Lab 4 Intro

Filesystem

Quick notes

- Lab3 due tomorrow
- Problem set 3 due tomorrow
- Lab 4 is out!
  - Design doc due next Thursday (02/29)
  - Lab due during finals week (03/13)

Think Back To Lab 1...

- Files were read-only
  - open denies O_WRONLY and O_RDWR flag for files

But For Lab4

1) Make the filesystem writable
    a) remove file write restriction (no need to check T_DEV, will fail lab1test and that's fine)
    b) support file overwrite (write to existing blocks, implement writei)
    c) change the inode layout to support file extension
    d) support file creation and deletion (allocate/free inode, adjust data in root directory)
2) Support concurrent filesys operations
3) Make the filesystem crash-safe
    a) implement some form of logging

Prologue: Tour of the xk Storage Layer

Once Upon a Time

Major Layers: Filesys

- File System, Files, and Directories (fs.h/fs.c, file.h/file.c, extent.h)
- Persistent Data:
  - on disk inode: metadata for file/directory (struct dinode)
  - extents: how inode tracks its data location
  - inode file: special file where the file data is a list of disk inodes
  - bitmap: used to keep track of free and used blocks on disk
  - superblock: metadata for the whole file system
    - tracks location of bitmap sectors and inode file sectors
- In memory Data:
  - struct inode (in memory/cached copy of the inode)
  - open inode array (cache for in memory inodes)

Major Layers: Block Cache

- Block Cache/Buffer Cache (bio.c)
    - brings block/sector into memory and manages them (evict, writeback)
    - struct buf
        - metadata for managing buffer, buf->data = sector data
    - APIs
        - bread: brings the sector into memory, locks (exclusive access) the cached block
        - bwrite: marks the cached block dirty
        - brelse: releases the lock on the block
- IDE Connector (ide.c)
    - block interface, no need to modify it, can read if curious

Userland

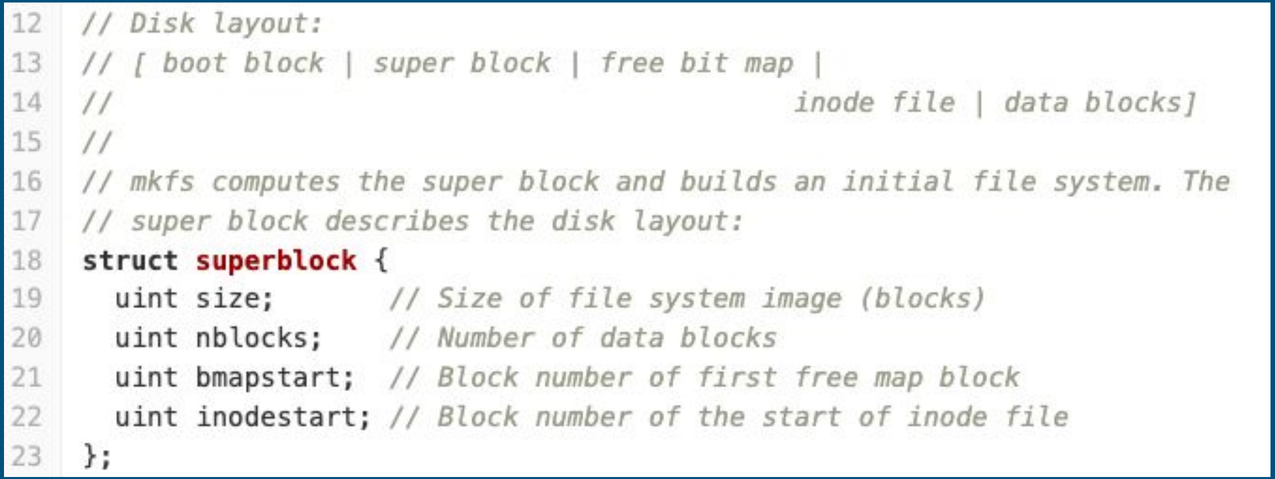
KERNEL LAND

System Calls	File API	Inode API	Block API	IDE API

write()
open()
	filewrite()
	fileappend()
	filecreate()
		writei()
		readi()
			bread()
			bwrite()
			brelse()
				iderw()

