



The xv6 Filesystem



Reference

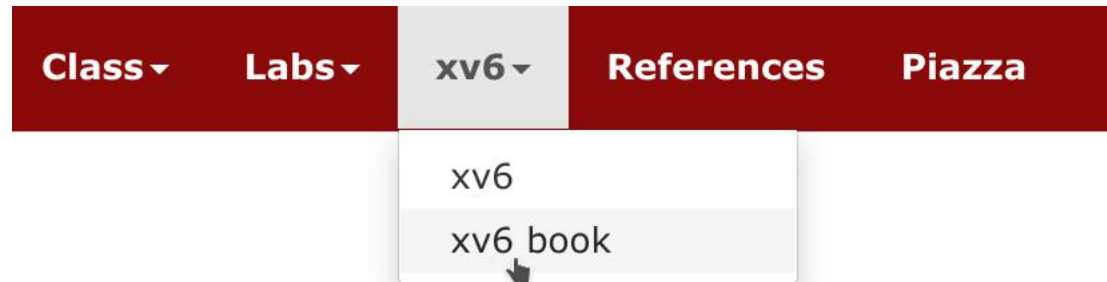
- The xv6 book
 - Chapter 8 “File system”
- <https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf>
- For latest version, google “mit xv6”

<https://pdos.csail.mit.edu> › 6.828 ⋮

[6.S081 / Fall 2021 - MIT PDOS](#)

Sep 22, 2021 — Separately, **6.828** will be offered in future terms as a graduate-level seminar-style class focused on research in operating systems.

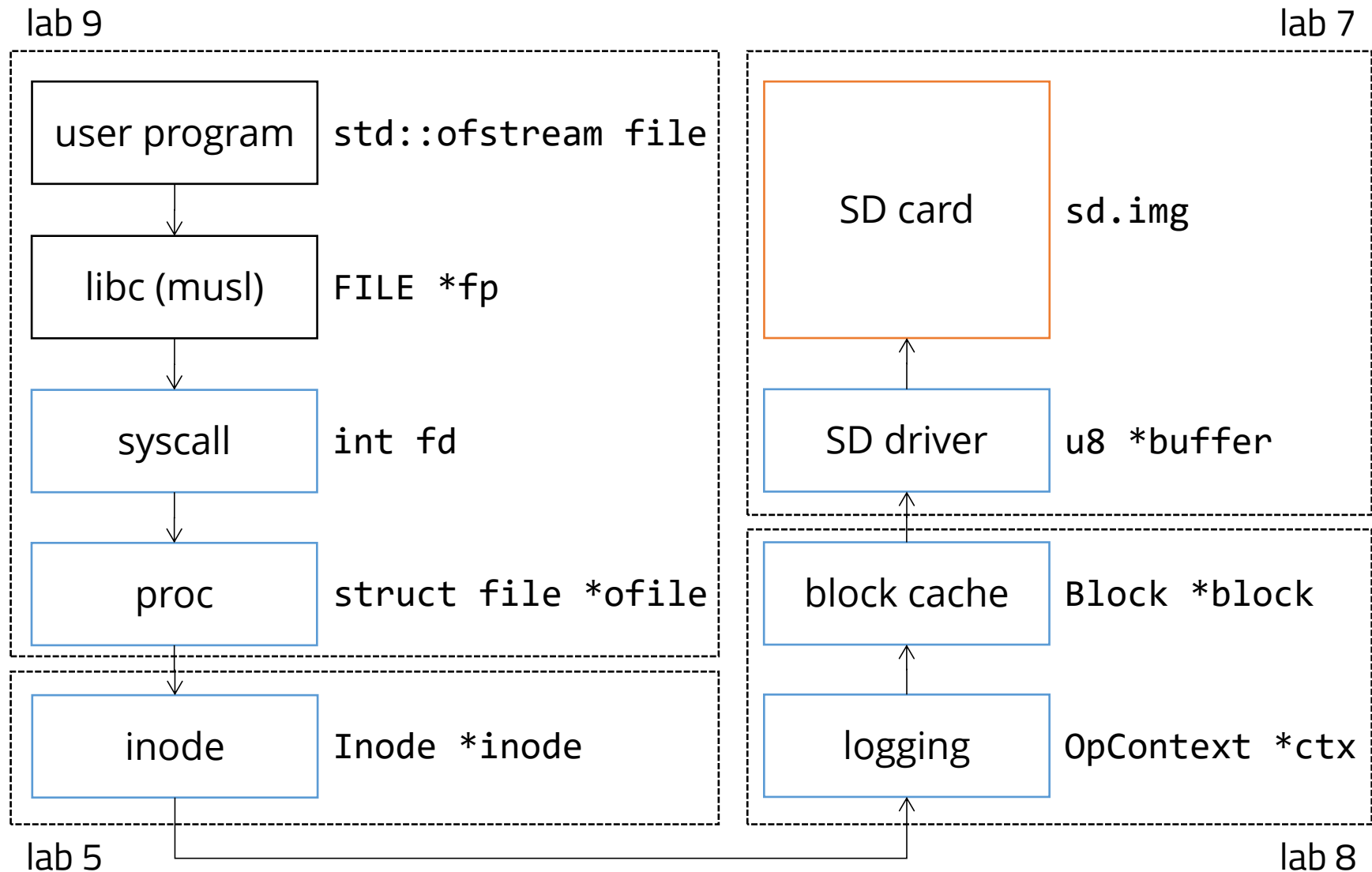
[References](#) · [Lab 1](#) · [Xv6](#) · [Lab 5](#)



