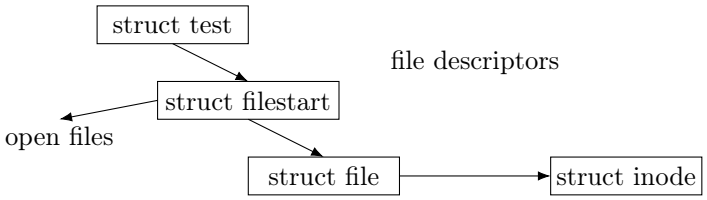
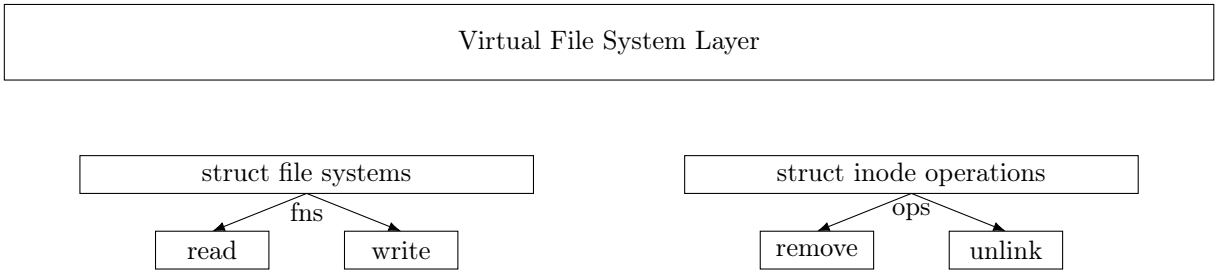
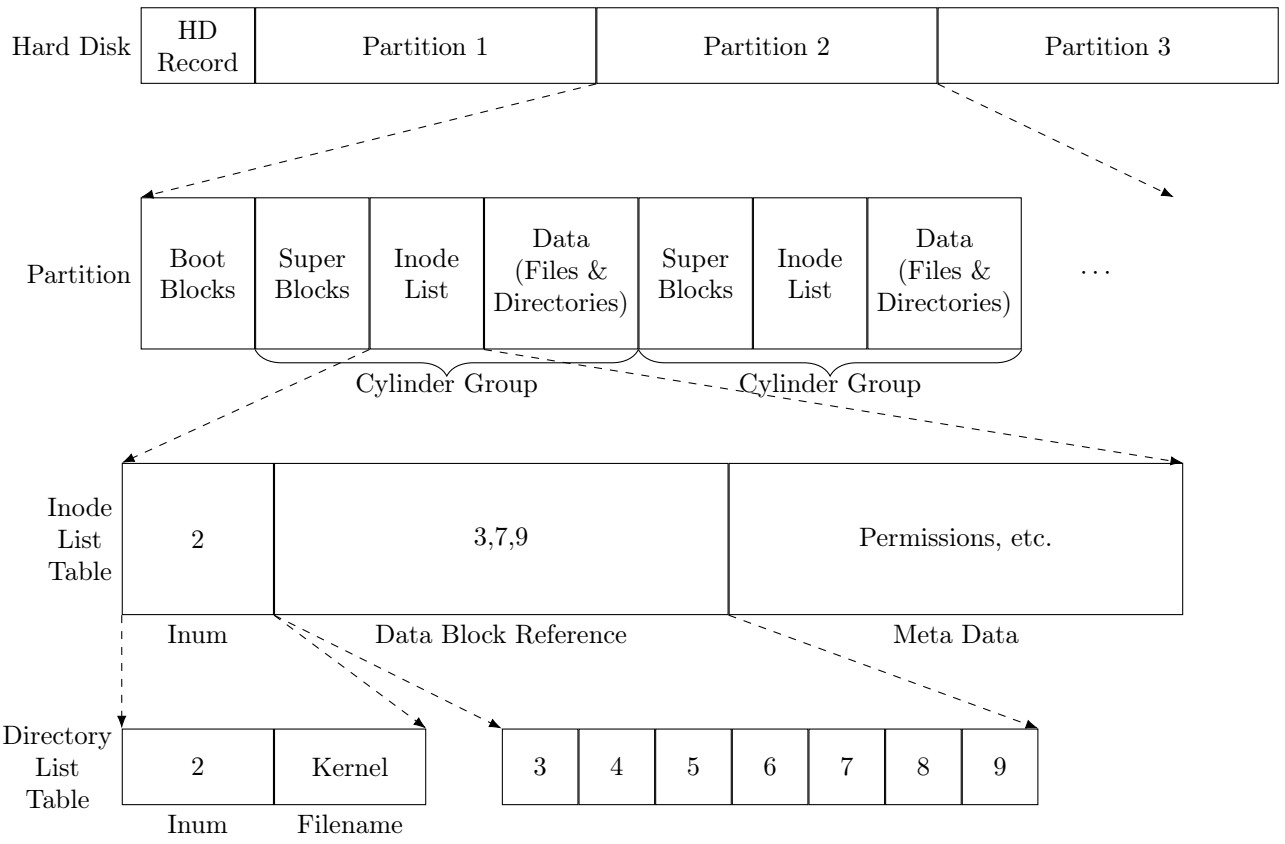
We seek to create a hybrid system that utilizes the best parts of both systems; we want to utilize both the RAM and disk space.