Persistent Data

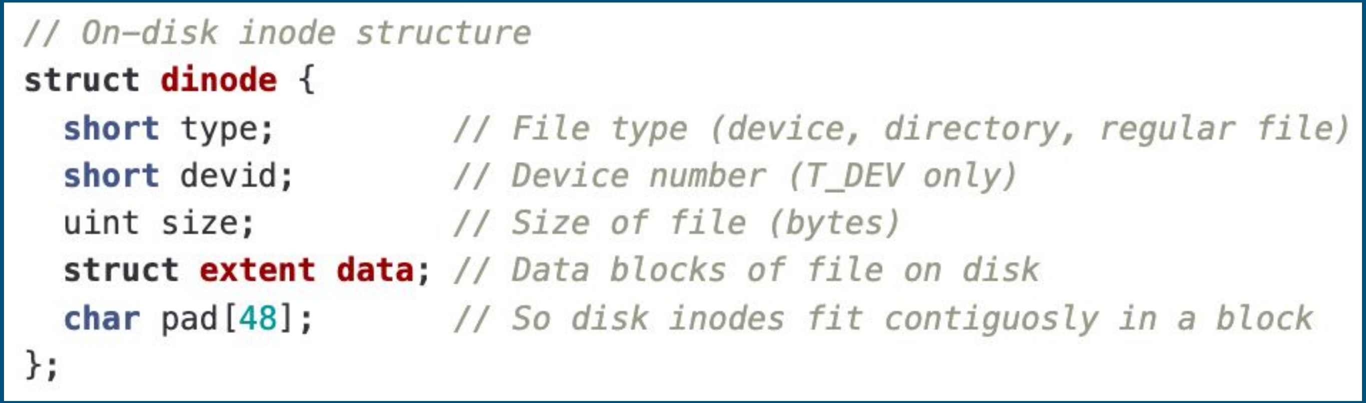
FS: Superblock



```
// Disk layout:
// [ boot block | super block | free bit map |
//                                                   inode file | data blocks]
//
// mkfs computes the super block and builds an initial file system. The
// super block describes the disk layout:
struct superblock {
  uint size;       // Size of file system image (blocks)
  uint nblocks;    // Number of data blocks
  uint bmapstart;  // Block number of first free map block
  uint inodestart; // Block number of the start of inode file
};
```

track metadata for the entire file system, persistent structure
- tracks location for bitmap
- tracks location for metadata table (inode array / inodefile)

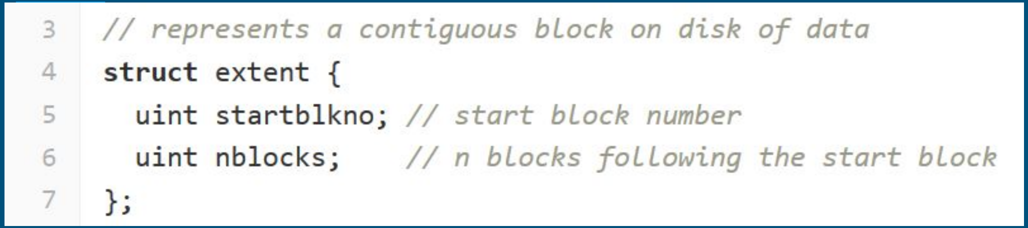
FS: dinode



```
// On-disk inode structure
struct dinode {
  short type;          // File type (device, directory, regular file)
  short devid;         // Device number (T_DEV only)
  uint size;           // Size of file (bytes)
  struct extent data;  // Data blocks of file on disk
  char pad[48];        // So disk inodes fit contiguosly in a block
};
```

- disk inode: metadata for files/directories
- lives on disk, cannot have any pointer fields or locks (why?)
- defines data layout: currently 1 extent

FS: extent



```c
// represents a contiguous block on disk of data
struct extent {
  uint startblkno; // start block number
  uint nblocks;    // n blocks following the start block
};
```

- A way to track how/where data is stored
- One extent tracks a contiguous chunk of sectors
    - startblkno: sector number of the beginning sector
    - nblocks: number of sectors
    - tracks sectors [startblkno, startblkno + nblocks)
- Multiple extents can track multiple chunks of sectors

