P5 - Filesystem

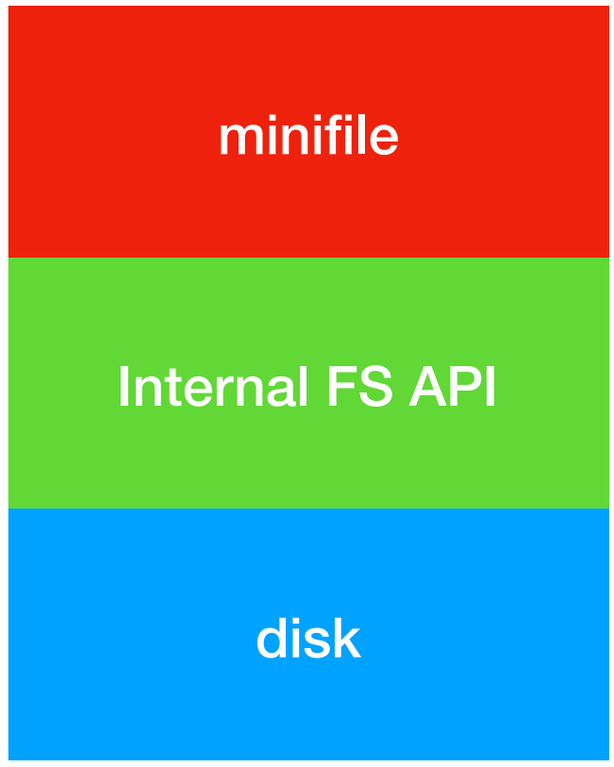
Drew Zagieboylo
4/27/18

Filesystems

- Provide permanent storage
- Goals:
  - Common use cases are fast
  - Low storage overhead
  - Simple structure

Filesystems

- Several Layers of Abstraction
  - File ‘Handles’ (Descriptors) - given
