

Directories and Vnodes

**Code Review Reflection
(Procs) and resubmission
due EOD Friday!**

HW3 Programming Partners

Group 1: Ishita and Sadhvi

Group 2: Philippe and
Katherine

Group 3: Livia and Vadym

Group 4: Tian and Anika

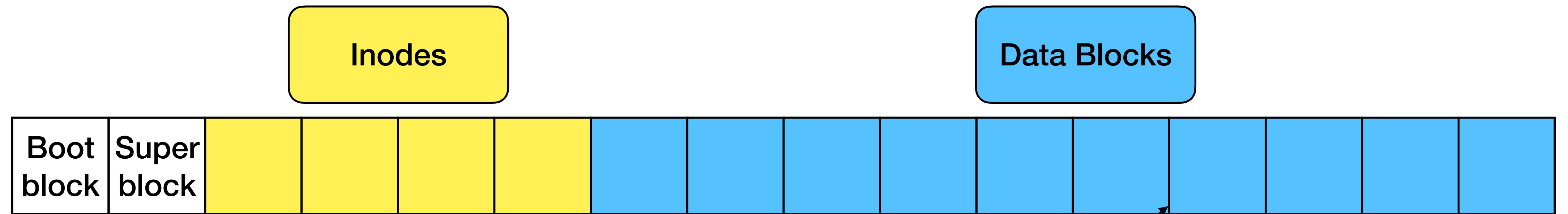
Group 5: Alan and Sudharsan

Group 6: Yaseen, Chengyi,
and Henok

Outline

- Revisiting disk layouts for regular files and directories
- Navigating a file system tree
- Freeing vnodes

Review: System V File System Disk Layout



```
1 /* inode */
2 typedef struct svfs_inode
3 {
4     union {
5         long svi_next_free;
6         long svi_size;
7     } svi_un; // either a pointer to the next free inode or the file size
8
9     long svi_number; // inode number
10    long svi_count; // number of nodes that refer to this inode
11
12    long svi_direct_blocks[SV_NDIRECT_BLKs]; // pointer to direct blocks
13    long svi_indirect_block; // pointer to indirect block
14 } svfs_inode_t;
```

No defined indirect block type! Implied as being a big array of longs referring to other data blocks in the file!

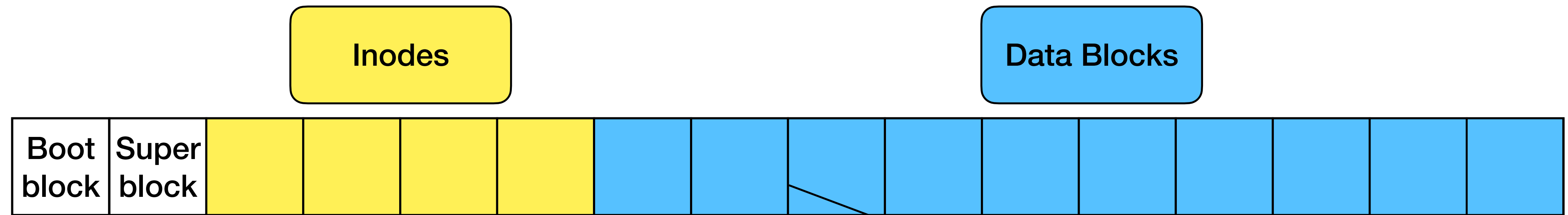
Review: System V File System Disk Layout

- Every file in the file system (regular files and directories) have an inode entry on disk and one or more data blocks on disk
 -  inodes describe file metadata (including where the data blocks are) and data blocks are where file data is stored
- Data blocks often have “special roles” beyond merely storing file data (e.g., acting as indirect blocks)
- For different data block roles, we need to have ways (either explicit or implied) to interpret the data for that role

Chat with your neighbor(s)!

Consider the inode and data block structure that we've described for the System V File System. How can we use this structure to implement directories? Think about the types of blocks that would need to be allocated and the data that goes in those blocks.