We build our file system based on the following abstraction.





We can then, on the virtual file system layer, implement:

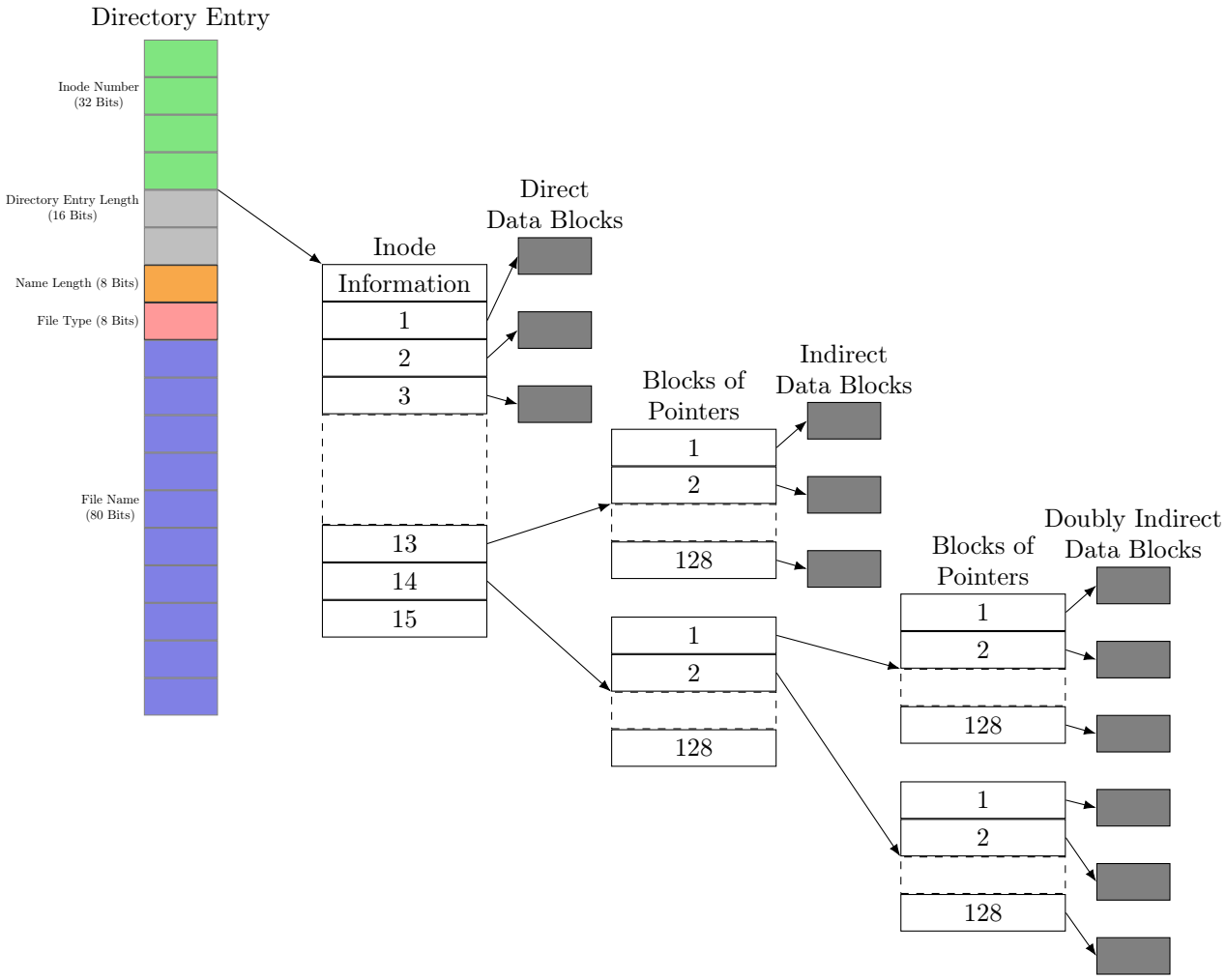


SECTIONS

- Super Block := file system metadata including the root block number

  - Block Bitmap maps index to FREE (0) or ALLOCATED (1) for all blocks
  - (in BFFS) Inode Bitmap maps index to FREE (0) or ALLOCATED (1) for all inodes
- Inode List/Table table of fixed size inode entries that store file metadata and ptr to data
- Data groups of blocks are divided into sectors
  - these are actually tracks, since they can be read quickly
  - these cylinder groups form the partitions of the file system
  - the first directory is left empty since unlink operates on the previous
  - a file referenced by a directory name is a hard link

