UNIX File System

COS 316: Principles of Computer System Design

Lecture 4

Wyatt Lloyd

Today’s Goals

- Learn specifics of the Unix File System
- See 5 examples of naming within it
- Reason about portability, generality, and isolation
- Get first exposure to layering

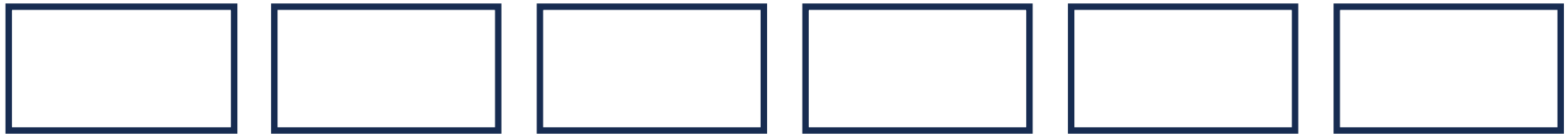
Unix File System

- Unix and its descendants around since 1970s
  - (Named by Prof. Brian Kernighan)
- Descendants:
  - Linux
  - BSD
  - Mac OS X
  - iOS
  - Android

Unix File System

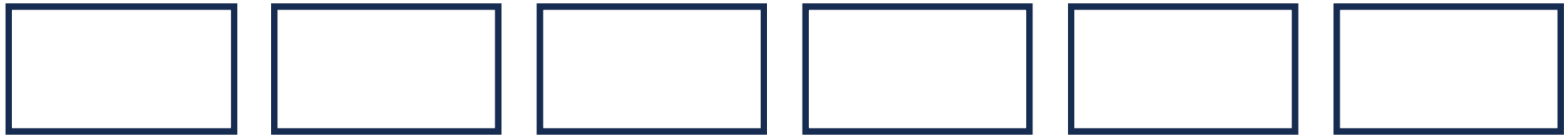
- The UNIX operating system's API has remained relevant since the 1970s
- From "mini"-computers to todays rack-scale servers and personal devices alike!
- The UNIX file system has been even more influential and constant

Disks and What We Want To Do With Them



- Disks provide persistent storage (survives power off) and are “big”

- Want applications to be able to use it without knowing its details
- Want to let users store and organize their data
- Want to let users share data



Key Abstraction

- Data is organized into “files”
  - A linear array of bytes of arbitrary length
  - Meta data about the bytes (modification and creation time, owner, permissions)
- Files organized into “directories”
  - A list of other files or sub-directories
- Common root directory named "/"

UNIX File System Layers

Layer	Purpose
Block	organizes persistent storage into fix-sized blocks
File	organizes blocks into arbitrary-length files
Inode number	names files as uniquely numbered inodes
Directory	human-readable names for files in a directory
Absolute path name	a global root directory

UNIX File System Layers

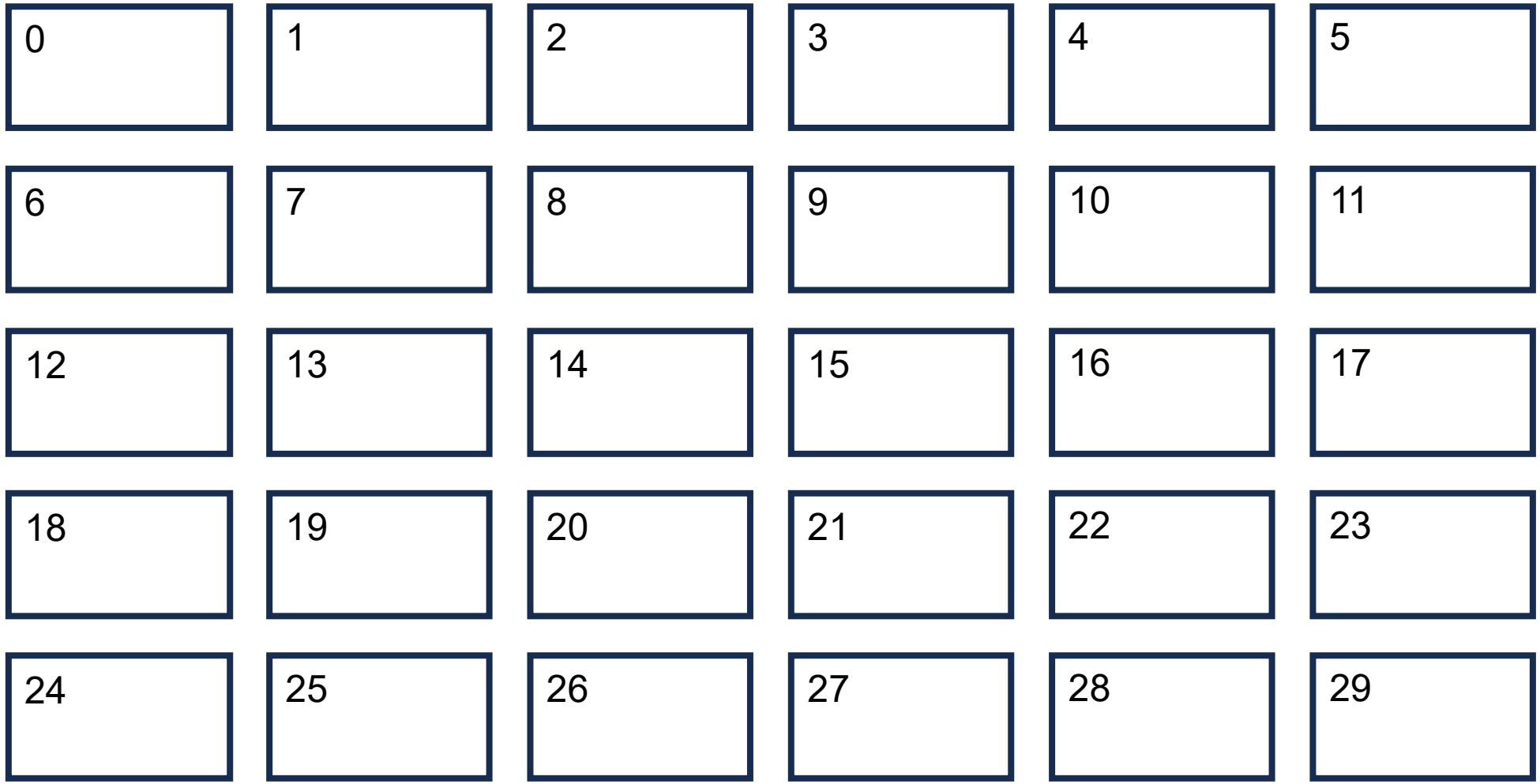
- For each of these we'll look at:
  - Values
  - Names
  - Allocation mechanism
  - Lookup mechanism

- And ask:
  - How portable?
  - How general?
  - Can it isolate?