Directories in System V File System



```

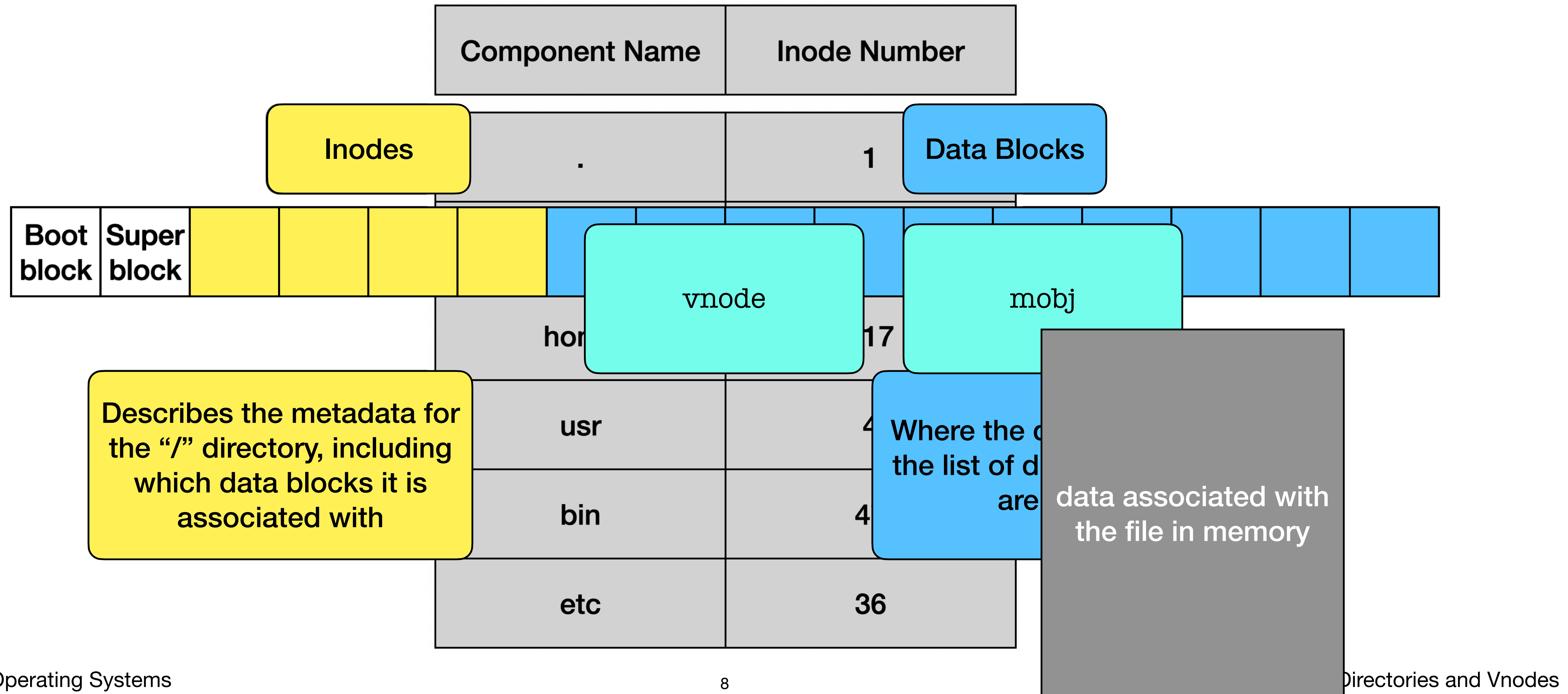
1 /* inode */
2 typedef struct svfs_inode
3 {
4     union {
5         long svi_next_free;
6         long svi_size;
7     } svi_un; // either a pointer to the next free inode or the file size
8
9     long svi_number; // inode number
10    long svi_count; // number of nodes that refer to this inode
11
12    long svi_direct_blocks[SV_NDIRECT_BLKs]; // pointer to direct blocks
13    long svi_indirect_block; // pointer to indirect block
14 } svfs_inode_t;
  