FS: extent

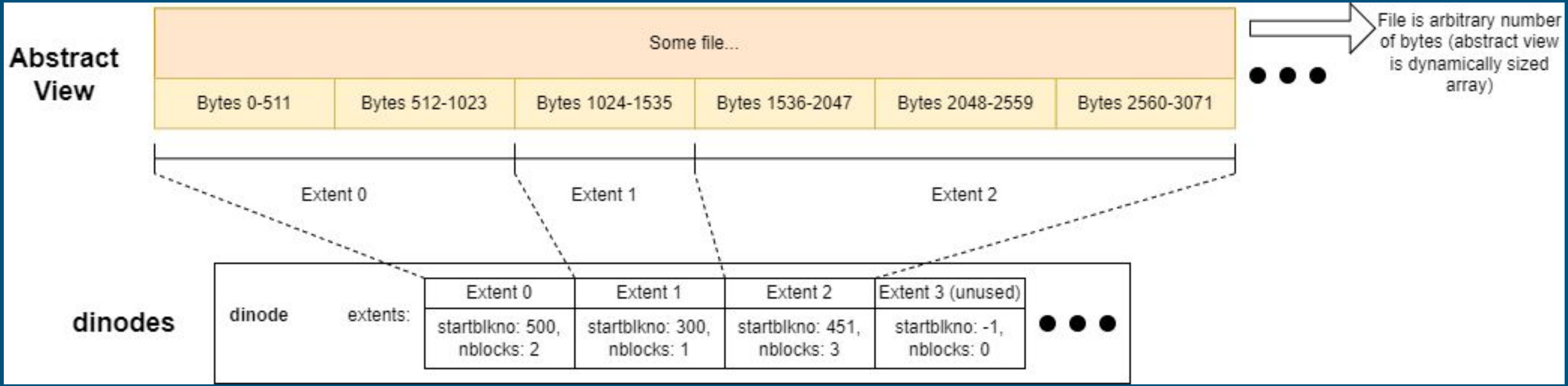
```
struct extent {
        startblkno: 100;
        nblock: 2;
}
```

- This means that the file's data takes up two sectors: 100 and 101
  - byte 0-511 of the file can be found in sector 100
  - byte 512-1023 of the file can be found in sector 101

FS: More on dinode

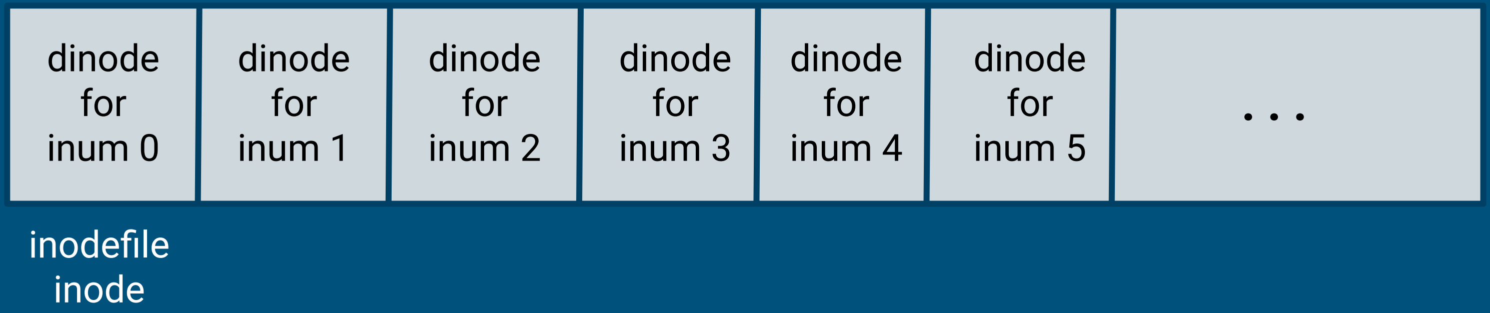
- To support file growth, need to modify the disk inode to support multiple extents
  - can cap at 30 extents (would allow 30 total file extensions)
  - need to keep sizeof(struct dinode) a power of 2 (currently 64)
    - multiple dinodes must fit fully within a single sector (512 bytes)

disk inode with multiple extents



FS: inodefile

- Special file for storing on metadata for file/directory
    - data is an array of on disk inodes (dinode)
    - data block starts at sb.inodestart (a block number)
- Where is the metadata for inodefile?
    - it's the first dinode in inodefile's data, inum = 0
    - inodefile is special in that its metadata is stored within its data
    - how do we find metadata (first data block)? superblock tells us where the data starts!