  - FileSystem Data structures - you define
  - Disk - given



The Disk

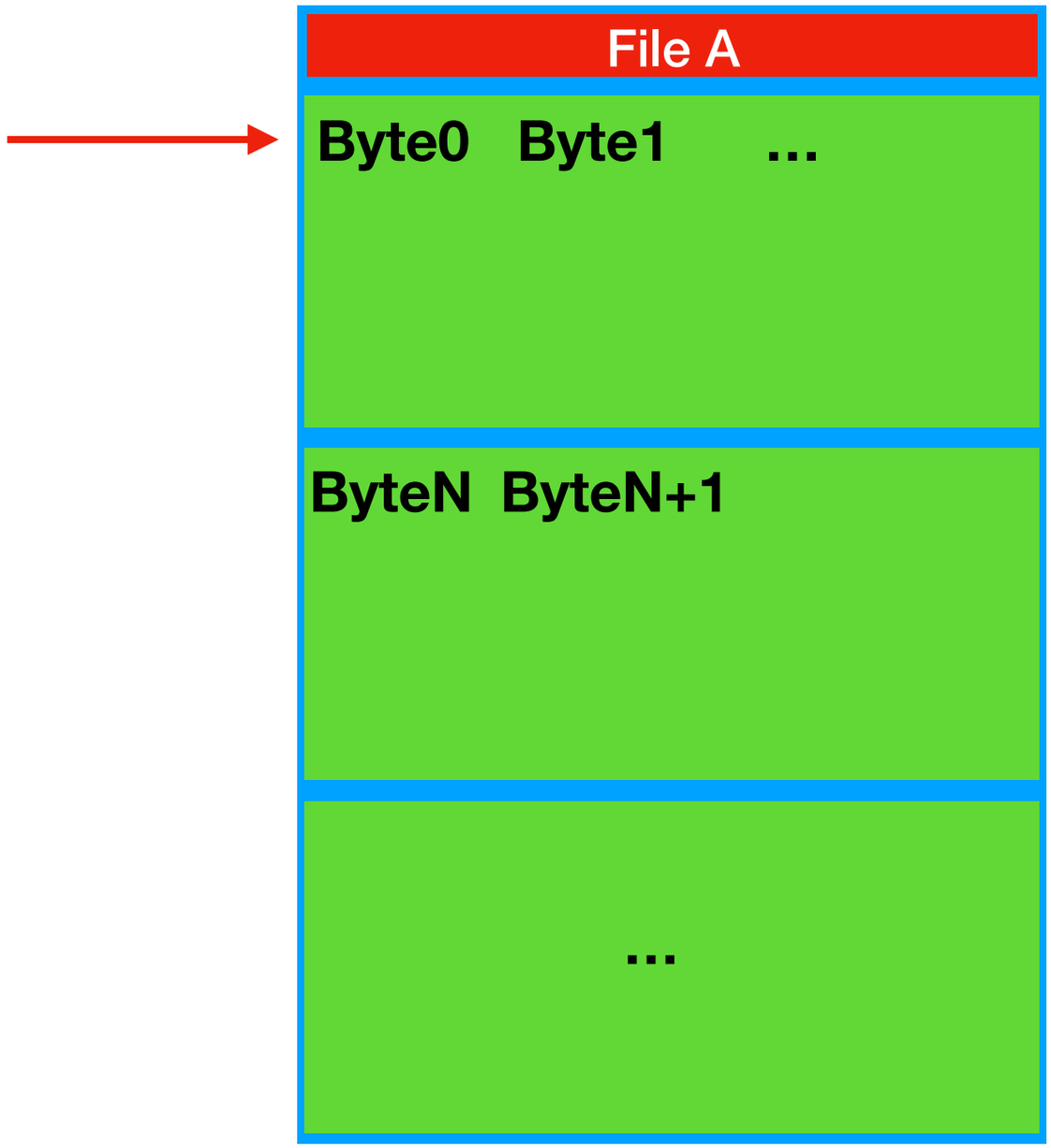
- Independently controlled
- Asynchronous access
- Re-ordered requests
- `disk_read_block(disk_t* disk, int blocknum, char* buffer)`
- `disk_write_block(disk_t* disk, int blocknum, char* buffer)`