```

```

1 /* directory entry */
2 #define NAME_LEN 24
3
4 typedef struct dirent {
5     long d_ino; // entry inode number */
6     long d_off; // seek pointer of next entry */
7     char d_name[NAME_LEN + 1]; /* filename */
8 } dirent_t;
  
```

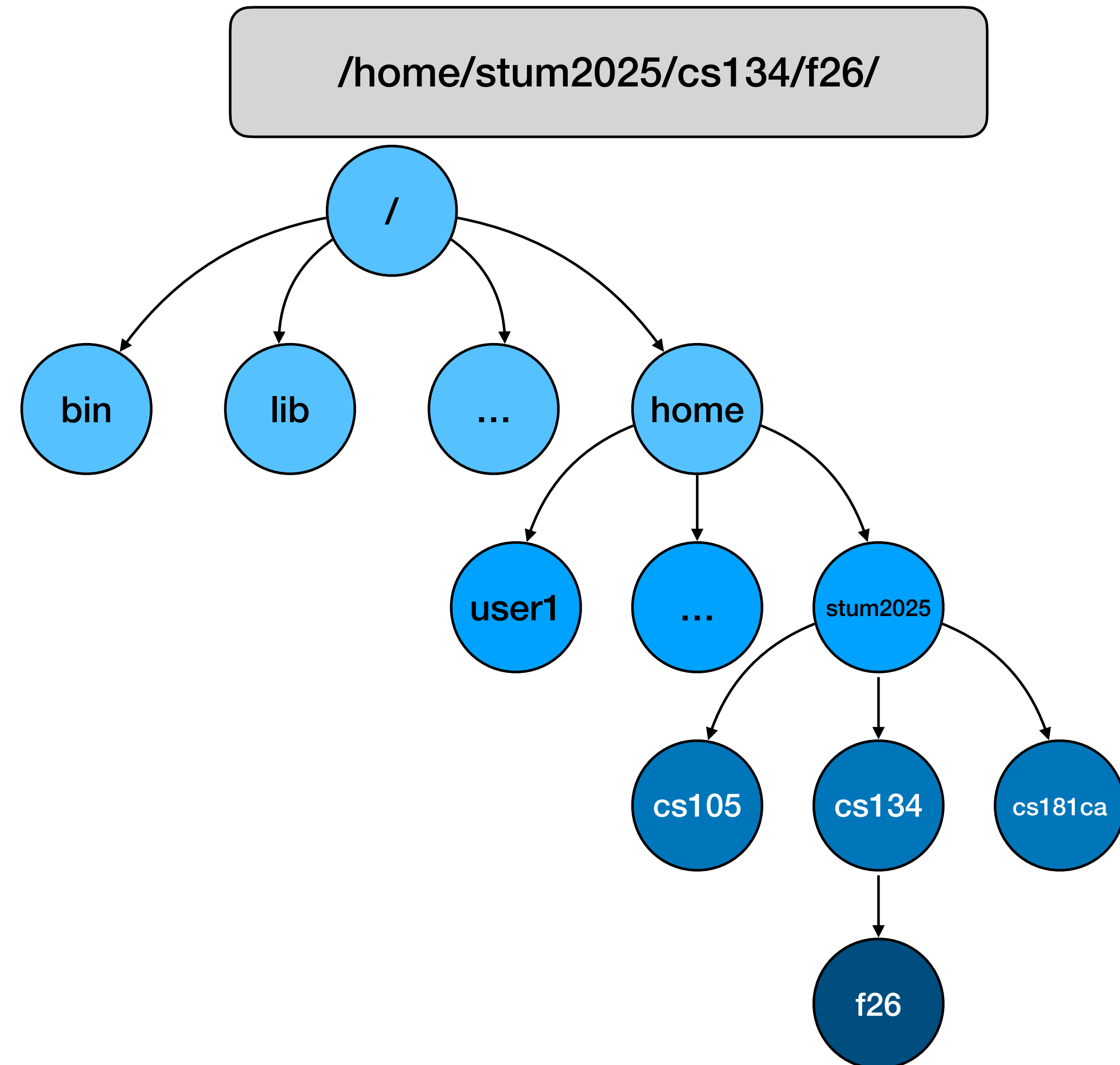
Data block is implied to be interpreted as a list of directory entries, but (by default) we do not have a “directory entry data block” type

Directories in System V File System



File System Tree

- File systems can generally be thought of as a tree-like structure
- In the System V file system, we create the associated vnode with a file on disk every time that we touch it
- When traversing a path, every node in the path is a file that we have accessed, so vnodes are created and added to our OS kernel



Chat with your neighbor(s)!