FS: bitmap (kernel/fs.c)

Bitmap sectors live on disk and track the usage information of all disk sectors

- each bit in the bitmap tracks the usage info a sector on disk
  - 0th bit set to 1 means sector 0 is used
  - 100th bit set to 0 means sector 100 is free
  - bitmap itself lives in sectors
- bitmap sectors start at sb.bmapstart
- Existing xk API helps manage the bitmap for you

FS: bitmap (kernel/fs.c)

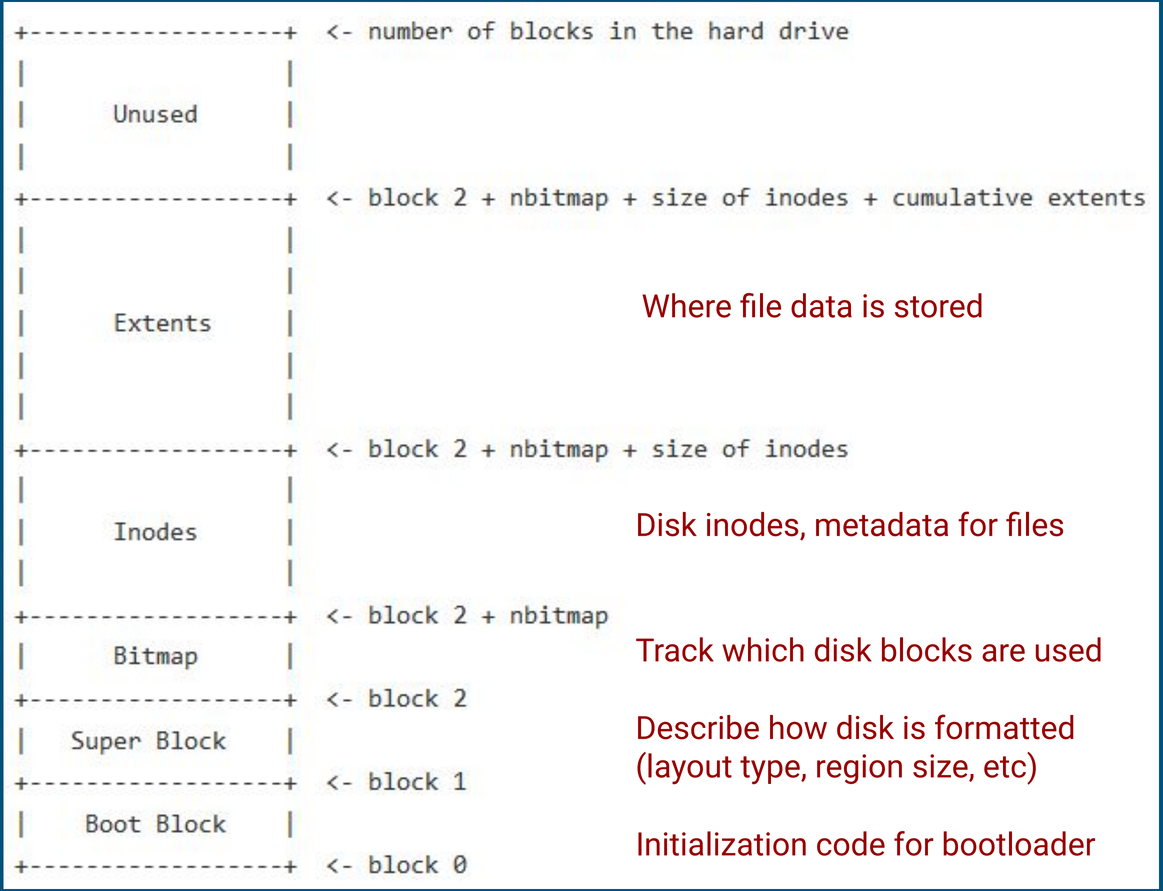
- balloc()
    - Allocates consecutive blocks for a given device [should use devid from inodefile's inode]
    - Panics when not enough consecutive blocks available
    - Does not guarantee that block contents have been zeroed
- bfree()
    - Frees consecutive blocks for a device
    - Will not free contiguous blocks belonging to different bitmap sectors

WARNING: these functions do not change the bitmap on disk. You will need to update them to write changed bitmap sectors to disk

Initial Disk Layout

How things are stored on disk

mkfs.c (a POSIX program, not an xk program!) writes the initial disk image following this layout



```
+------------------+  <- number of blocks in the hard drive
|                  |
|      Unused      |
|                  |
+------------------+  <- block 2 + nbitmap + size of inodes + cumulative extents
|                  |
|                  |
|      Extents     |
|                  |
|                  |
+------------------+  <- block 2 + nbitmap + size of inodes
|                  |
|      Inodes      |
|                  |
+------------------+  <- block 2 + nbitmap
|      Bitmap      |
+------------------+  <- block 2
|    Super Block   |
+------------------+  <- block 1
|    Boot Block    |
+------------------+  <- block 0
```

