

File System Software

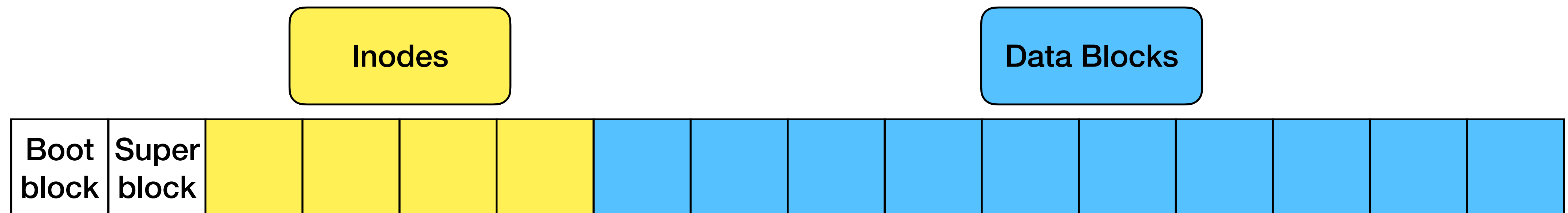
**Code Review in class on
Monday!**

Outline

- Software file system goals
- Mounting a file system
- File system in memory versus on disk
- Implementing free lists for data and inodes

Review: File System as Disk Layout

- We can think of the file system as both the organization of data on disk and the subsystem to maintain the data from disk
- Our subsystem needs to implement tasks such as: (1) reading data from files, (2) writing data to files (e.g., growing the number of data blocks allocated), (3) maintaining knowledge of where free blocks are on disk, and (4) allowing programs to run within a directory



Accessing Files on Disk (Mounting)

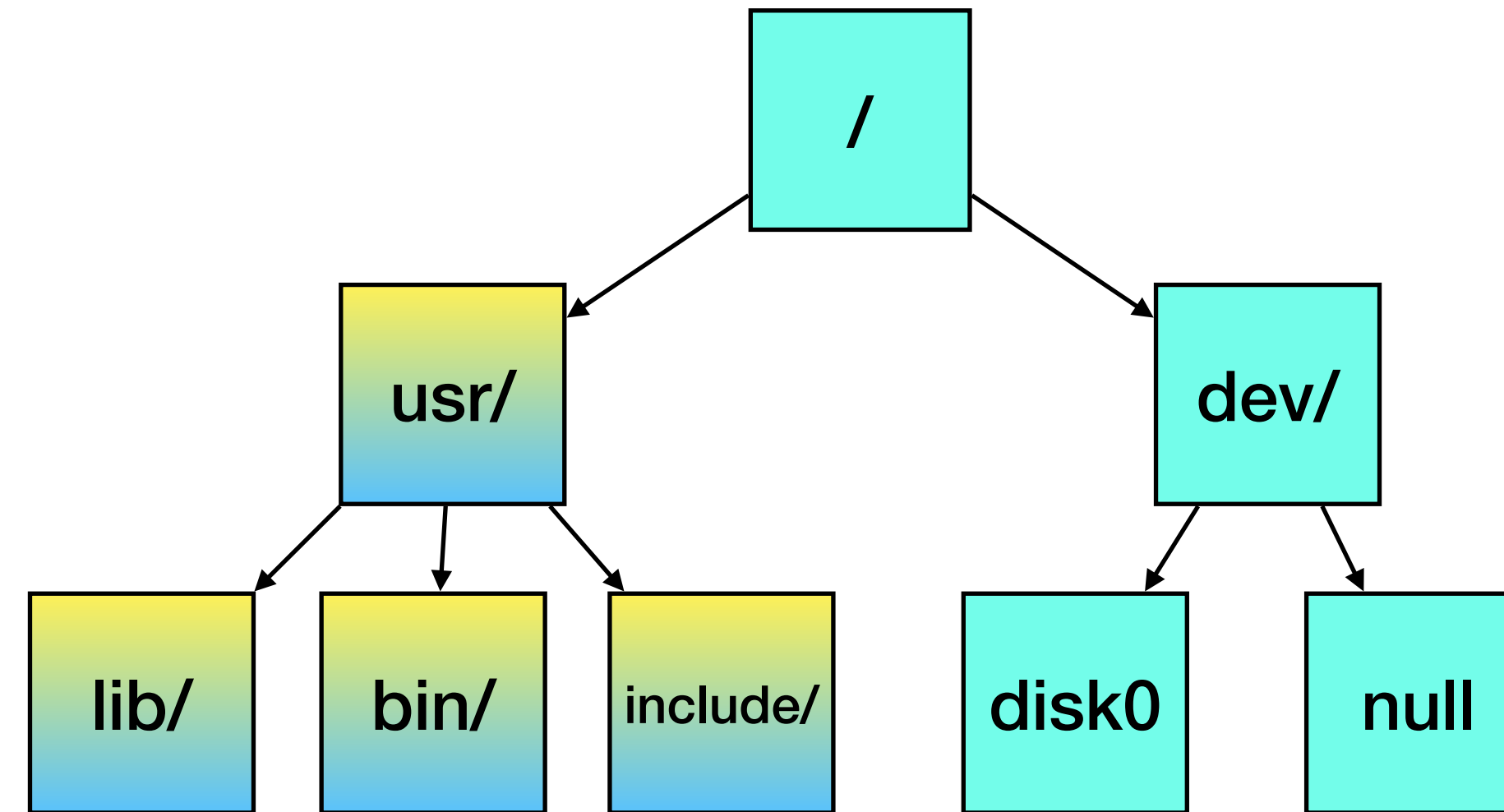
On boot, we have no visibility of the file system, so we create temporary files with no backing in the file system to give ourselves a place to start