Traversing directories in file system tree is a huge bottleneck! In 1989, 25% of execution time in the kernel was spent navigating the directory system

Consider our file system implementation, how would we implement user space operations such as “ls”, “cd”, and “rmdir” using vnodes? Think about the types of attributes of the various objects and the how they are stored in the file system overall.

Reading a directory requires getting to the directory and reading the data in its data block from the vnode

Changing directory requires getting to the directory to change to and setting the cwd in the process

Deleting a directory requires changing the data in the parent directory and removing the entry in that data... anything else?

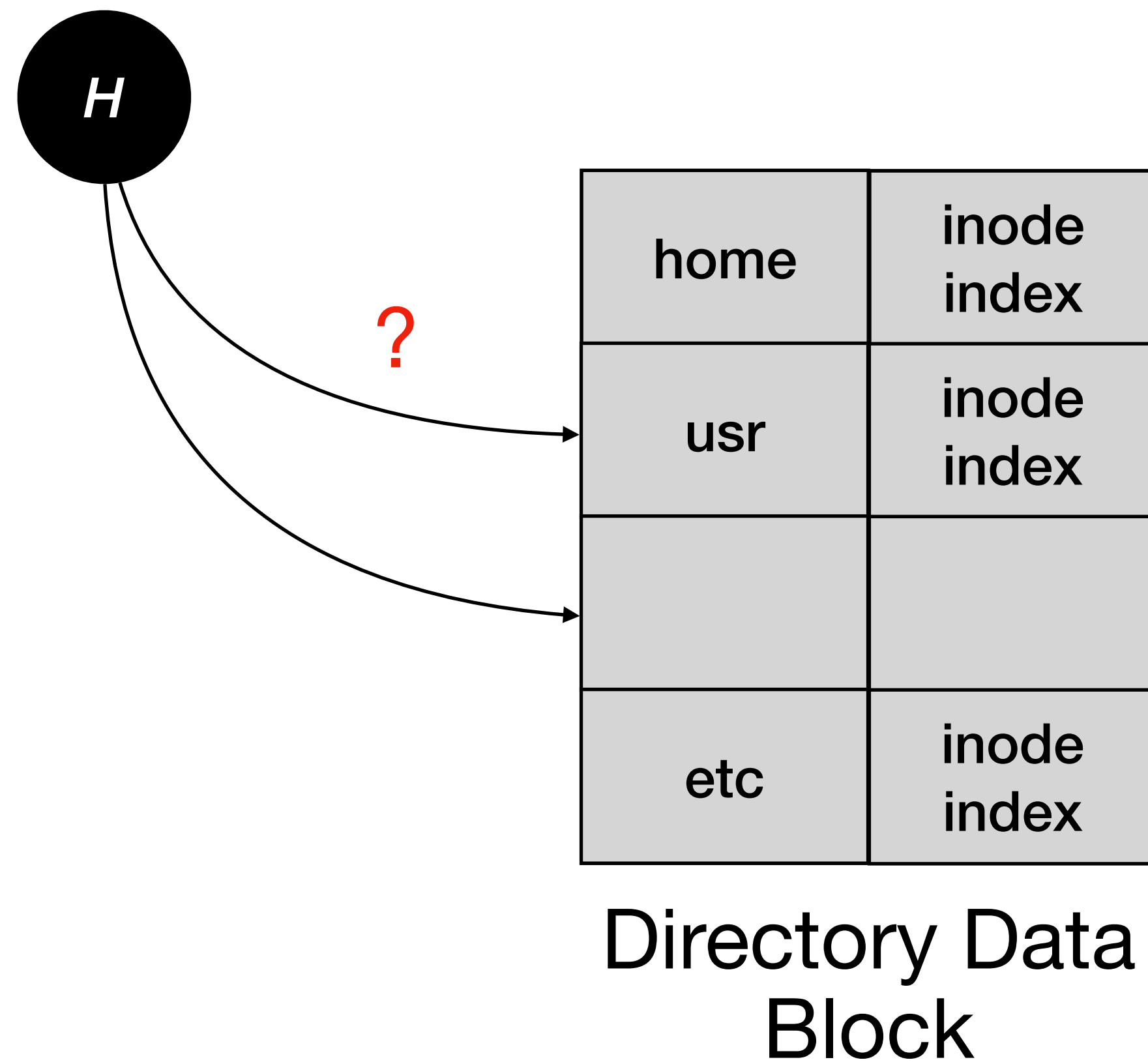
Implementing Directories in the File System