Block Layer

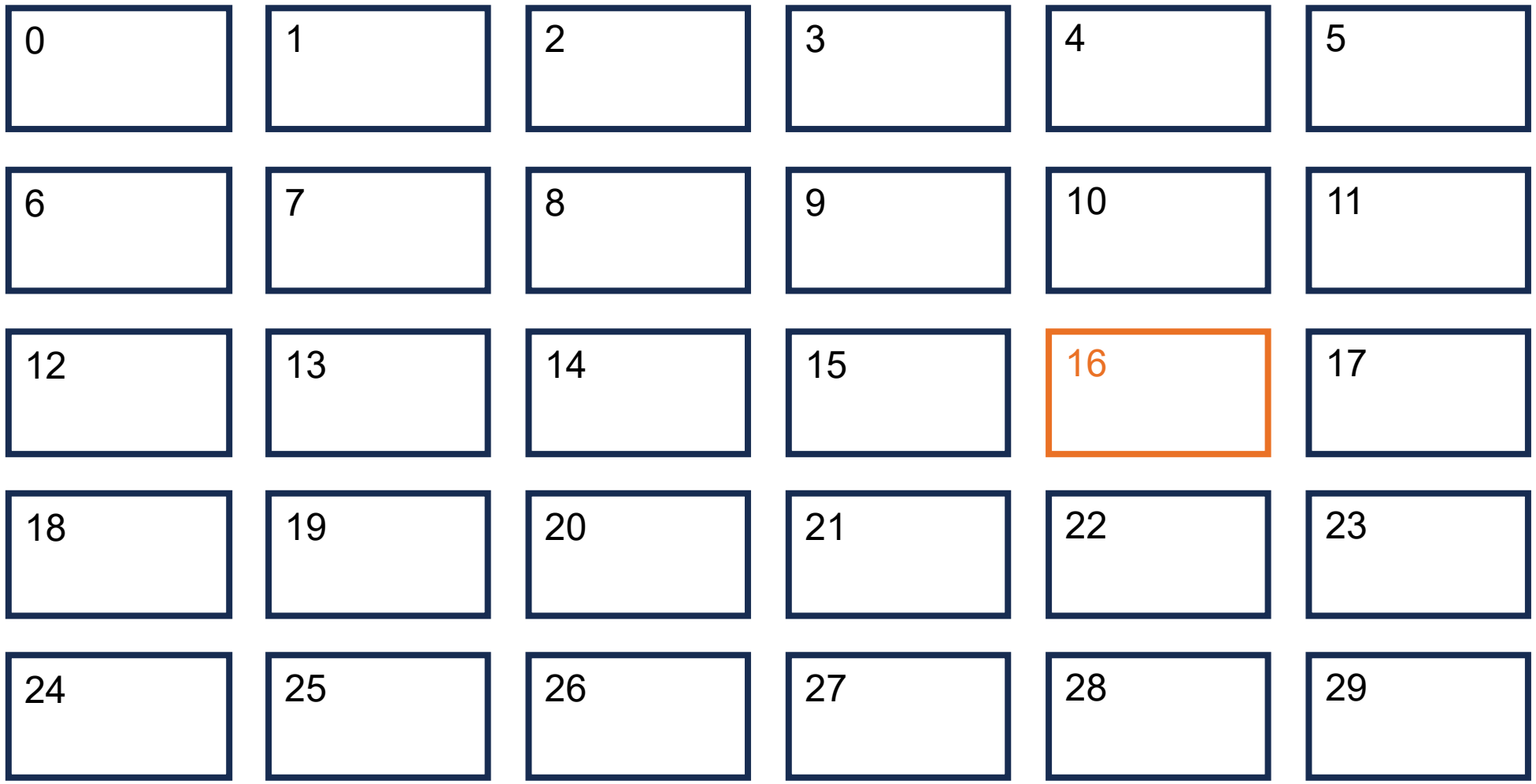
- Underlying resources differ
  - Tape has contiguous magnetic stripe
  - Disk has plates and arms
  - NAND flash (SSDs) even more complex to deal with wear leveling, data striping...
- **Values**: fix-sized “blocks” of contiguous persistent memory
- **Names**: integer block numbers
- **Lookup:** return data stored in block indexed by block number

Block Layer


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

Block Layer

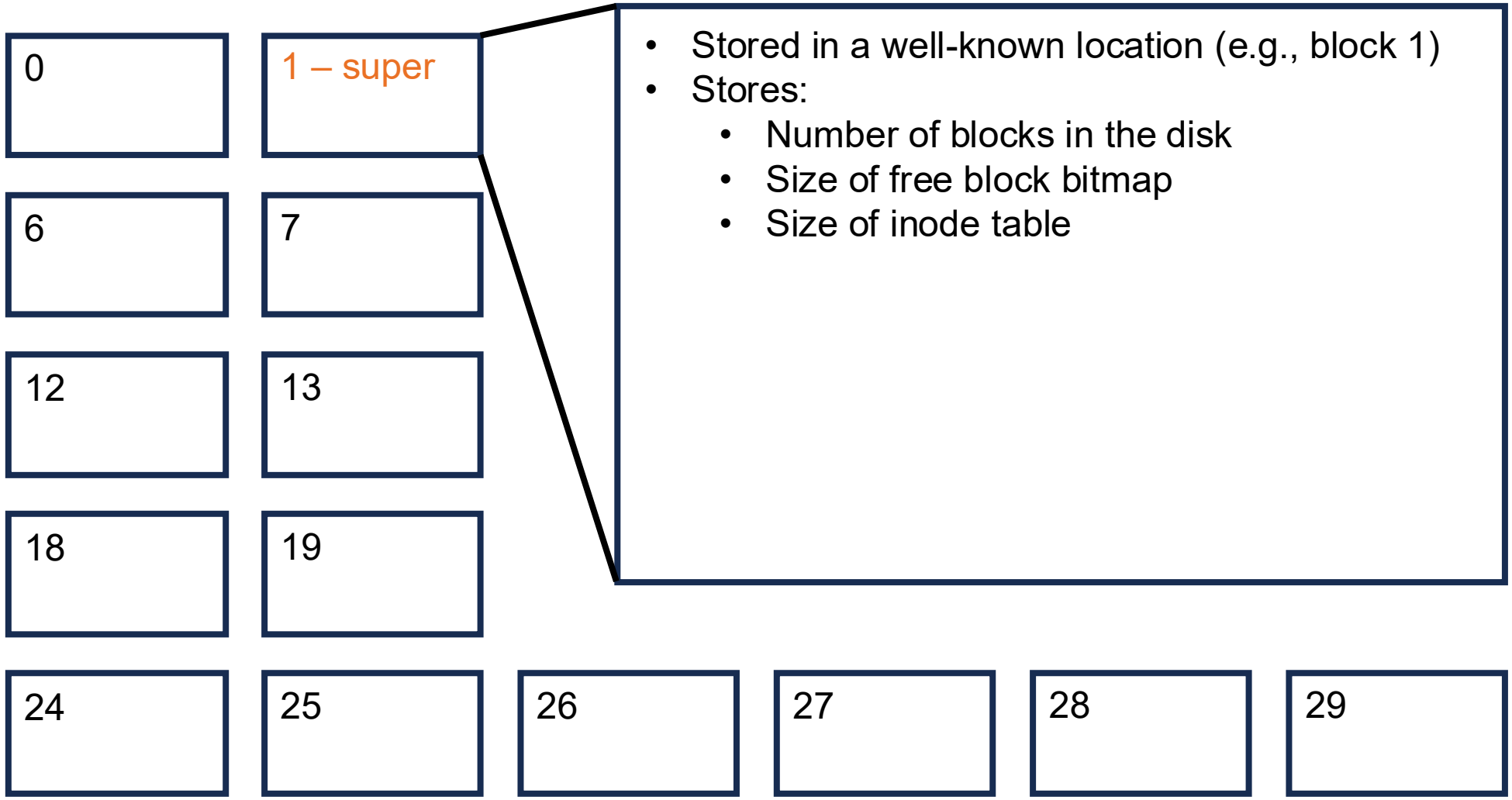
Lookup(16)



