – functions required for this to work

1. Create superblock on disk
2. Create inodes on disk
3. Create block bitmap in memory and disk
4. Create inode bitmap in memory and disk
5. Create initial Directory Structure

DUE: Mar 8th, 2019

Expectations for Project SFS – mar 8th 2019

- Cmds in SFS shell to do all the following:
  - Format
  - Create directory - mkdir
  - Create initial directory structure
  - Create initial bitmaps for data blocks and inode blocks
  - Create initial inode blocks
  - Read and Write to inodes, bitmaps, and datablocks
  - Dir function – display the root directory

Zip all files to Moodle

For the diligent: extra functionality

- cd – change directory
- Mkdir – in the current directory
- Display current directory or path
- Create a file