0.1 Inodes



The data block array stores the block numbers of partitions of data

- 0 means the section is empty
- a file with many 0's is referred to as holey

Inode Metadata (in order):

- owner (32-bit)

- timestamp (last modified time)
- access time (mtime, sysclock)
- inode change (ctime)
- permissions
- file type (directory/regular file/...) (cached, since this one doesn't change)

There are a bunch of new problems though:

1. slow copying (fixed with binary trees)
2. file data is in a linked list, so parsing is slow
   - thankfully, lseek() is O(1)
   - Berkeley Fast File System addresses this with an inode bitmap held in RAM
3. There is an arbitrary limit on the size of our file!
   - a 10 block array can store 10 blocks $\times$ 8192 bytes/block = 82 kB
   - We solve this with an indirect block which points to a block of block pointers. This adds 1023 blocks $\times$ 8192 bytes/block + 82 kB = 8 mB
   - This is still not good enough, so we introduce a doubly indirect block. This adds 1023^2 blocks $\times$ 8192 bytes/block = 8 GB
   - With a triply indirect block, we meet a decent file size (2^{35} blocks = 4TB)
   - If use a 4×10^{12} byte file as a database:
     - we can initialize to zero and avoid any writing
     - space will exhaust; 4×10^{12} may not represent the available space
     - triply indirect block data takes three block accesses

   BUT The above inode approach causes Internal Fragmentation!

How bad is this fragmentation?

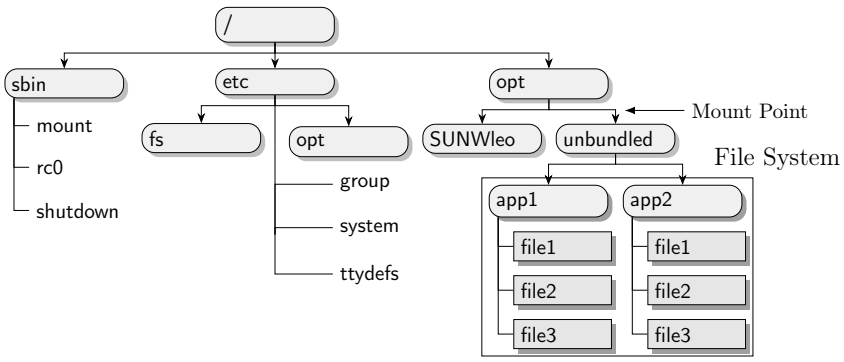
- file size = 0 blocks — 48 bytes wasted for the inode
- file size = 1 blocks — 44 bytes + 8191 bytes = 8235 bytes wasted

If we make a file in the following way:

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC, 0666);
if (lseek(fd, 1000000000, SEEK_SET)) abort();
else write(fd, "Hello", 5);

$ ls -l big
-rw-rw-rw- 5 ...
# the file size is still listed as 5!
# wasted 44 (inode) + 8188 (indirect) + 8198 (double) + 8187 (triple) bytes!
```

0.2 Mounting



How could we keep track of multiple file systems? We pass keeping track to the user by *mounting*

- choose one of the devices to be the root
- mount the other onto the root

The Mount Table maps a device name to its path; a device name therefore acts like a symbolic link

We identify files by inode number, but these are not shared among devices! We need a (device number, inode number) pair! namei() can resolve a name into one of these pairs, recursively, but this introduces a scalability issue!

To solve this, we COULD store pairs in the parent directory, but we want to be able to unmount and mount anywhere. The hosting filesystem should have no reference to the hosted and (more importantly) hard links cant cross devices, since devices can have different types of file system

Similarly, but not as mounting, we can put a file in "chroot jail"

```
$ chroot("subdir")
# now the file can only see files below it
# to get out, we can run:

ln /usr/bin/emacs subdir/bin/emacs
ln /dev/tty subdir/dev/tty
ln /dev/dsk/00 subdir/dev/dsk/00
open /dev/dsk/00 // for rw- permissions
ln {some other file to accessible location}
```

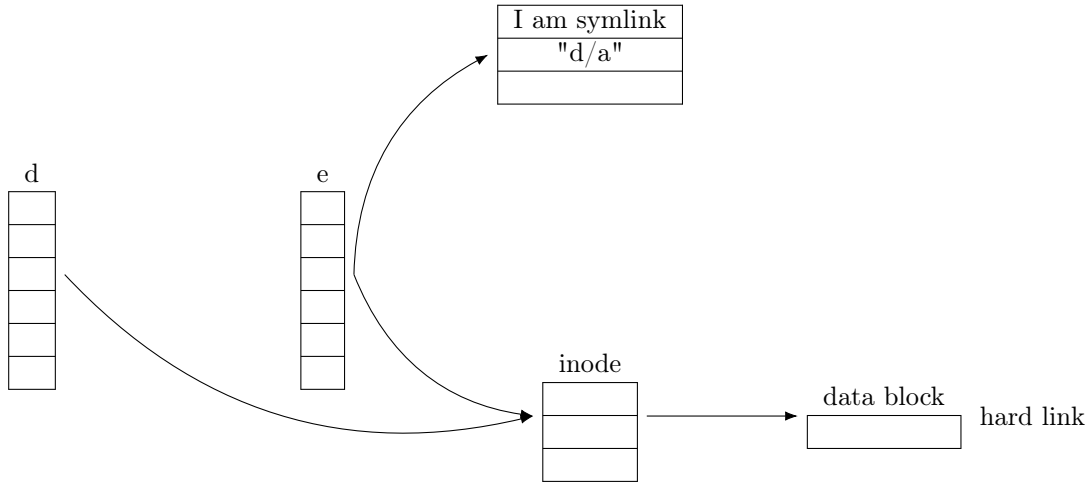
0.3 File Types

UNIX supports the following file types:

File Type	Symbol	Created by	Removed by
Regular File	-	Editors, cp, etc.	rm
Directory	d	mkdir	rmdir, rm -f
Character device file	c	mknod	rm
Block device file	b	mknod	rm
UNIX domain socket	s	socket(2)	rm
Named pipe	p	mknod	rm
Symbolic link	l	lm -s	rm

Symbolic Links (l)

:= a file whose contents contain another file's name. This differs from a hard link, which maps to the inode number.



File names are resolved recursively by namei(). As such, symlinks can introduce problems

```
# ex 1)
$ ln -s . loop
$ ls -l loop
/loop/loop/loop/.....

# ex 2) (assume . is 'd')
$ ln -s .. loop
$ ls -l loop
/loop/d/loop/.....
```

How do we resolve this?

- find, grep, and similar operations do not follow symlinks
- namei() has a depth limit on recursions

```
$ ln -s linkloop linkloop
$ cat linkloop
# sets errno and returns -1!
```

Device Drivers (b||c)

both character and block devices are windows into a drive

- character — stream-ish
- block — storage-ish (read & write are limited to a fixed size)

For example: the serial port driver sends and receives bytes by wire.

```
$ mknod /dev/ser1 c 59 23
# mknod —— make node
# /dev/ser1 —— path to device
# c —— character
# 59 23 —— device ID
```

These form the basis of file systems! They are antiquated, but common in IOT devices, which dont use an OS.

These drivers suggest users can access data outside of establishes guidelines. In fact, root can access memory and read and write by path!

Knowing this, how can we protect our data?

```
# we want to protect our proposal
$ rm prop.txt
# the file is gone right? NOPE: if there were links, it doesn't get deleted
$ (rm prop.txt; ckst) < prop.txt
# (ckst is self defined to check link count)
# what if someone was already reading it?
$ chmod 700 .
# they could still get into the directory by stealing our disk!
$ shred prop.txt
# overwrite our data 2-3 times with random data
# BUT the OS may have copied the data elsewhere
$ shred /dev/ds2b
# we shred our entire file system!
# but what if another device copied it?
```

MELT THE DRIVE (seriously... even physical shredding leaves readable magnetic residue)

Since apps generally use block writing/perform writes on the block level, the block driver sees a large set of requests; how does it schedule these? We have a few algorithm choices:

1. SHORTEST SEEK TIME FIRST
   (a) perform the shortest seek first
   (b) maximal throughput
   (c) very unfair — allows starvation
2. FIRST COME, FIRST SERVED
   (a) perform the oldest seek first
   (b) low throughput
   (c) very fair
3. ELEVATOR
   (a) a hybrid approach
   (b) perform the closest request in a set direction
   (c) good throughput
   (d) prioritizes locations near the center

   we can use a non-snaking path to avoid this

BUT these algorithms are designed for disks! What can we do for blocks? (which don't have the same seek time?

(4) Anticipatory Scheduling

- is done at the OS or controller level
  - naively:
    $$\{P1, 1000, P2\} \implies P1, 1000, P2$$
  - anticipatory:
    $$\{P1, 1000\} \to \{1000\} \to \text{ dally } \to \{P2, 1000\} \to \{1000\} \implies P1, P2, 1000$$

A write cannot be switched with a read/write to the same location.

This introduces a new set of risks.