Minifile API

- A subset of the Unix File API
  - can lookup details with the ‘man’ command
- Create/read/write/delete Files
  - different “modes” allow slightly different behavior
- Create/Delete Directories
- “Deletion” functions are optional
  (but will net you some E.C.)

Minifile API -Files

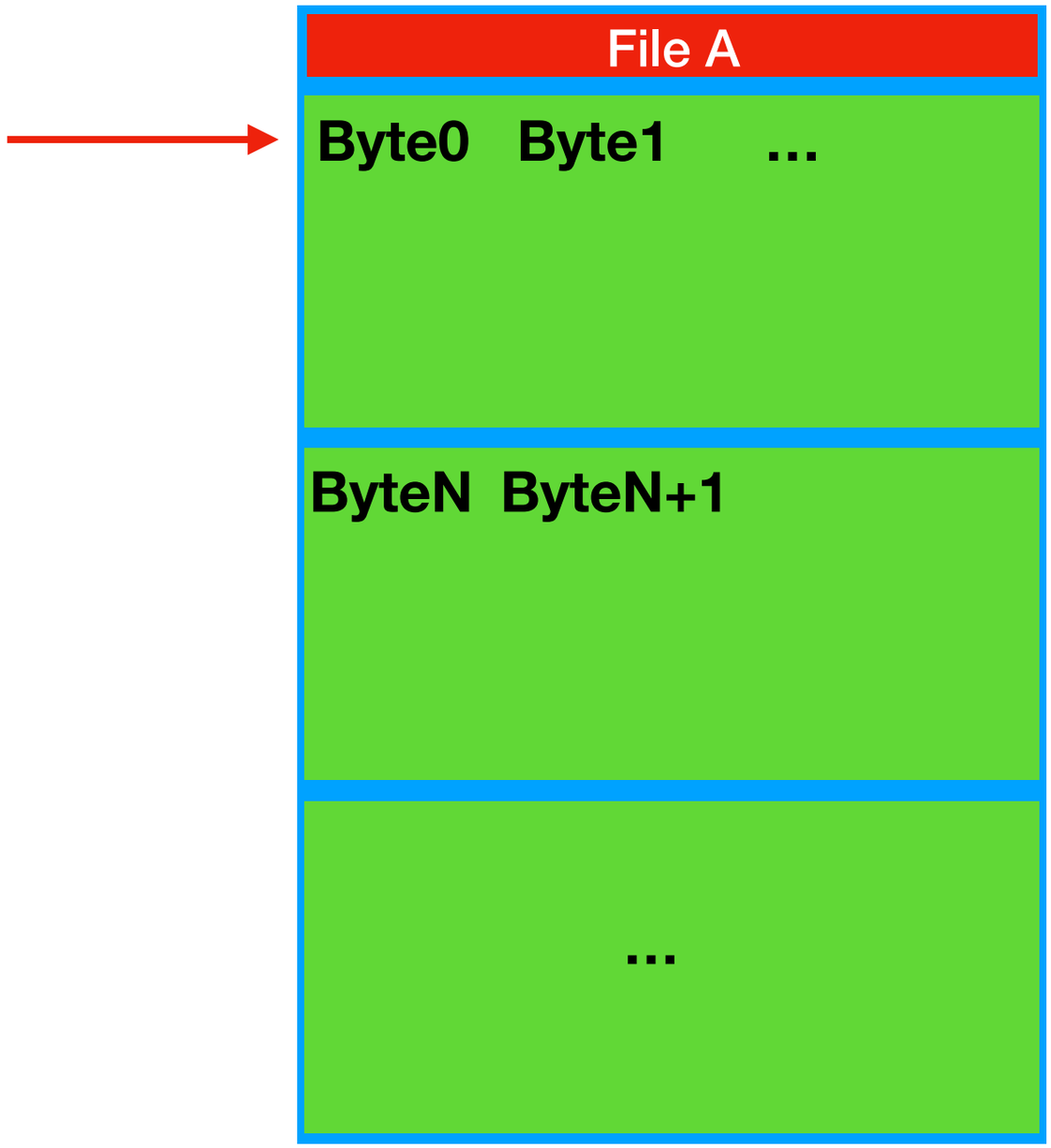
- Opening a File
  - Creates a stateful File Handle (`minifile_t`)
  - Maintains a “cursor” which points to a byte in the file
  - Independent *Read* and *Write* “cursors”