- Using a linear list of entries in a directory will work fine for small file systems (e.g., 10s of entries) but does not scale when we consider large directories (1000s of files per directory) ➡ 25% of OS kernel time spent on directory traversing in BSD 4.2
- If we cache recently used directories (e.g., in the memory of the OS kernel), we can improve performance a little bit (10% of time in BSD 4.3), but a significant amount of time is still spent traversing this structure
- The bottleneck of the system comes from a linear scan of the direct data blocks and the indirect data blocks just to find the inode of the file! If we don't currently have a mobj for the file or the frames are not in memory, this may mean several disk operations!

Chat with your neighbor(s)!

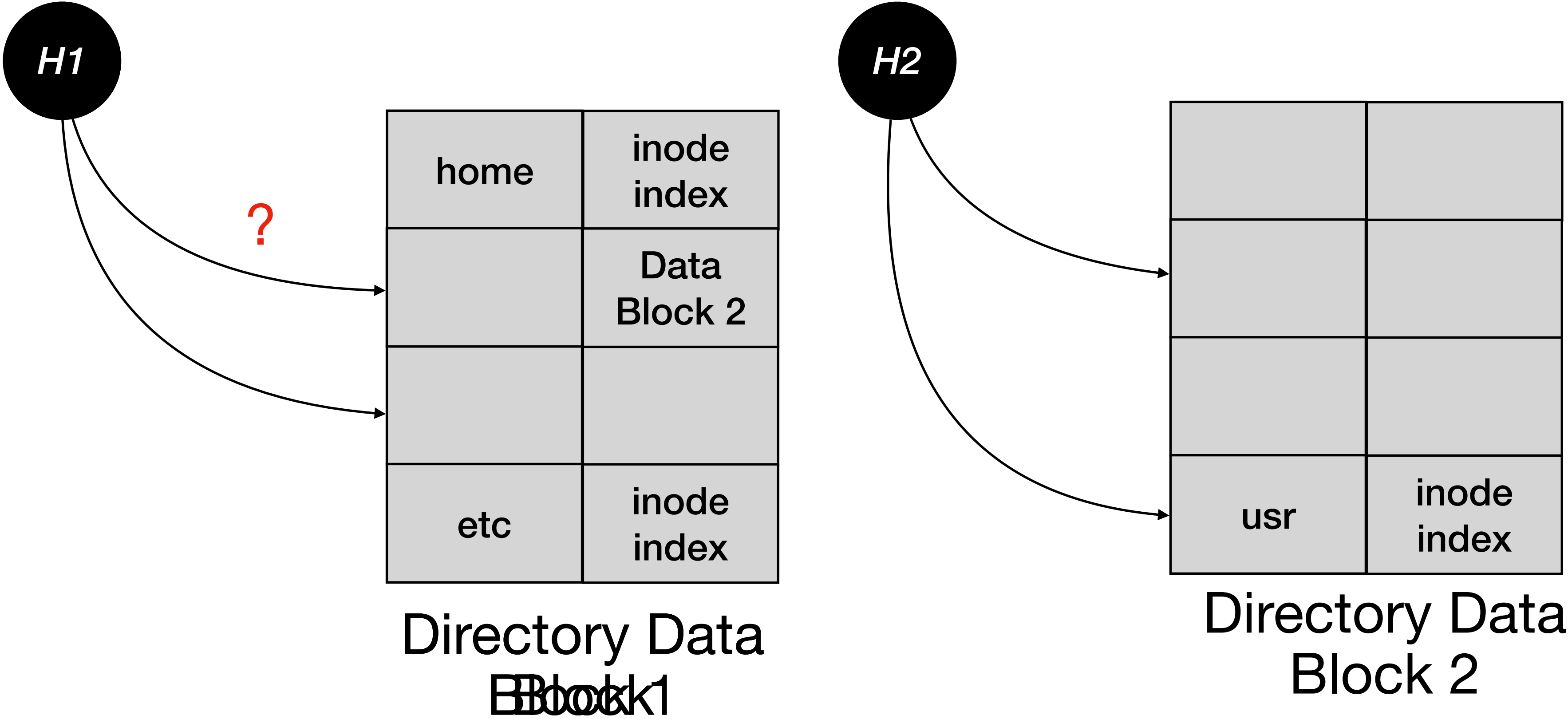
To build an efficient implementation of directory vnodes, we need to do better than System V FS (keep all vnodes in a list and traverse list to find vnode with the equivalent inode index). What alternative approaches could we use?