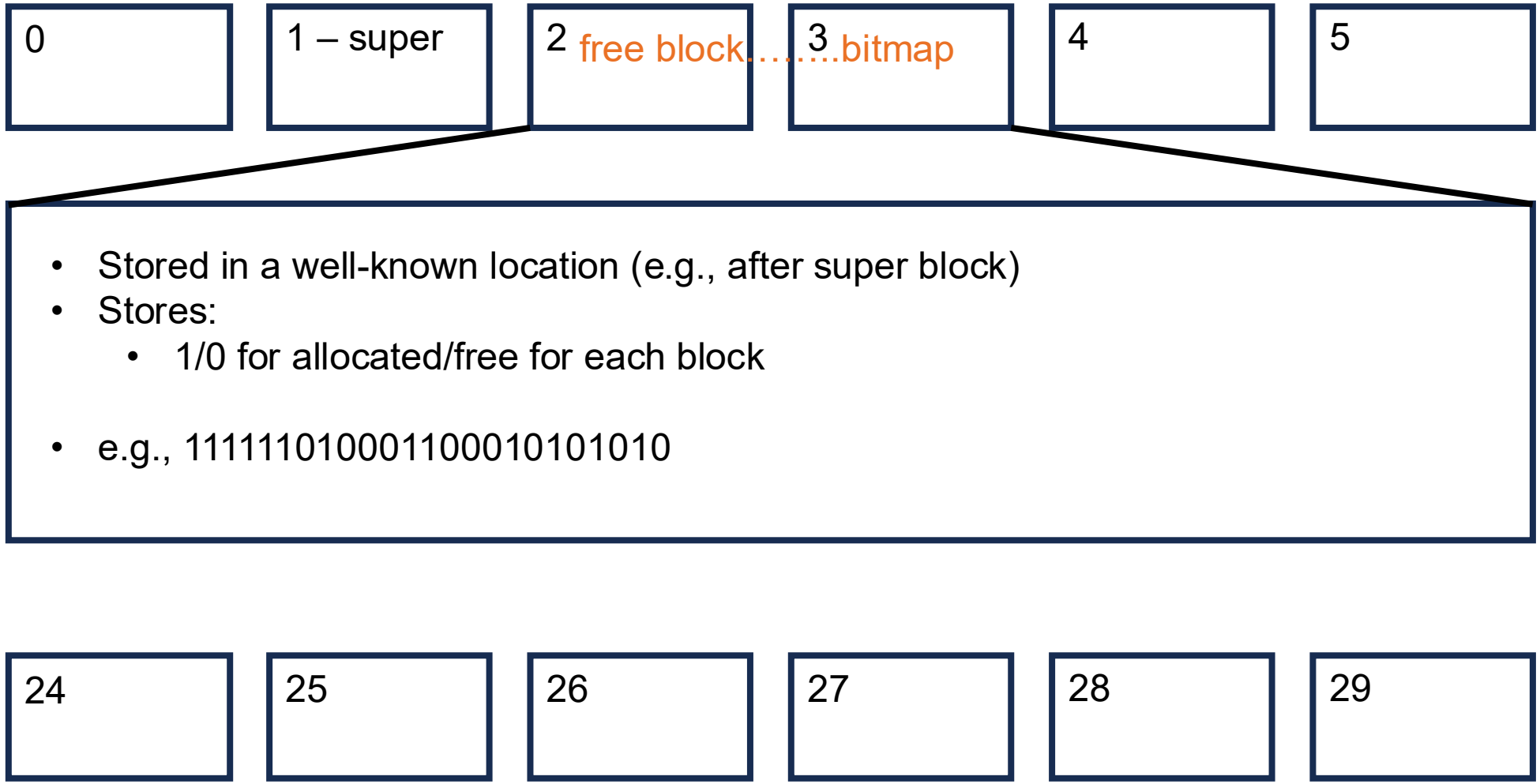
Block Layer: Allocation

- How do I allocate a new block, making it available for writing and reading?
  - Need to find a currently unused block
- Store unused blocks in a data structure in a known location
  - Free block bitmap

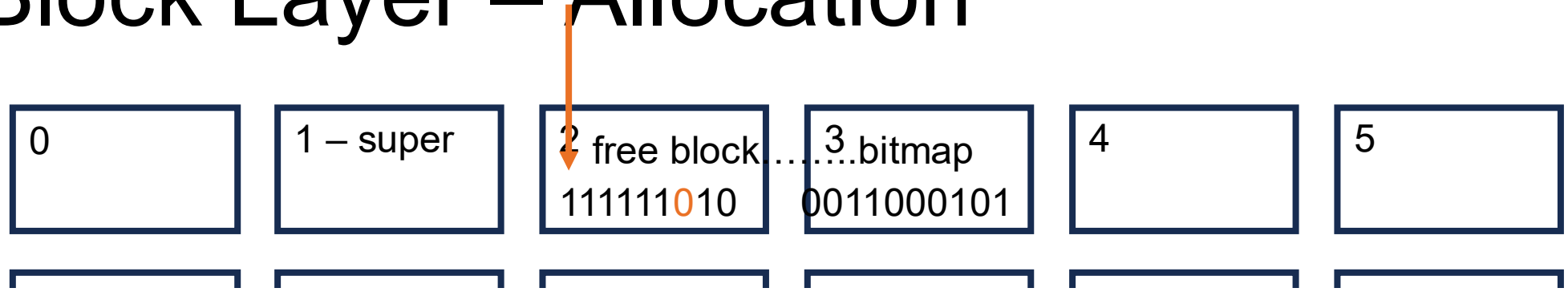
Block Layer – Super Block



Block Layer – Free Block Bitmap

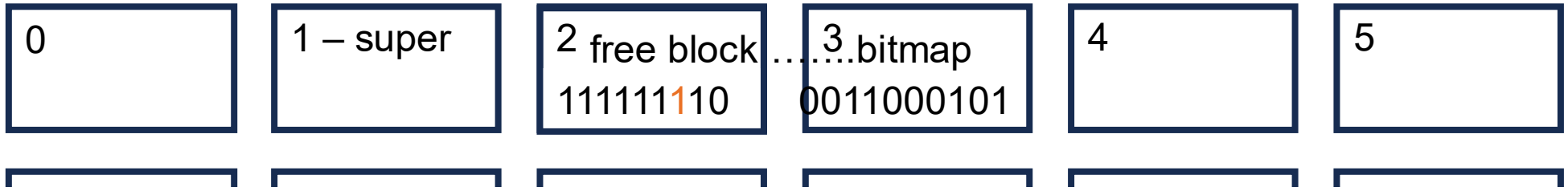


Block Layer – Allocation



- Find the next free block

Block Layer – Allocation



- Find the next free block
- Mark it used
- Return its block number (e.g., 7)

Block Layer: Portable? General? Isolation?

- How portable?
  - Implemented efficiently for most persistent storage media
    - Tape, HDDs, floppy disks, SSDs, ... even network attached storage!