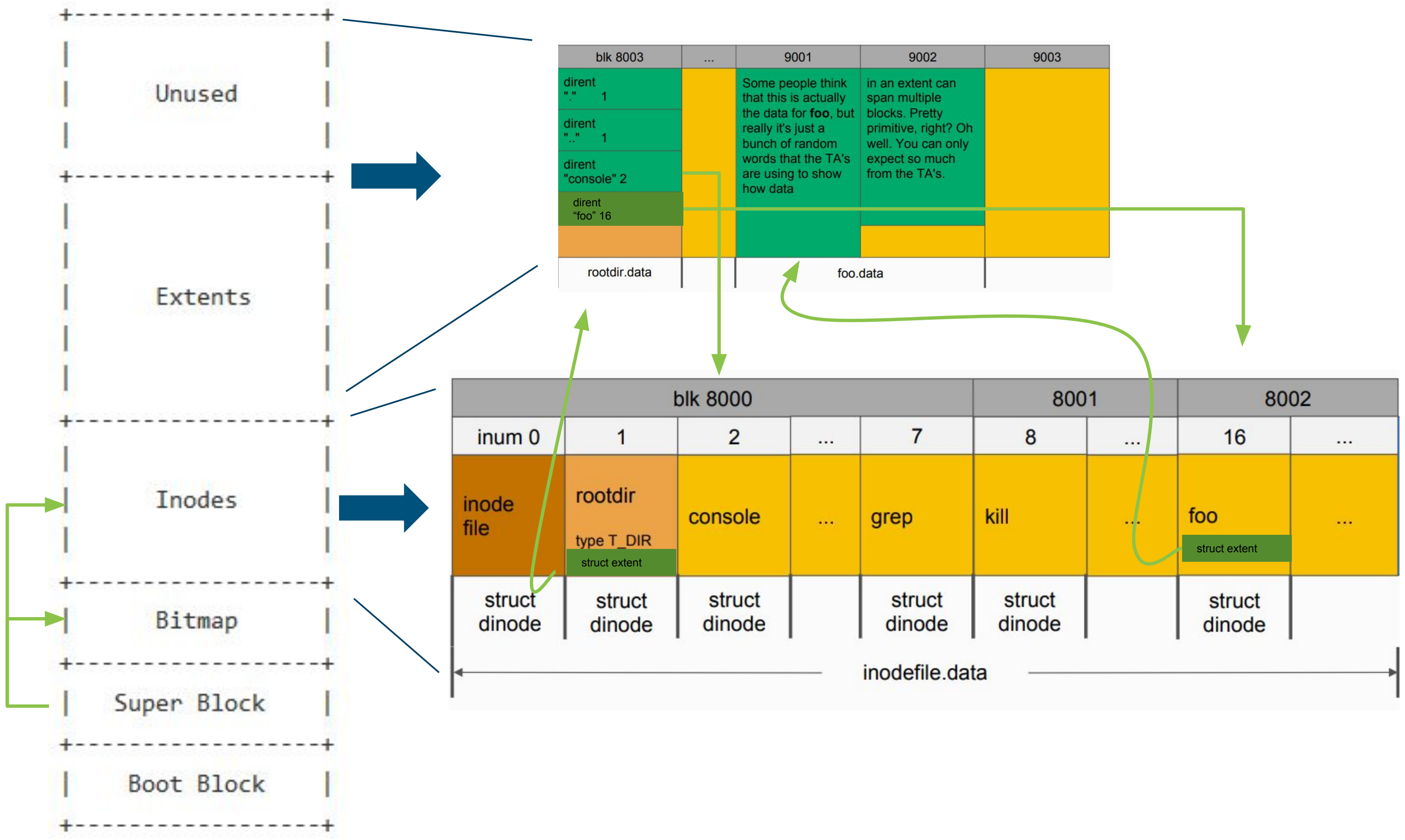
Where file data is stored

Disk inodes, metadata for files

Track which disk blocks are used

Describe how disk is formatted (layout type, region size, etc)