Create a “vnode” directory in memory for our file system root

`mount /dev/disk0 /usr`

After “mounting” the disk, the rest of the file system has a representation in the system-wide accessible root

Create a “vnode” directory in memory for each of the devices for which we have a driver



Memory

Disk

Inodes

Data Blocks



Accessing Files on Disk (Mounting)

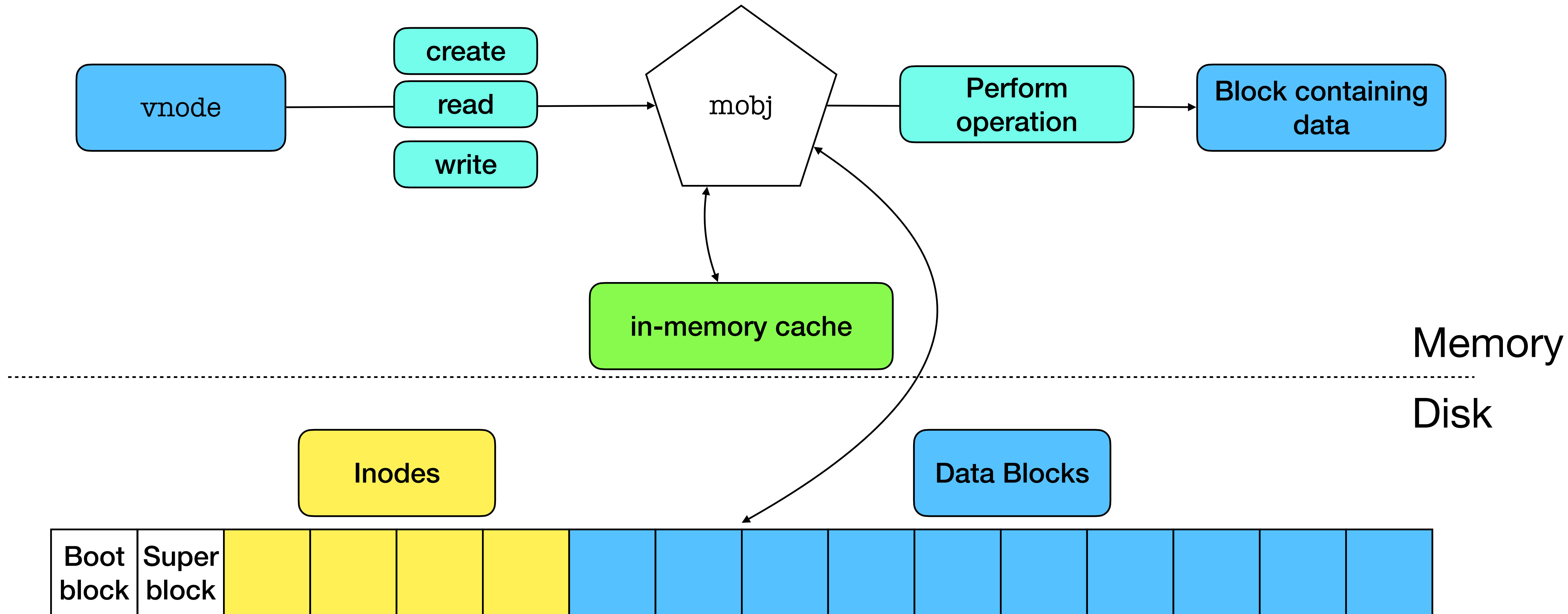
- Our operating system needs to have a way to refer to files and file data as variables to serve user requests ➡ to ease the facilitation of these requests, most operating systems implement a system of *vnodes*
 - A *vnode* is an OS kernel struct that is closely related to a file that lives on disk
- To boot a system, the OS kernel creates a vnode for its file system root that it *mounts* the file system from disk into
- By performing the mounting operation, a link is created between the vnode and the related file in the file system