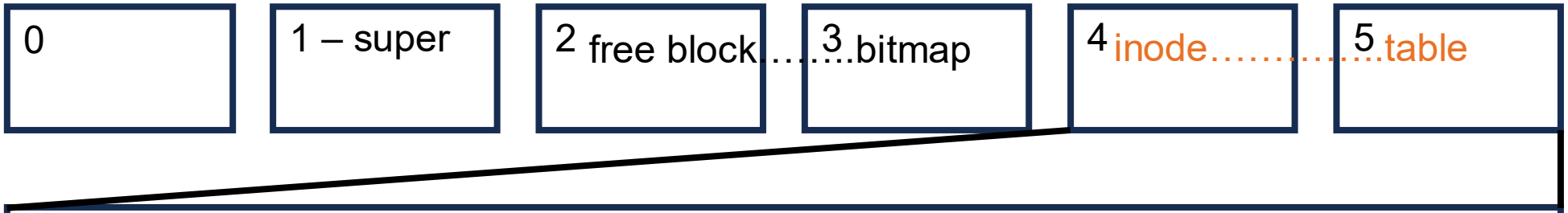
- How general?
  - Lose some expressiveness: block size, performance characteristics
  - But not much

- Isolation?
  - Block numbers are global, they always represent the same physical location

File/Inode Layer

- A file is a linear array of bytes of arbitrary length:
  - May span multiple blocks
  - May grow or shrink over time
- How do we keep track of which blocks belong to which file?
- **Values**: inode struct: { array of block numbers, filesize, … }
- **Names**: inode number
- **Allocation**: Store unused blocks in a data structure in a known location
  - Inode table
- **Lookup:** return inode struct stored in inode table indexed by inode number

File/Inode Layer: Allocation



- Stored in a well-known location (e.g., after free block bitmap)
- Stores:
  - inode structs { blocks[N]; filesize }

- Find an unused inode (e.g., linear scan)
  - mark it used
  - populate it (e.g., filesize 0)
  - return the inode number

File/Inode Layer: Lookup(inode 22)

File/Inode layer: Portable? General? Isolation?

- How portable?
  - Can implement for any block device …
- How general?
  - Applications completely lose locality information
  - Fine for most applications, but not for specific use cases, e.g., databases
- Isolation?
  - A name always refers to particular data, so no inherent isolation

Directory Layer

- Structure files into collections called “directories”. Each file in a directory gets a human readable name—i.e., an ASCII string
  - Directories can contain files as well as other sub-directories

- **Names**: Human readable names within a “directory”
  - resume.docx
  - a.out
  - profile.jpg

- **Values**: Inode numbers

- **Allocation:** Reuse file allocation

- **Lookup:** starting from a working directory, traverse hierarchy of directories

Directory Layer: Allocation

- Directories are a special type of file
  - Indicated by boolean flag to the inode struct
- Allocation reuses file allocation

Directory Layer: File Contents

- A directory file's contents are a simple map from name to inode numbers
  - resume.docs → 27
  - a.out → 14
  - profile.jpg → 92

Directory layer: Portable? General? Isolation?

- How portable?
  - Can implement for any inode & file layer—simply uses file layer for storage
- How general?
  - Assumes a hierarchical structure to file system.
  - Works poorly for relational or structured data
    - “please find all JSON files with the field foo”
- Isolation?
  - All lookups are relative to some base directory!
  - Can isolate applications by giving them different starting points
    - `$: man chroot`

Absolute path name layer

- Each running UNIX program has a “working directory” (wd)
- File lookups are relative to the wd
- What if we want to name files outside of our wd’s directory hierarchy?
  - e.g., share files between users
- What if we want globally meaningful paths?

Absolute path name layer

- Solution:
  - Special name /, hardcoded to a specific inode number
  - All directories are part of a global file system tree rooted at /
    - the “root” directory

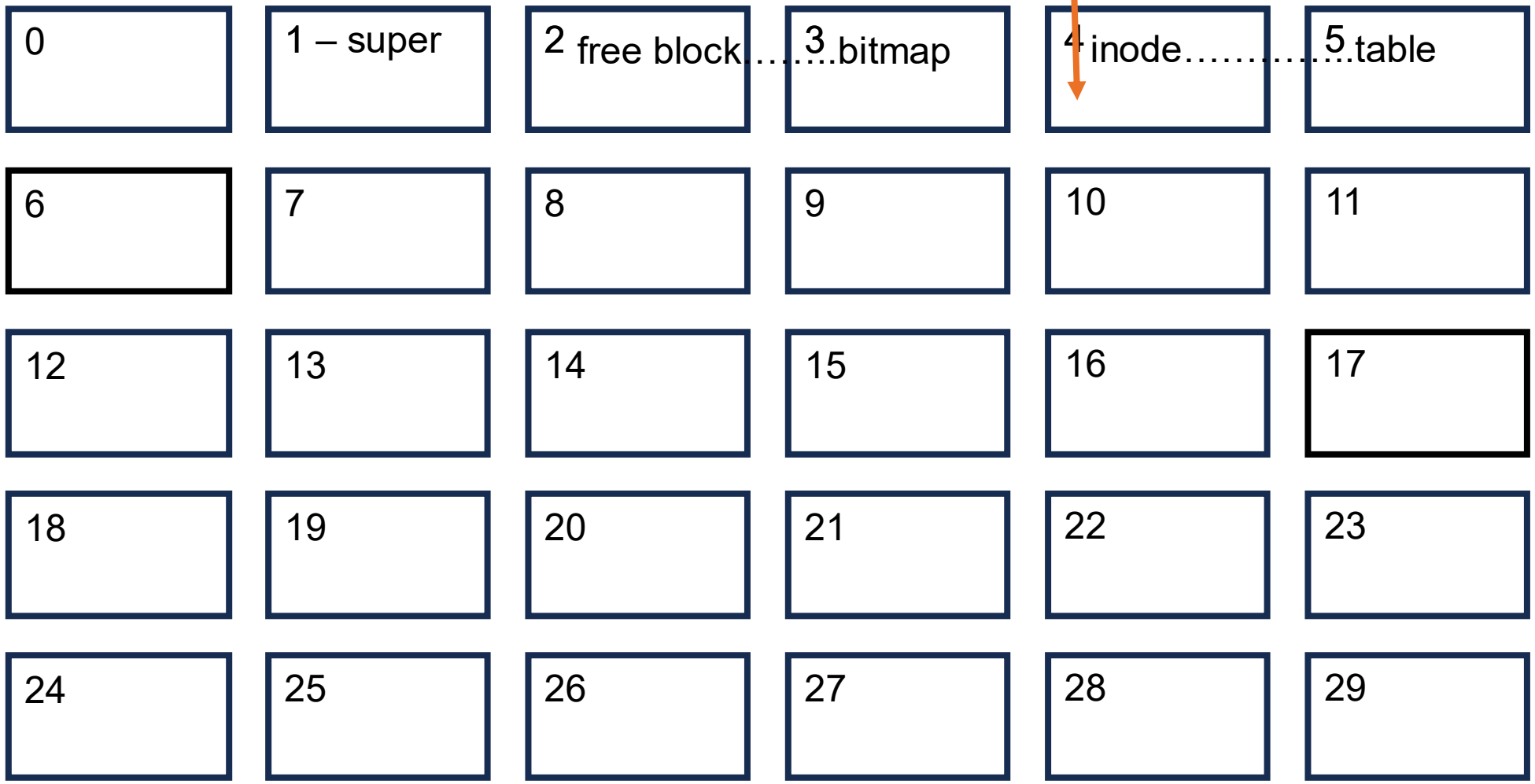
- **Names**: One name, /
- **Values**: Hardcoded inode number, e.g., 1
- **Allocation**: nil
- **Lookup**: return 1

