CS 111 Project 2:
Journaling File System

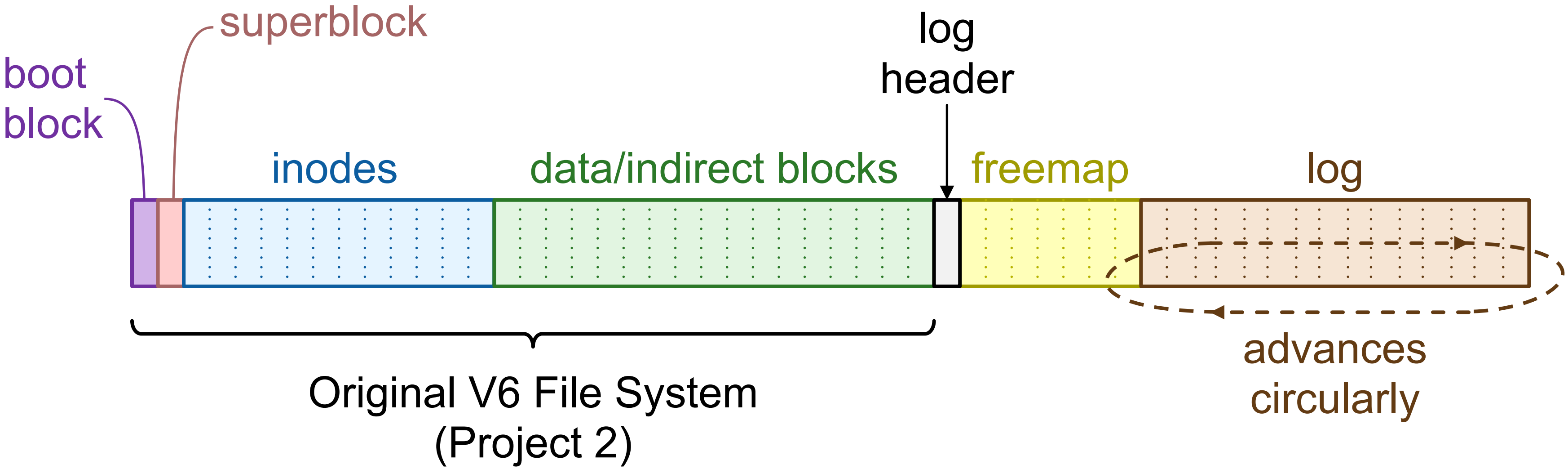
Introduction

- **Your mission: implement part of crash recovery in a journaling file system and explore crash recovery mechanisms**
  - Replay log entries
  - Answer readme questions
- **Not much code to write!**
- **Due Thursday at 11:59pm (late submissions through Saturday)**

Logging Overview

- **Problem: file system operations often require updates to multiple blocks**
  - Example: to create a new file, must
    - Add entry to data block of directory
    - Update directory's inode
    - Write file's inode
- **Potential inconsistencies: system could crash with some (but not all) blocks written to disk**
- **Log allows consistency to be restored quickly after crashes:**
  - Record info about updates in append-only log
  - Identify groups of related ops in log: transactions
  - Make sure log flushed to disk before any affected block
  - After crash, replay all complete transactions from log
- **This implementation logs only metadata (not data of regular files)**