Chat with your neighbor(s)!

Suppose there is a file in a file system F_1 (on disk) that maintains an image (e.g., a file with a .iso extension) that itself contains another file system F_2 . In this system, F_1 has a subdirectory “/mnt” and we have mounted F_2 into this directory. What happens to F_1 and F_2 if we “cd” into “/mnt” and create a file?

This methodology is similar to the implementation of a virtual machine!

Accessing Files on Disk (Interface)



Chat with your neighbor(s)!

Suppose we are running a version of an operating system that is disk/file system agnostic. How does our mobj know which frames to request from the disk device when reading a file?

Accessing Files on Disk (Interface)

- To implement an `mobj` structure, we need to have precise knowledge of how to read the file system and what the interface to access the file system looks like
 - ➡ If the user wants to read, write, or create a file, then the `mobj` serves as the wrapper for the disk driver to know which blocks from disk it needs to request
 - ➡ The layout of the disk needs to be communicated to the `mobj` structure as the functions it should call to perform these operations
- Once we have this abstraction, accessing the file system merely becomes the task of making the right `mobj` calls

Chat with your neighbor(s)!

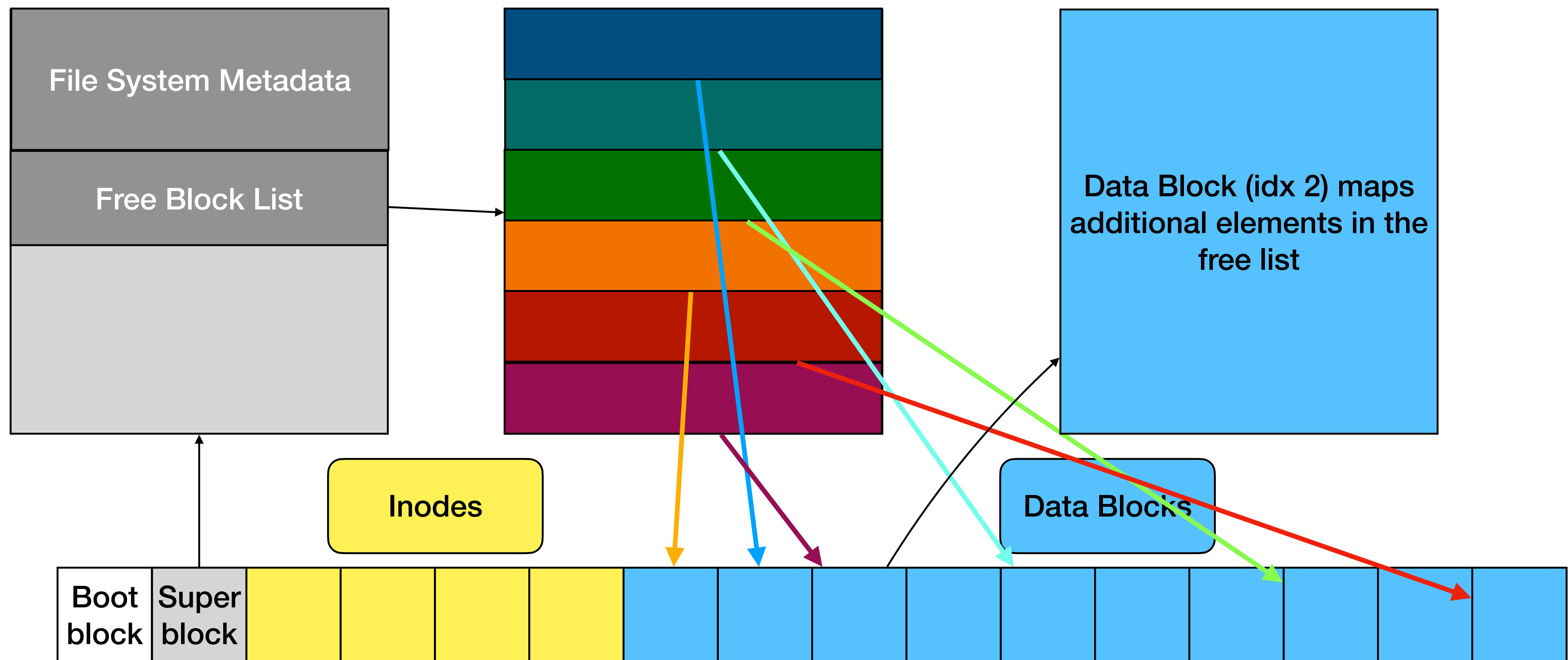
Our `mobj` type maintains the mapping of memory objects (e.g., `vnodes` in OS kernel code) to locations on disk. If we want to grow our `vnode`, how might our operating system implement this system?

We need to maintain a list of free blocks somewhere in file system

We need to know where free inodes are to create new files and we need to know where free data blocks are to append to files

The free lists may be maintained in the OS kernel or in the file system itself!