Naming in UNIX File System: Recap

1. Absolute paths translate to paths starting from the “root” directory
2. Paths translate to recursive lookup for human-readable names in each directory
3. Human readable names translate to inode numbers
4. Inode numbers translate to inode structs
5. Inode structs translate to an ordered list of block numbers
6. Block numbers translate to blocks—the actual file data

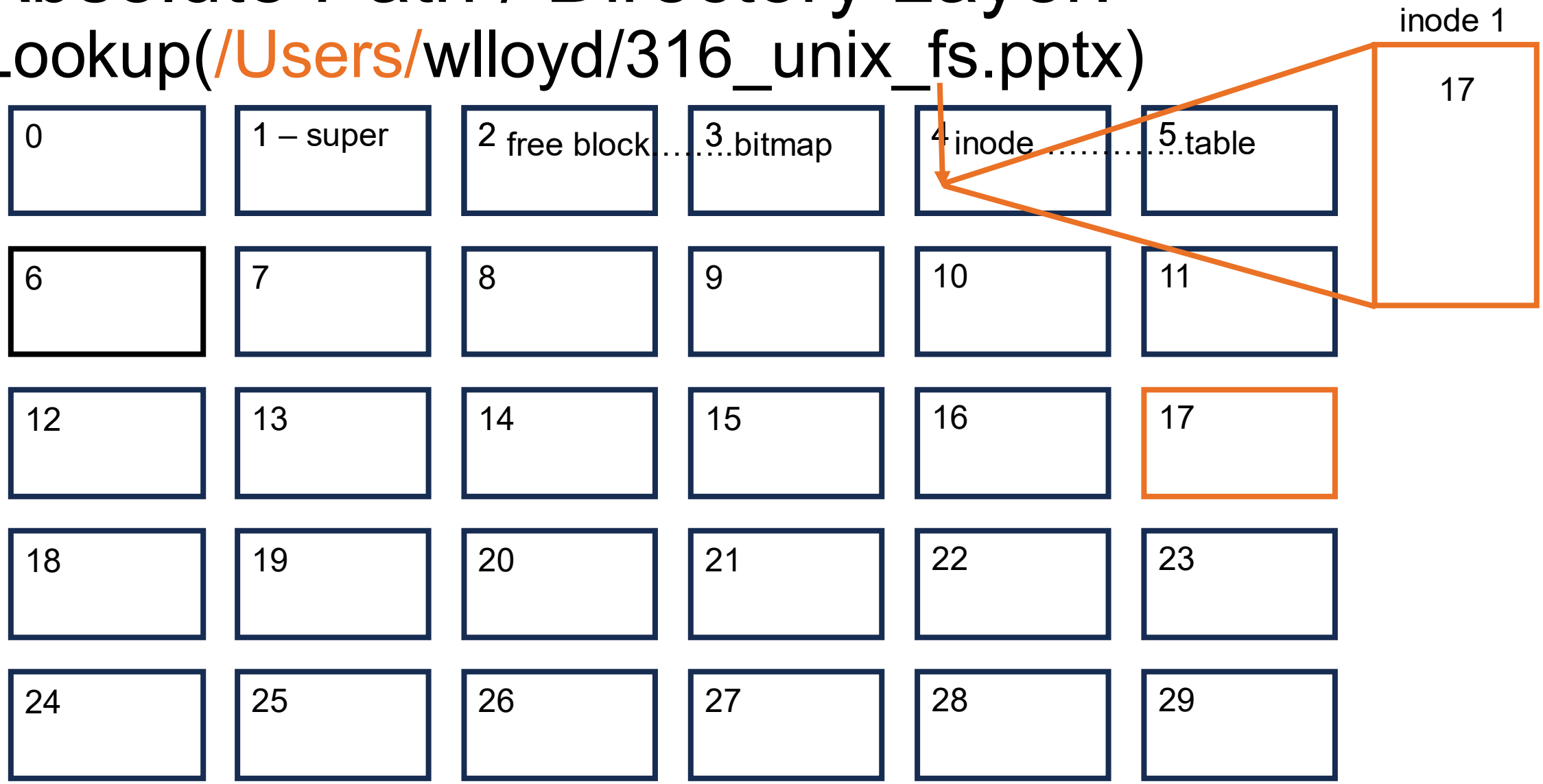
Absolute Path / Directory Layer:

Lookup(/Users/wlloyd/316_unix_fs.pptx)



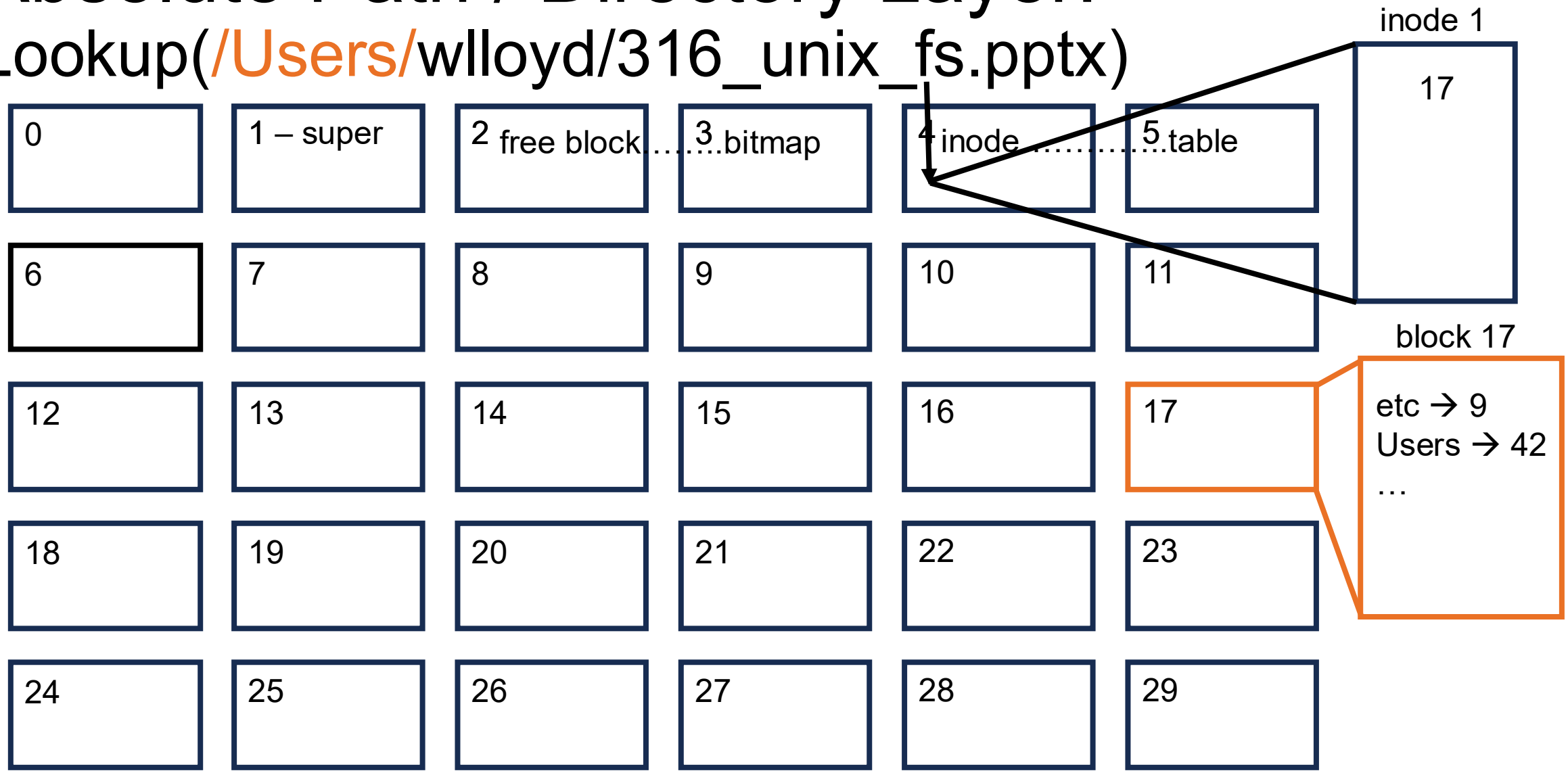
Absolute Path / Directory Layer:

Lookup(/Users/wlloyd/316_unix_fs.pptx)

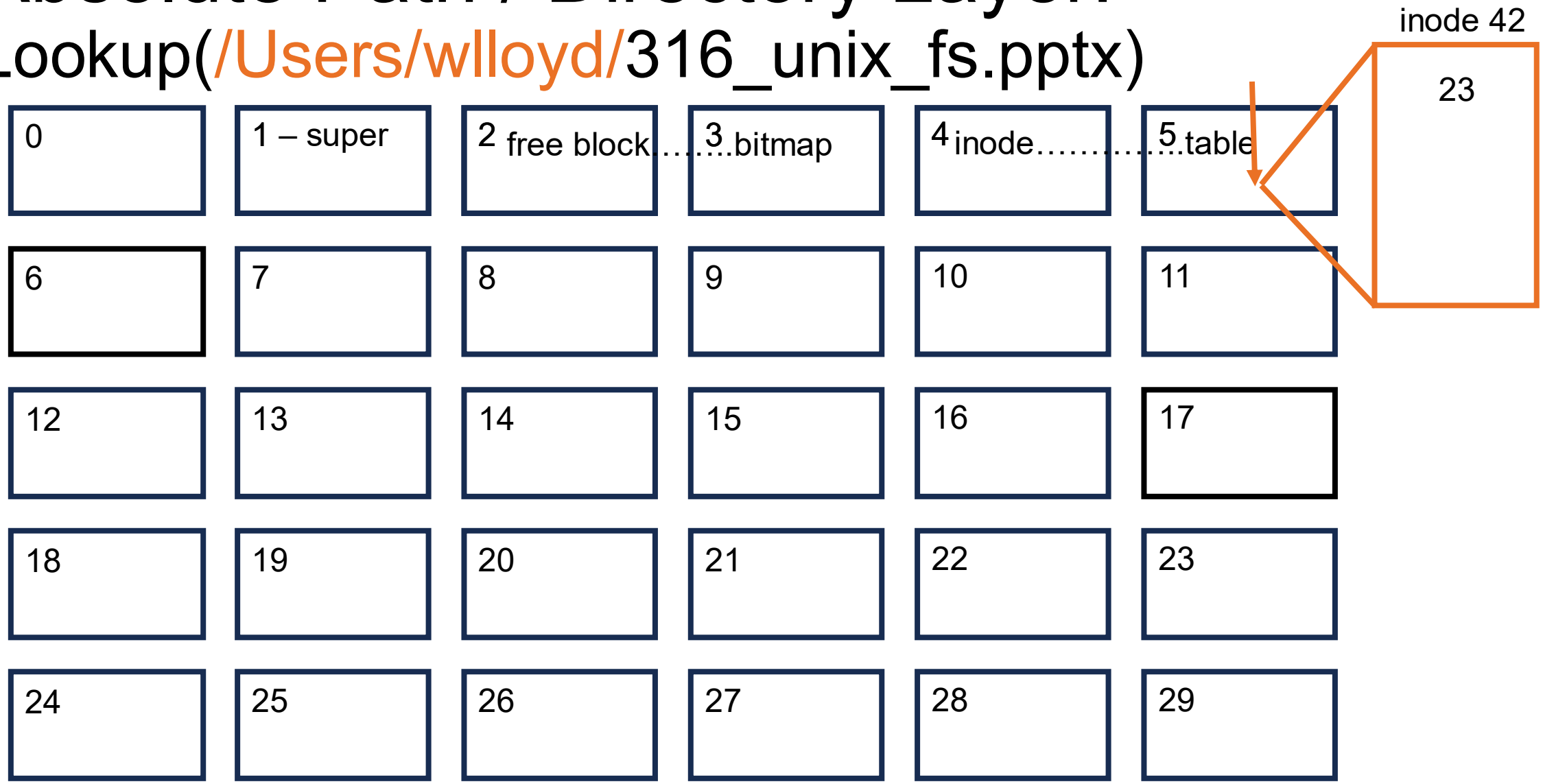


Absolute Path / Directory Layer:

Lookup(/Users/wlloyd/316_unix_fs.pptx)