Extended V6 Disk Layout



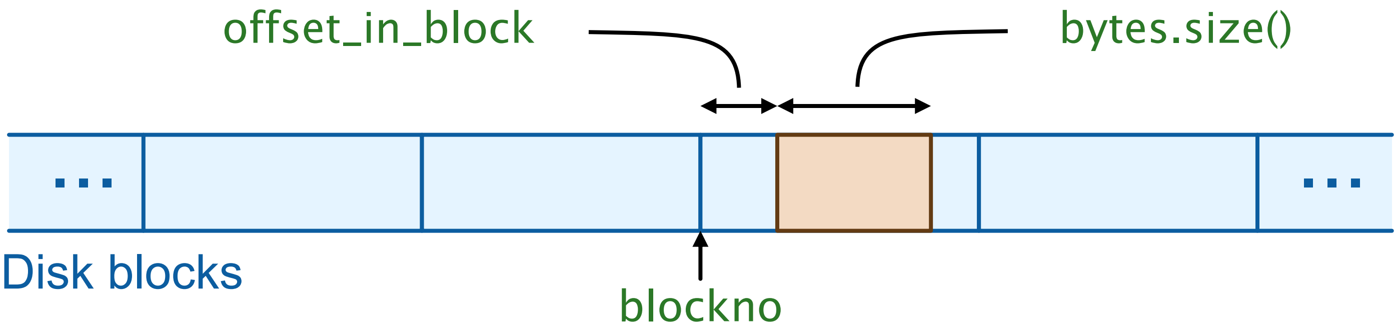
- **Added log storage**
- **Replaced “chunky linked list” of free blocks with bitmap**
  - More modern, efficient
  - Linked list operations don’t work well with log: not idempotent

Log Entries

- **Must be idempotent:**
  - Updates may or may not have occurred to disk blocks before crash
  - Or, system could crash again while replaying log
  - Replaying log entry must work even if disk blocks already updated
- **Example: suppose log entry says "append new entry <name, inumber> to directory?"**
- **For this project, 3 primary log entry types:**
  - Patch bytes
  - Allocate block
  - Free block

LogPatch

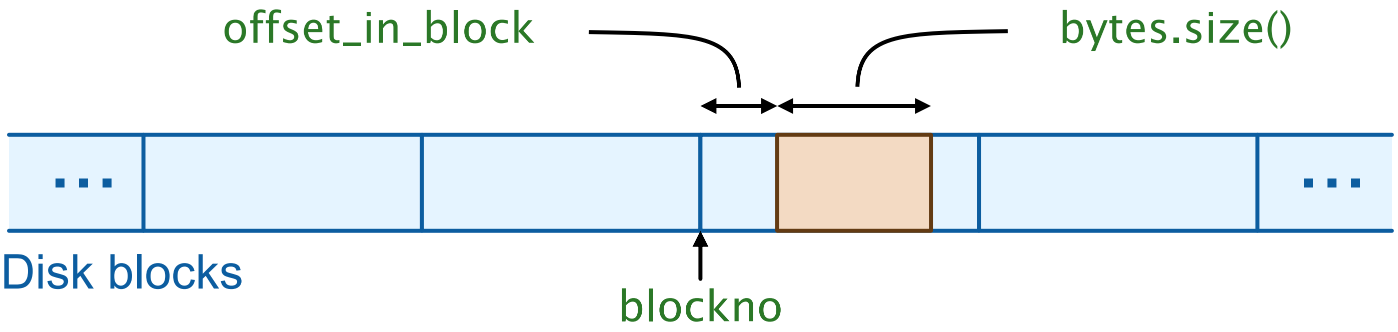
```
struct LogPatch {
    uint16_t blockno;
    uint16_t offset_in_block;
    std::vector<uint8_t> bytes;
};
```



LogPatch

```
struct LogPatch {
    uint16_t blockno;
    uint16_t offset_in_block;
    std::vector<uint8_t> bytes;
};
```

- **Creating a file:**
  - One patch to write new entry in directory
  - One patch to update directory inode
  - One patch to initialize file inode