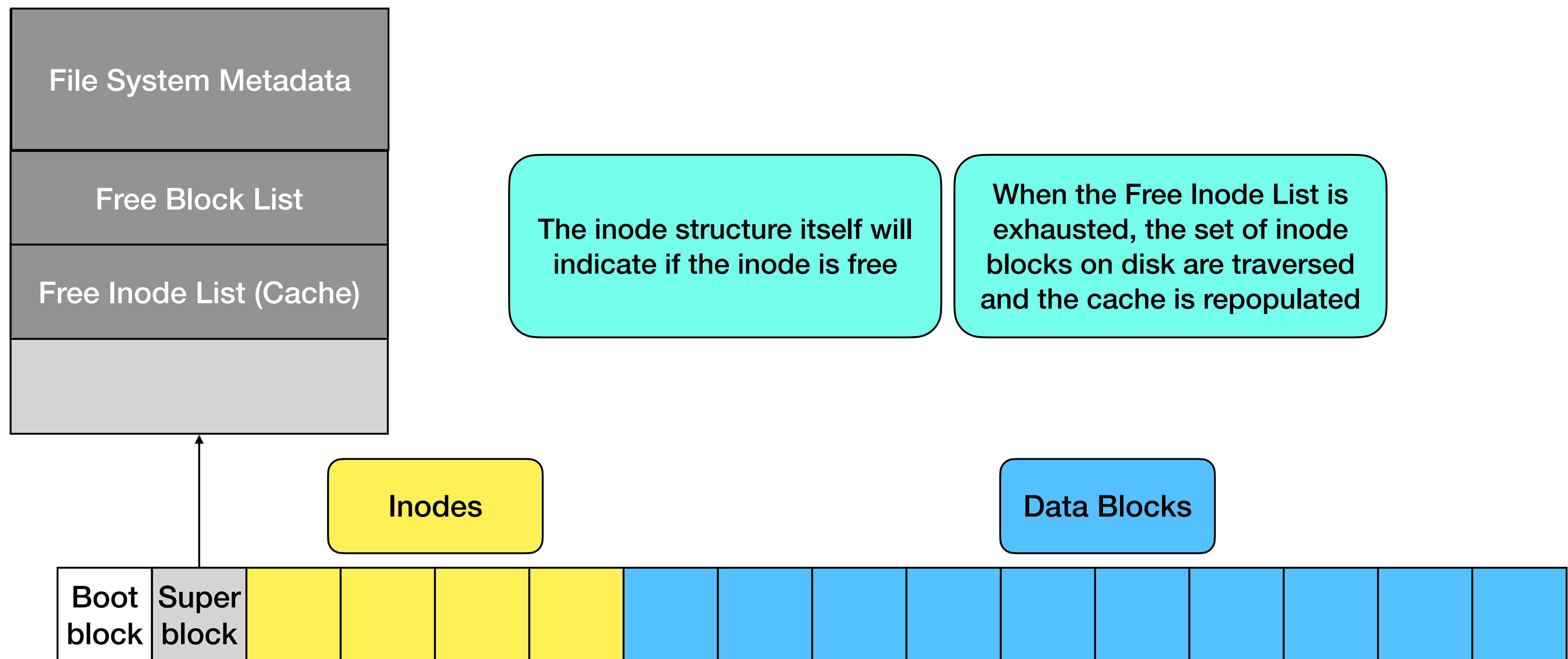
Implementing Free Lists



Implementing Free Lists

- If we implement a system in which, on boot, a list of free elements is maintained in the OS kernel code itself, then we would need to reserve potentially large amounts of kernel memory to maintain the maximum number of free blocks on disk
- Instead, we can leverage the superblock of the file system to indicate which blocks in the disk are not currently in use ➡ when we run out of space in the superblock, the last entry points to a data block that will maintain the rest of the pointers

Implementing Inode Lists



Implementing Free Lists

- If we implement a system in which, on boot, a list of free elements is maintained in the OS kernel code itself, then we would need to reserve potentially large amounts of kernel memory to maintain the maximum number of free blocks on disk
- Instead, we can leverage the superblock of the file system to indicate which blocks in the disk are not currently in use ➡ when we run out of space in the superblock, the last entry points to a data block that will maintain the rest of the pointers
- We also need to maintain a list of which inodes are free so that we can create new files in the file system ➡ we can optimize this by keeping a “free inode cache” in the super block

Our (Modified) Process Structure

Our previous declaration of the process structure has no knowledge of the file system or current directory

We will come back to the vnode type in more depth next week!

```
1 /* Process descriptor */
2 typedef struct proc
3 {
4     pid_t p_pid;           // Process ID
5     char p_name[MAX_STRING_LEN]; // Process name
6
7     list_t p_threads;      // Threads list