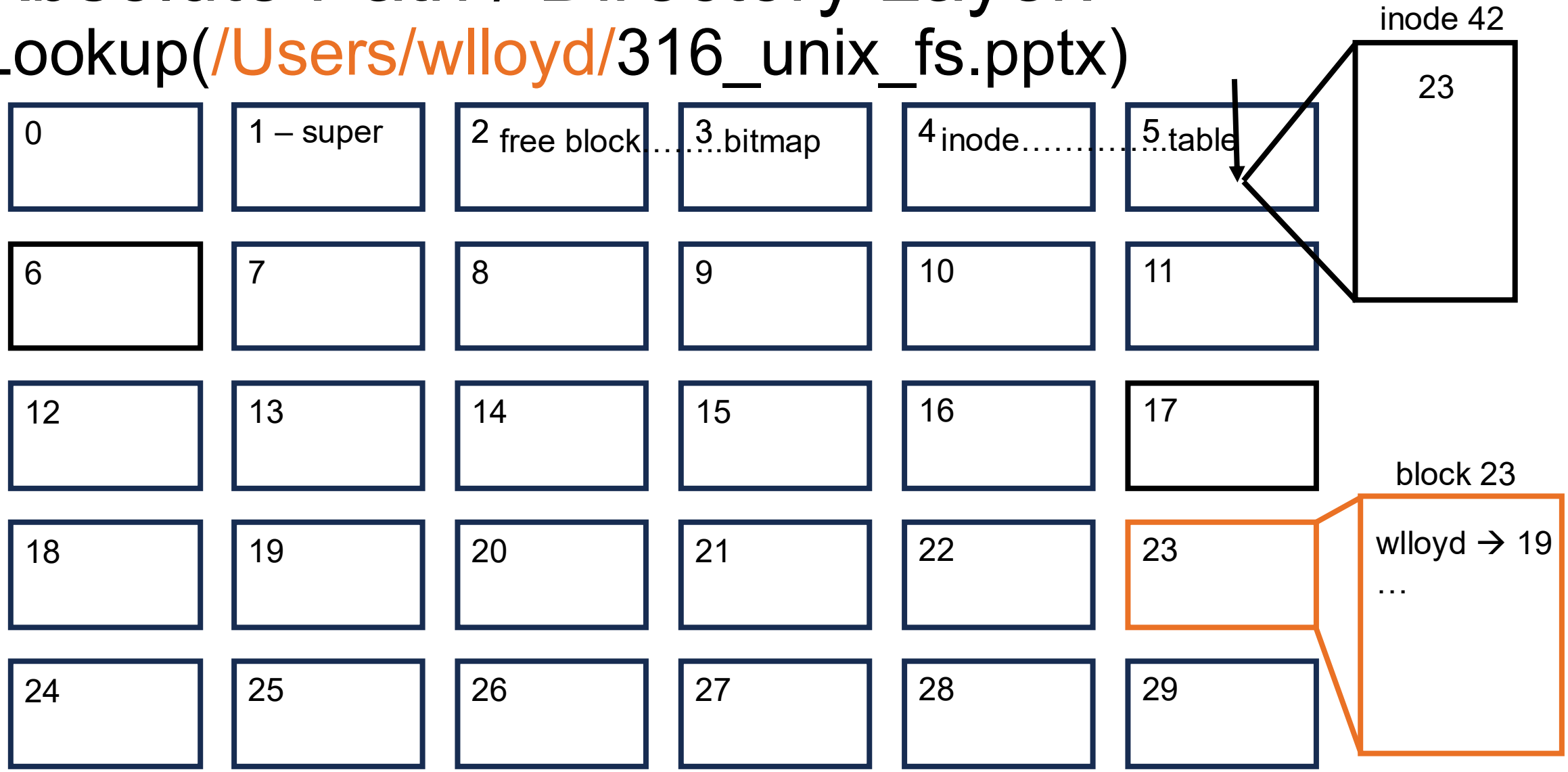


Absolute Path / Directory Layer:
Lookup(/Users/wlloyd/316_unix_fs.pptx)



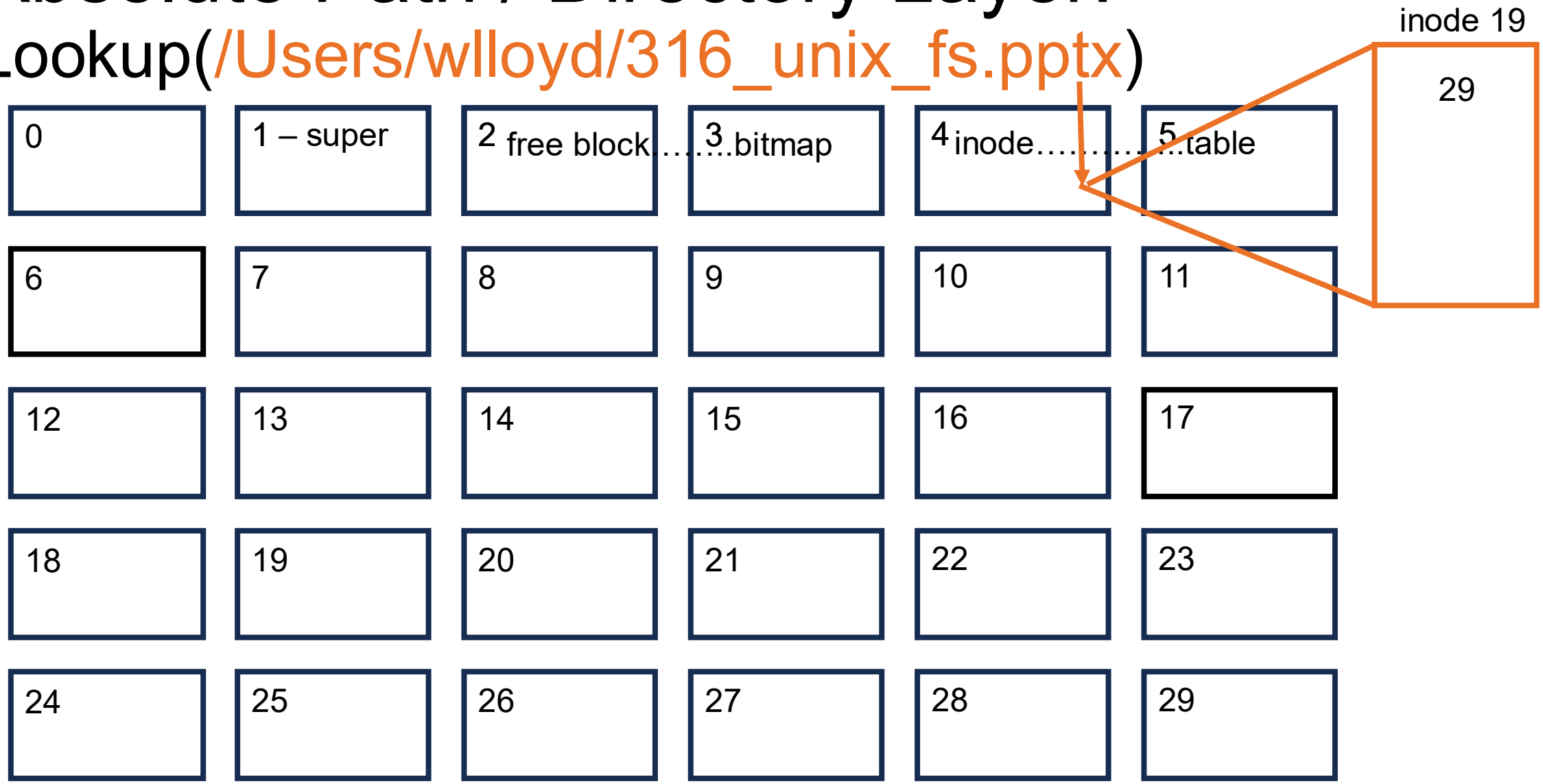
Absolute Path / Directory Layer:

Lookup(/Users/wlloyd/316_unix_fs.pptx)