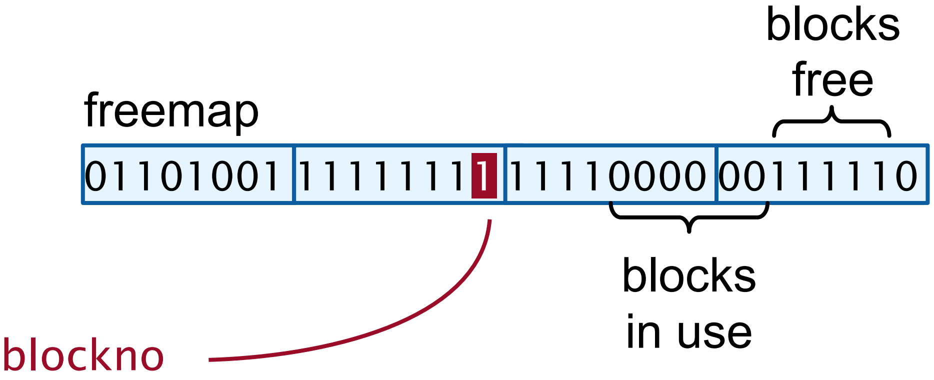


LogBlockAlloc and LogBlockFree

Mark block as either allocated or free:

```
struct LogBlockAlloc {
    uint16_t blockno;
    uint8_t zero_on_replay;
};

struct LogBlockFree {
    uint16_t blockno;
};
```



01101001 11111111 11110000 00111110

Other Log Entries

Mark transaction boundaries:

```
struct LogBegin {
    // No data!
};

struct LogCommit {
    // No data!
};
```

No entries will be replayed from a transaction unless both LogBegin and LogCommit are present

Log wrap-around:

```
struct LogRewind {
    // No data!
};
```

Replaying the Log

- **Code we've written:**
  - Read log info from disk
  - Find the beginning and end of the region to replay, check for consistency
  - Read log entries from disk
  - Make sure each transaction is complete
  - Invoke your code to replay individual entries
- **You write methods in replay.cc to replay each log entry type:**

```
void V6Replay::apply(const LogPatch &);
void V6Replay::apply(const LogBlockAlloc &);
void V6Replay::apply(const LogBlockFree &);
```

Reading and Writing the Disk

```
class V6Replay {
    V6FS &fs_;
    ...
}


struct V6FS {
    ...
    Ref<Buffer> bread(uint16_t blockno);
    Ref<Buffer> bget(uint16_t blockno);
    ...
}
```

- bread and bget both return pointer to a block in the file cache
- bread: read contents of block from disk
- bget: doesn’t bother to read from disk
- Only use bget when you are going to completely overwrite block!!!

Ref<Buffer>

- **Smart pointer:**
  - Use just like Buffer*
  - Maintains a reference count for the cache block
  - Cache block won't be evicted as long as there are Ref's for it

```
struct Buffer : CacheEntryBase {
    char mem_[SECTOR_SIZE];
    …
    void bdwrite();
};
```

- **Can read or write mem_ directly (e.g. memcpy / memset)**
- **Call bdwrite() when finished writing: marks cache block dirty**

Block Allocation Bitmap

```
struct V6Replay {
    ...
    Bitmap freemap_;
    ...
}

if (freemap_.at(blockno)) ...   /* Is block free? */
freemap_.at(blockno) = true;    /* Mark block free. */
freemap_.at(blockno) = false;   /* Mark block in use. */
```

Unlike other parts of the disk, the Bitmap is entirely loaded into memory

Check out the implementation of `Bitmap` in `bitmap.hh`!

- How does it allow individual bits to be addressed?

Part 2: Short Answer

- **Exploration of included tools like:**
  - `dumplog` to print out the log
  - `fsck` to check image for consistency
  - `mountv6` to mount a filesystem image to try out
- **Spec walks through how to use them and what to look for**
- **Demos from lecture and section may also be helpful**

Part 3: Ethics and Trust

- **OS runs commands in a privileged ‘kernel’ mode that users cannot**
- **What if a user could execute such commands directly?**
  - Can we trust the system with private files and confidential information?
- **What implicit trust do we have in OSes when we use them?**
- **What can users and OS developers do about this?**

Project Infrastructure

- **Based on FUSE (File System in User space):**
  - File system code runs in a user application
  - Linux kernel forwards file system requests to the application
  - Result: a fully-functional file system!
- **Check out the extra assign2 infrastructure design page for lots of cool (optional) info on how all this works**

Questions?