Minifile API -Files

- Call ‘read’ on file to read N bytes

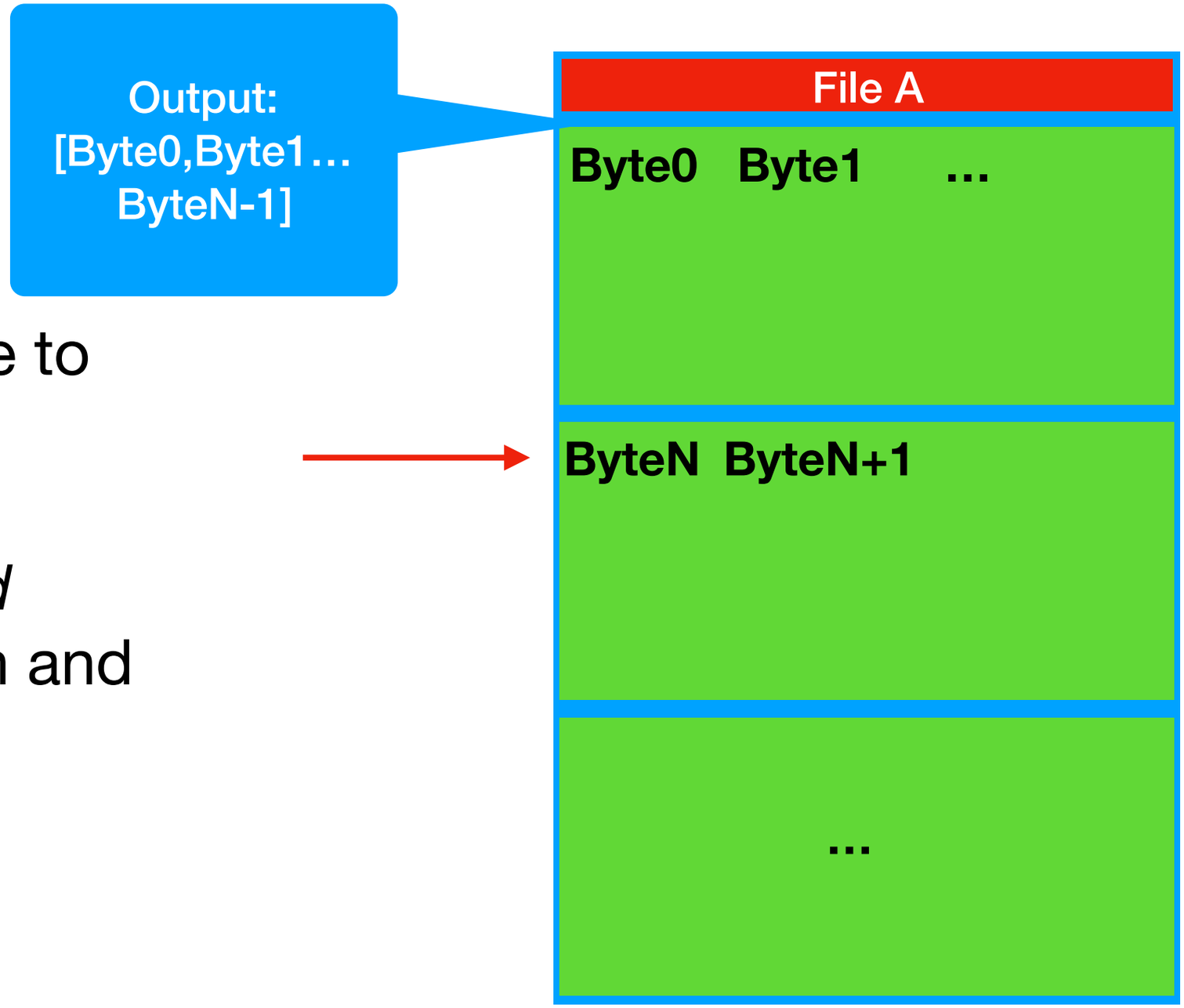
- Reads from *read* cursor’s position and advances