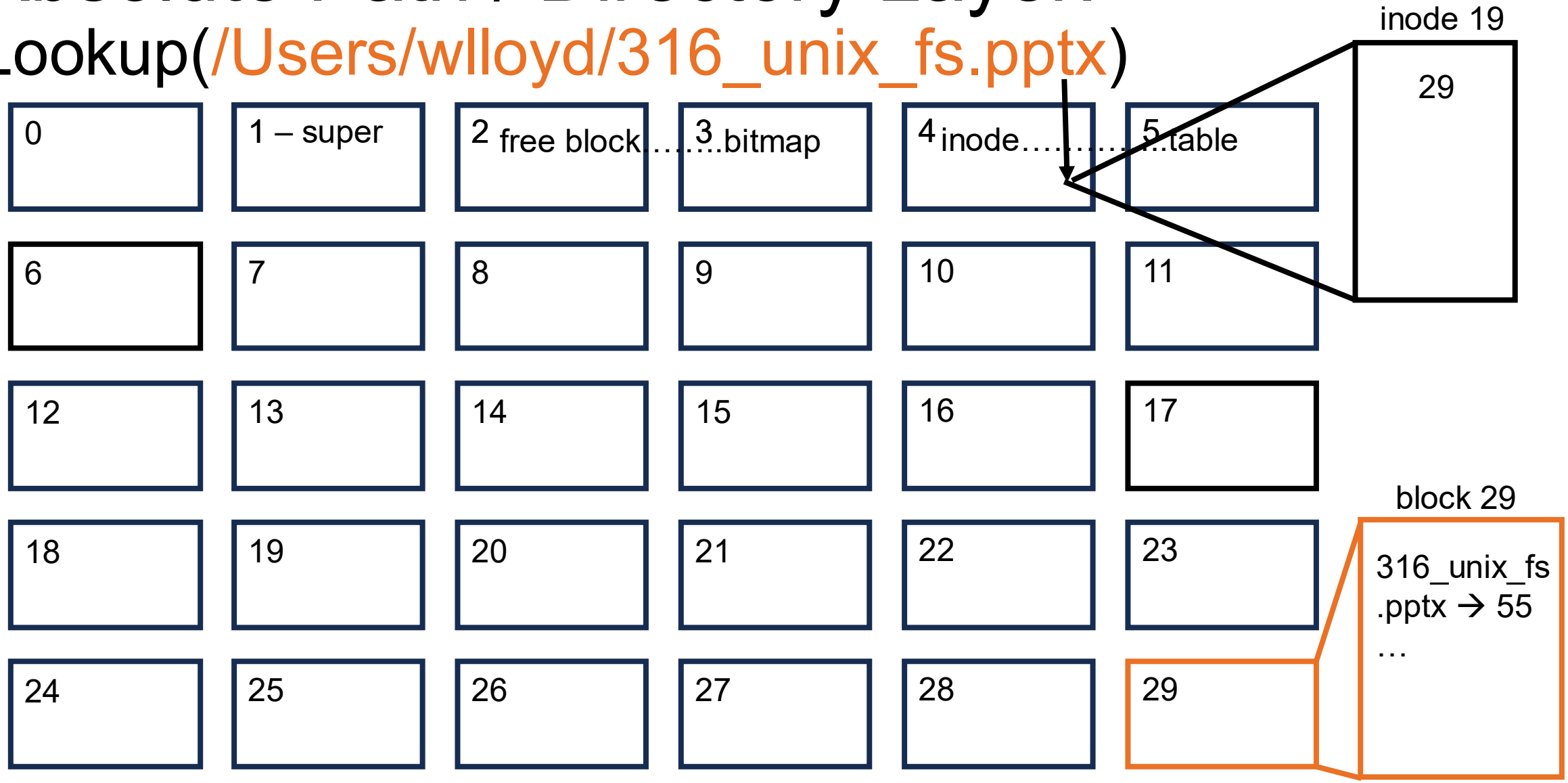
Absolute Path / Directory Layer:

Lookup(/Users/wlloyd/316_unix_fs.pptx)



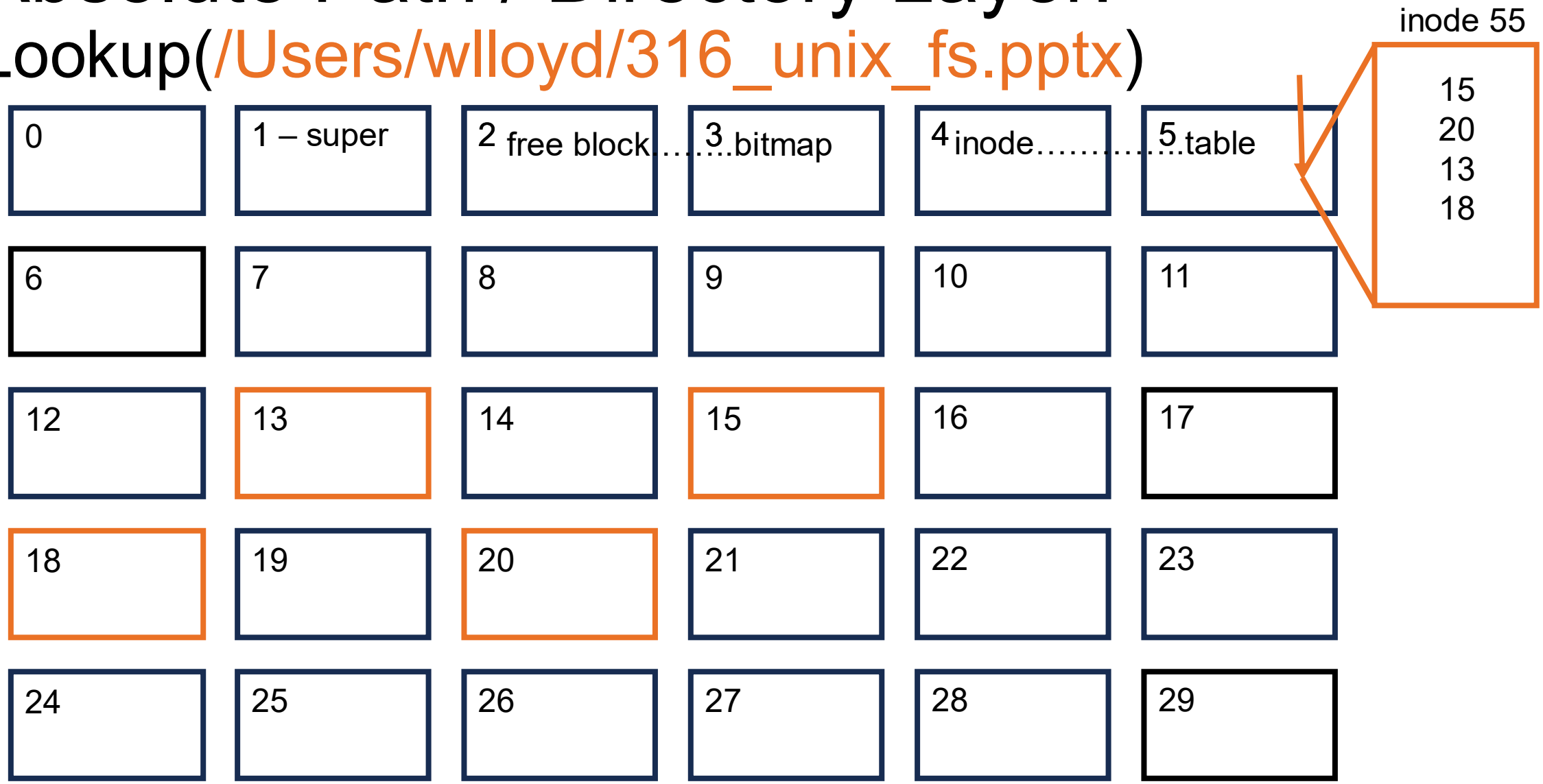
Absolute Path / Directory Layer:

Lookup(/Users/wlloyd/316_unix_fs.pptx)



Absolute Path / Directory Layer:

Lookup(/Users/wlloyd/316_unix_fs.pptx)



Summary

- Learned specifics of the Unix File System
- Saw 5 examples of naming within it
- Reasoned about portability, generality, and isolation
- Got first exposure to layering