8     spinlock_t p_threads_lock; // lock for threads list
9     list_t p_children;     // Children list
10    spinlock_t p_children_lock; // lock for children list
11    struct proc *p_pproc;   // Parent process
12
13    list_link_t p_list_link; // Link of list of all processes
14    list_link_t p_child_link; // Link on parent's list of children
15
16    long p_status;          // Exit status
17    proc_state_t p_state;   // Process state
18 } proc_t;
```



```
1 /* Process descriptor */
2 typedef struct proc
3 {
4     pid_t p_pid;           // Process ID
5     char p_name[MAX_STRING_LEN]; // Process name
6
7     list_t p_threads;      // Threads list
8     spinlock_t p_threads_lock; // lock for threads list
9     list_t p_children;     // Children list
10    spinlock_t p_children_lock; // lock for children list
11    struct proc *p_pproc;   // Parent process
12
13    list_link_t p_list_link; // Link of list of all processes
14    list_link_t p_child_link; // Link on parent's list of children
15
16    vnode_t *p_cwd;         // The current working directory
17
18    long p_status;          // Exit status
19    proc_state_t p_state;   // Process state
20 } proc_t;
```

Summary

- File systems can be *mounted* from disk into kernel managed memory by building a file system tree out of vnodes
- To access files on disk, we can use a mobj structure to abstract the user-level file access methods from the disk lookup methodology
- By implementing the free list in the file system itself, we reduce the spatial overhead of the OS kernel
- Using the same vnode abstraction, we can modify our process structure to track the current working directory of the process