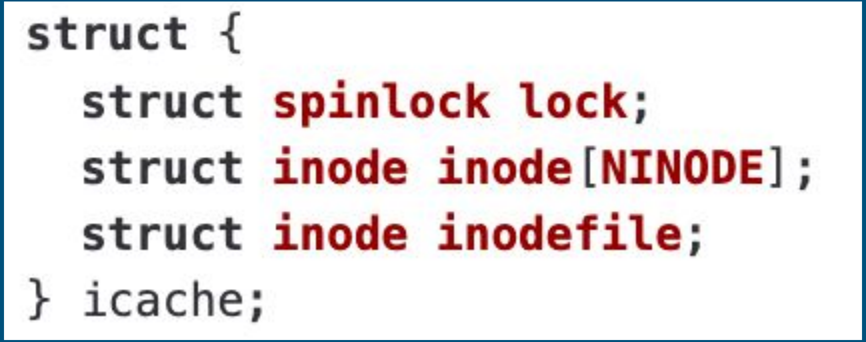
Initialization code for bootloader



Some people think that this is actually the data for **foo**, but really it's just a bunch of random words that the TA's are using to show how data in an extent can span multiple blocks. Pretty primitive, right? Oh well. You can only expect so much from the TA's.

In Memory Data

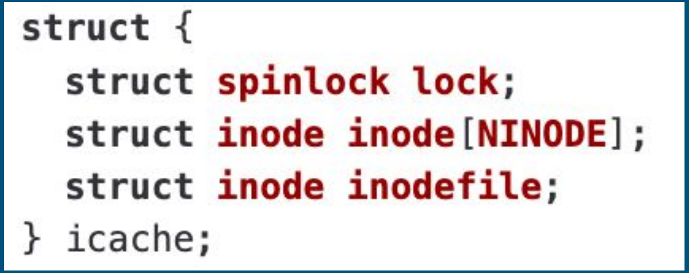
FS: icache



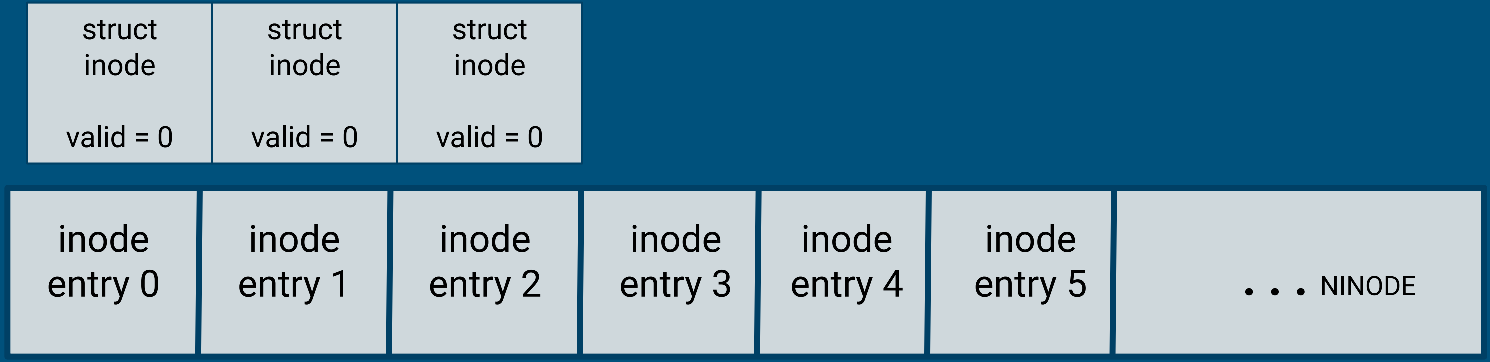
```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
  struct inode inodefile;
} icache;
```

For ease & speed of access, we keep a cache of on disk structures in memory. Includes a lock protecting accesses to the cache, an inode cache for on disk inodes, and the cached inode for the inodefile itself.