Implementing Directories in a File System (Hashing)

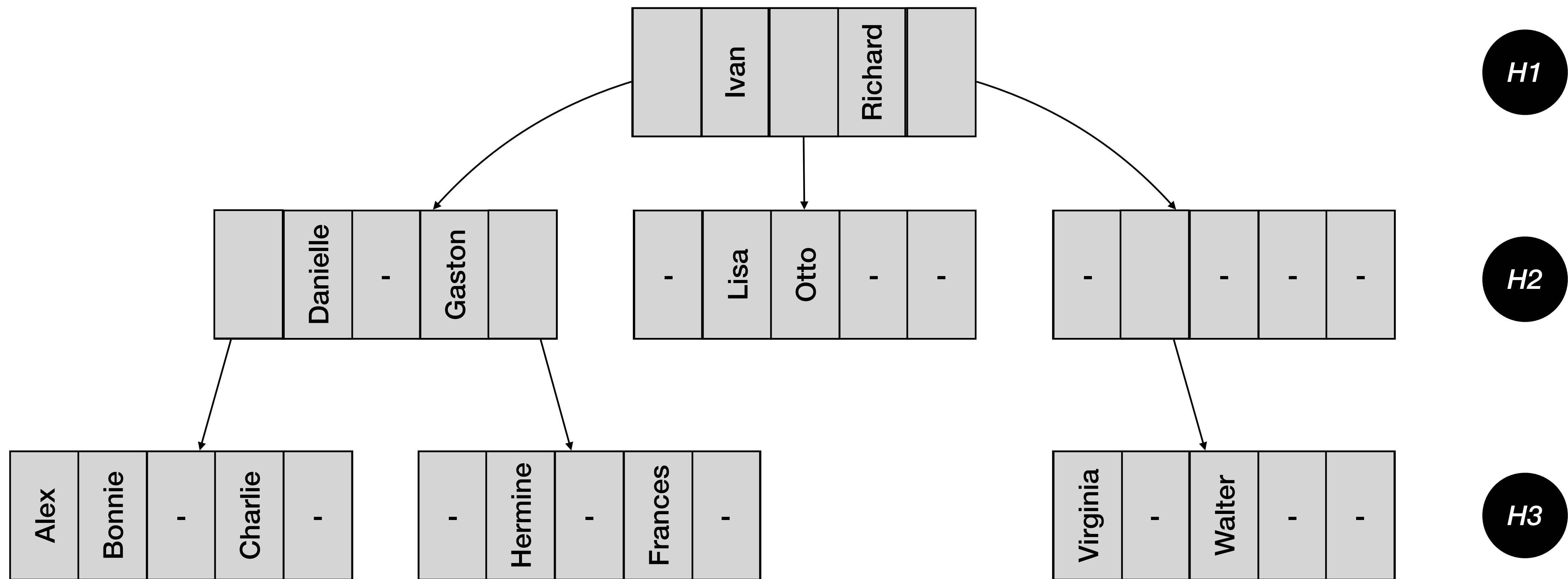


Hashing works well in the event of no collisions, but collisions require expensive operations that can make our system slow again...