Minifile API -Files

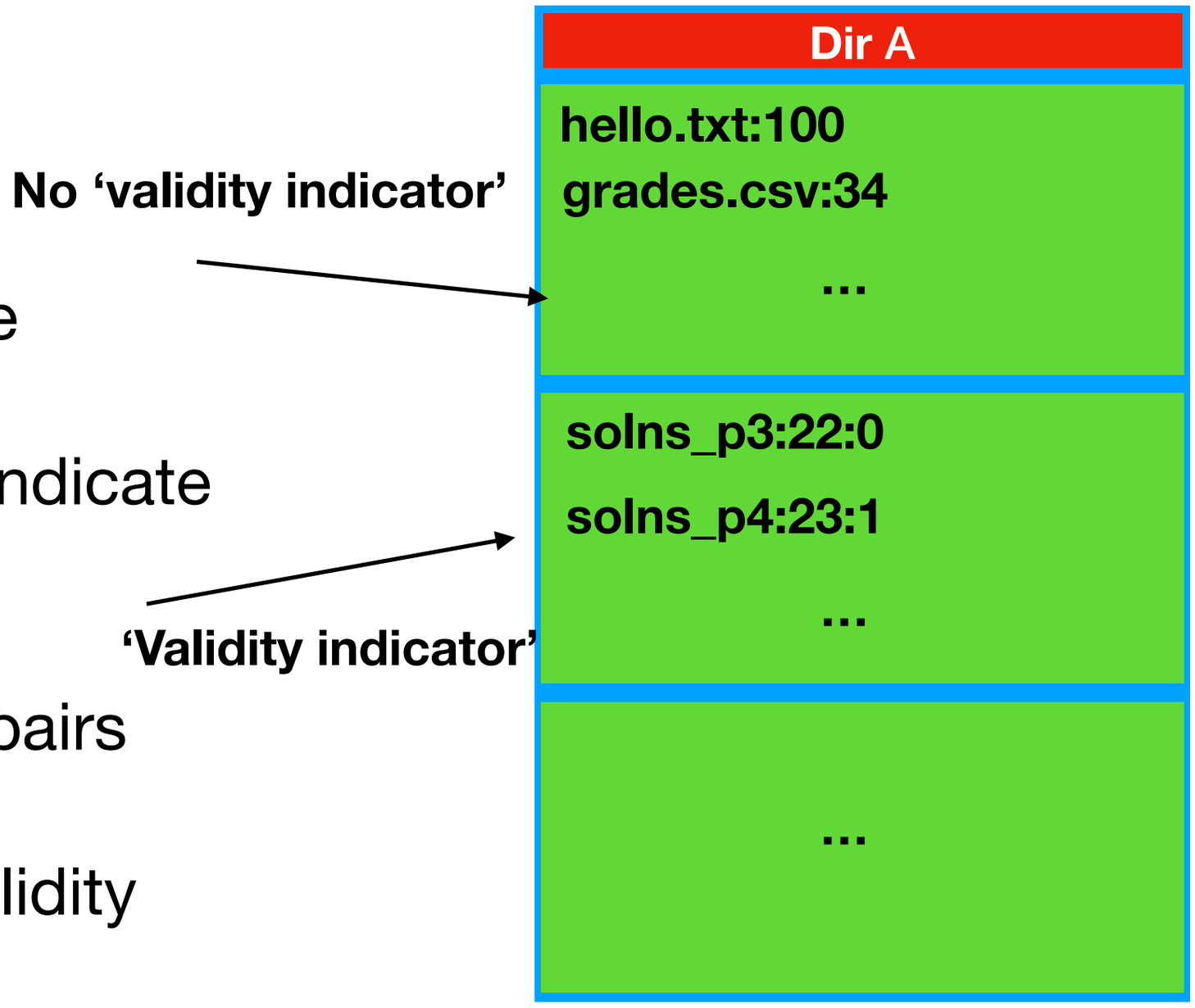
- Call ‘read’ on file to read N bytes

- Reads from *read* cursor’s position and advances



Minifile API -Dirs

- Directories
- Special kind of file
  - Bit in inode to indicate
- Data = A list of *name*:*inodeNum* pairs
- Maybe helpful: validity



Minifile API

- Creating directories
  - just like creating files
- Removing directories
  - just like removing files (except it has to be empty!)
- `minifile_ls`
  - `['hello.txt', 'solns_p3', 'grades.csv', 'solns_p4', '.', '..']`
  - `'.' and '..' are optional`

Minifile API - Concurrency

- Real file systems have extensive support
- Your obligation: serialize all operations on a `minifile_t`
- Concurrent operations on the same file
  via different file handles
  have **undefined behavior**
  - E.g.
    - F is a file handle for ‘/drew/foo.txt’
      T1 calls write(F)
      T2 calls read(F)
    - Either:
      - T1 writes, THEN T2 reads
      - T2 reads, THEN T1 writes

Internal FS API

- This is a *recommended strategy* for structuring code
- You define it!
- Needs to do things like:
    - Allocate/Free Inode
    - Allocate/Free Datablock
    - Read/Write block *n* from File
- Can then use to implement the **Minifile** API

MKFS

- “Make File System”
- Required file for submission (mkfs.c)
- Generates a file system containing only 1 directory (the ‘root’ directory)
- Uses the Linux file “MINIFILESYSTEM”

```
int main(int argc, char** argv) {
    use_existing_disk = 0;
    disk_name = “MINIFILESYSTEM”;
    minithread_system_initialize(…)
    …
```

File System Review

- For this project
  - Superblock (examples)
    - Root Inode Num
    - Num of First Free Inode
    - Num of First Free Datablock
- Inodes:
  - 11 direct blocks
  - 1 indirect block
    - Multiple Inodes fit in 1 block -> can just use 1
- Free List(s)
  - Keep Track of unused blocks

File System Review

- http://www.cs.cornell.edu/courses/cs4410/2018sp/schedule/slides/11-filesystems.pdf