icache.inode array

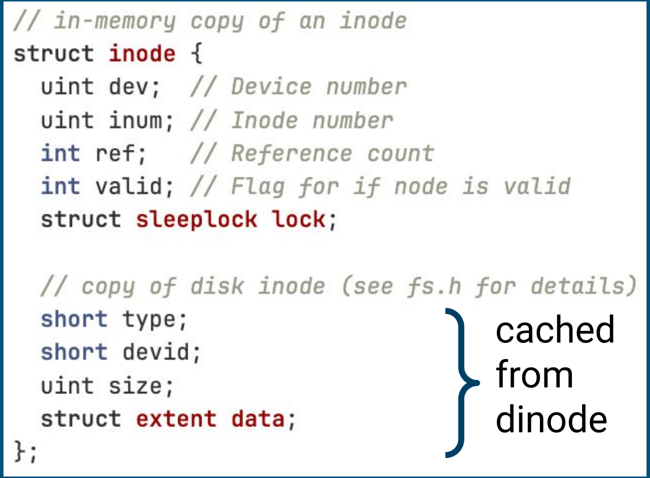


```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
  struct inode inodefile;
} icache;
```



initially, no dinodes are cached, all entries' valid field == 0

FS: inode



```
// in-memory copy of an inode
struct inode {
  uint dev;   // Device number
  uint inum;  // Inode number
  int ref;    // Reference count
  int valid;  // Flag for if node is valid
  struct sleeplock lock;

  // copy of disk inode (see fs.h for details)
  short type;
  short devid;
  uint size;
  struct extent data;
};
```

- In memory inode
  - a cache copy of the disk inode & in memory bookkeeping, where does it live?
  - lock, refcount, valid (tracks whether the disk inode fields are populated with valid data)
- changes to inode is not automatically reflected on the actual dinode and vice versa
  - locki will synchronize the inode with dinode when inode->valid == 0

Part A: Writable FileSys

should first modify file_open to allow opening files in write mode
(and patch lab1 tests if you want to)

Types of writes

- overwrite
  - overwrite data in existing data blocks
  - file size (metadata) is not changed after an overwrite
  - only file content is changed as a result of the write
  - file of length 100 bytes, write 20 bytes at offset 0 is just an overwrite
- append
  - writing data past end of file, metadata changes!
    - given a file of length 100 bytes, writing 20 bytes at offset 100
      - writing 20 bytes at offset 90 will also cause an append!
  - may cause additional blocks to be allocated (also additional extent)
    - if appended data can fit within the current block no need to allocate new data blocks
      - how many blocks are allocated for a file with length 100?
      - do you need to allocate a new block for 20 more bytes?
    - otherwise, need to allocate data blocks

Append

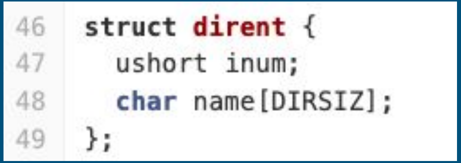
- If additional blocks are needed for appended data
  - can use bitmap function to find free blocks
  - need to modify data layout to allow tracking of multiple extents
- Consequences of modifying disk inode
  - recall that the initial filesys image is written by the POSIX program mkfs.c
  - changing disk inode's layout means that you need to modify mkfs.c so it can write the initial disk image with your new disk inode
  - mkfs.c has visibility into struct dinode
- append and overwrite will both call writei to perform the write!

Writei

- responsible for performing updates to data and metadata (inode)
  - changes in data blocks must be written out to disk
  - changes in dinode (metadata) also need to be written out to disk
- helpful function for interacting with block cache
  - bget, bwrite, brelse
  - can only read/write in units of blocks/sectors
  - hint: readi is a helpful example for how to interact with block cache

FS: directories

- Directories are like ordinary files (they have an inode associated with)
- Data is an array of directory entries (dirents)
- Dirent has two fields, name and inum



```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

Create

Be able to create a new file when O_CREATE is passed to file_open

- Allocate a new dinode in inodefile
    - should first check if there are any free dinode in the inodefile
        - how to tell if a dinode is free? reuse old fields, set type or size = -1 or add a new field
    - if not, create a new dinode by appending to inodefile
- Update data of the root directory to track a new dirent
    - new direent: new file name, dinode number
    - all files will be created under root dir, no nested directories for this lab
    - if root dir has any invalid dirent, can reuse that entry for your new dirent

Delete

- unlink(char* path) system call
    - If path exists and no open references to the file, delete from the file system*
        - Effectively undoing steps from file creation
    - Otherwise, error
- Supporting file deletion
    - free the dinode associated with file => inodefile can be fragmented
    - need to ensure file creation can fill holes in the inodefile
- Update parent directory's dirents to reflect the deletion
    - can mark a dirent as invalid, take a look at what inum is skipped in "dirlookup"

*unlink in Linux will delete the name from the file system, but keep the file object in memory until all references close - not necessary for our purposes