Implementing Directories in a File System (B+ Tree)



Implementing Directories in a File System (B+ Tree)



Freeing Vnodes (demo)

```
[stum2025@itbdcv-lnx04p demo11]$ pwd
/home/stum2025/cs134/demos/demo11
[stum2025@itbdcv-lnx04p demo11]$ mkdir temp
[stum2025@itbdcv-lnx04p demo11]$ touch temp/test1; touch temp/test2; touch temp/test3
[stum2025@itbdcv-lnx04p demo11]$ rm -rf temp
[stum2025@itbdcv-lnx04p demo11]$ ls
[stum2025@itbdcv-lnx04p demo11]$ stat /home/stum2025/cs134/demos/demo11/temp
stat: cannot statx '/home/stum2025/cs134/demos/demo11/temp': No such file or directory
[stum2025@itbdcv-lnx04p demo11]$ tat .
-bash: tat: command not found
[stum2025@itbdcv-lnx04p demo11]$ stat .
  File: .
  Size: 6          Blocks: 0          IO Block: 4096   directory
Device: fd02h/64770d  Inode: 889040658  Links: 2
Access: (0755/drwxr-xr-x)  Uid: (989988336/stum2025)   Gid: (989800513/domain users)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2026-10-07 13:54:06.041410641 -0700
Modify: 2026-10-07 13:53:59.099705445 -0700
Change: 2026-10-07 13:53:59.099705445 -0700
 Birth: 2026-10-07 11:16:13.710867842 -0700
[stum2025@itbdcv-lnx04p demo11]$ █
```

```
[stum2025@itbdcv-lnx04p demo11]$ pwd
/home/stum2025/cs134/demos/demo11
[stum2025@itbdcv-lnx04p demo11]$ cd temp/
[stum2025@itbdcv-lnx04p temp]$ ls
test1  test2  test3
[stum2025@itbdcv-lnx04p temp]$ ls
[stum2025@itbdcv-lnx04p temp]$ pwd
/home/stum2025/cs134/demos/demo11/temp
[stum2025@itbdcv-lnx04p temp]$ stat /home/stum2025/cs134/demos/demo11/temp
stat: cannot statx '/home/stum2025/cs134/demos/demo11/temp': No such file or directory
[stum2025@itbdcv-lnx04p temp]$ stat .
  File: .
  Size: 6          Blocks: 0          IO Block: 4096   directory
Device: fd02h/64770d  Inode: 687587473  Links: 0
Access: (0755/drwxr-xr-x)  Uid: (989988336/stum2025)   Gid: (989800513/domain users)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2026-10-07 13:53:43.869158195 -0700
Modify: 2026-10-07 13:53:59.099705445 -0700
Change: 2026-10-07 13:53:59.099705445 -0700
 Birth: 2026-10-07 13:53:30.441794128 -0700
[stum2025@itbdcv-lnx04p temp]$
```

Chat with your neighbor(s)!

To create a `vnode` in the file system, we allocate the space in kernel memory and insert the created structure into the `vnode_list` in the file system. How do we know when a `vnode` can be destroyed? Think about what happens if one or more processes each have a file descriptor pointing to that file.

If a process refers to a `vnode`, we cannot destroy it

To track how many references to a `vnode` there are, the `vnode` itself needs to track a count of “references” pointing to it

When creating an entry in the process file descriptor table, we need to also increment the `vnode` reference count!

Summary

- We described how to navigate the file system tree in terms of directories
- We detailed scalability issues with implementing directories as arrays of directory entries and described how we can fix this issue by using hashes and B+ trees
- We covered a corner case of the vnode lifecycle to provide motivation for how to free vnodes