Features

- Unix-like file and directory interface
 - Use inode, dirent, stat, ...
- One directory tree
 - No "C:", "D:", "E:", "F:"
- One disk storage
 - No mount points
- Concurrent access
 - Protect every data structure with lock and reference count
 - Fine-grained locks for path name lookup
- Crash consistency
 - With write-ahead logging, or journaling
 - No checksum, no rollback
- Block cache
 - No disk IO scheduler
- Asynchronous EMMC SD read/write

Overview



Synchronous API

- Block process when processing
 - Multiple processes can issue concurrent requests
- Only return when work is done
- The underlying implementation can be asynchronous
 - Submit work and wait for interrupts or notifications
- Ambitious student can develop an asynchronous framework
 - epoll, aio, io_uring, kqueue

```
typedef struct {  
    // read `BLOCK_SIZE` bytes in block at `block_no` to `buffer`.  
    // caller must guarantee `buffer` is large enough.  
    void (*read)(usize block_no, u8 *buffer);  
  
    // write `BLOCK_SIZE` bytes from `buffer` to block at `block_no`.  
    // caller must guarantee `buffer` contains at least `BLOCK_SIZE` bytes.  
    void (*write)(usize block_no, u8 *buffer);  
} BlockDevice;
```

Atomic Operation

- Writing a word in memory is often atomic
 - 8 bytes on 64-bit platforms
 - Multi-copy atomicity
- Writing a block (512 bytes) on disk is often atomic
 - For historic reasons, they choose this value
- We often manipulate multiple blocks simultaneously
 - write: write out new blocks, update inode and bitmap
 - rename: OSTEP section 39.8 “Renaming Files”
- Put a series of operations into one “transaction”
 - Cannot be partially done
 - Not database transactions, only atomicity is guaranteed

```
// begin a new atomic operation and initialize `ctx`.
// `OpContext` represents an outstanding atomic operation. You can mark the
// end of atomic operation by `end_op`.
void (*begin_op)(OpContext *ctx);

// end the atomic operation managed by `ctx`.
// it returns when all associated blocks are synchronized to disk.
void (*end_op)(OpContext *ctx);
```

Atomic Operation

- begin_op/end_op cannot be nested
 - Deadlock
- begin_op/end_op is prone to forgetting
 - Maybe you forget to lock last week
- Function that may write must carry OpContext *ctx
 - It's a remainder to caller
 - ctx can be NULL, which means direct write
 - Direct write is dangerous, which may break atomicity

```
// synchronize the content of `block` to disk.  
// `ctx` can be NULL, which indicates this operation does not belong to any  
// atomic operation and it immediately writes all content back to disk. However  
// this is very dangerous, since it may break atomicity of concurrent atomic  
// operations. YOU SHOULD USE THIS MODE WITH CARE.  
// if `ctx` is not NULL, the actual writeback is delayed until `end_op`.  
void (*sync)(OpContext *ctx, Block *block);
```

Fail-Stop Model

- SD card is fail-stop
 - If one operation fails, no subsequent operation can complete
 - No data corruption
- Filesystem module is fail-stop
 - Any unrecoverable fault results in kernel panic
 - Out of memory \Rightarrow panic
 - Disk full \Rightarrow panic
 - Invalid memory access \Rightarrow panic
 - Assertion failure \Rightarrow panic
 - Invalid parameter \Rightarrow panic
 - File name not found \Rightarrow return "NOT FOUND"
 - Summary: DO NOT return error codes, just panic
- Syscall interface should check user input
 - Reject them with return value

Locking

- Hold the lock before doing everything
 - Unless told not to do
 - `inodes.put`
- Use SleepLock
 - Currently no `sleep/wakeup`, we will fix it later

Reference Count

- The number of pointers pointed to an object
 - `increment_rc/decrement_rc`
 - `get/put`
- When reference count goes to zero, the object can be freed
 - `decrement_rc` will return true if this happens
- Hold the lock while modifying reference count
 - Unless you want to design lock-free algorithms
- See `common/rc.h`

Put It Together

- You will do it in lab 9

```
auto *p = inodes.get(ino);
OpContext _ctx, *ctx = &_ctx;
bcache.begin_op(ctx);
inodes.lock(p);
inodes.write(ctx, p, buf, 0, max_size);
inodes.unlock(p);
inodes.put(ctx, p);
bcache.end_op(ctx);
```

Reference

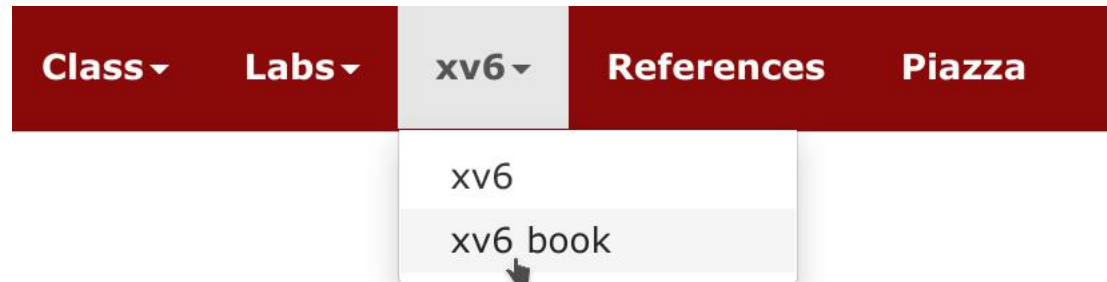
- The xv6 book
 - Chapter 8 “File system”
- <https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf>
- For latest version, google “mit xv6”

<https://pdos.csail.mit.edu> › 6.828 ⋮

[6.S081 / Fall 2021 - MIT PDOS](#)

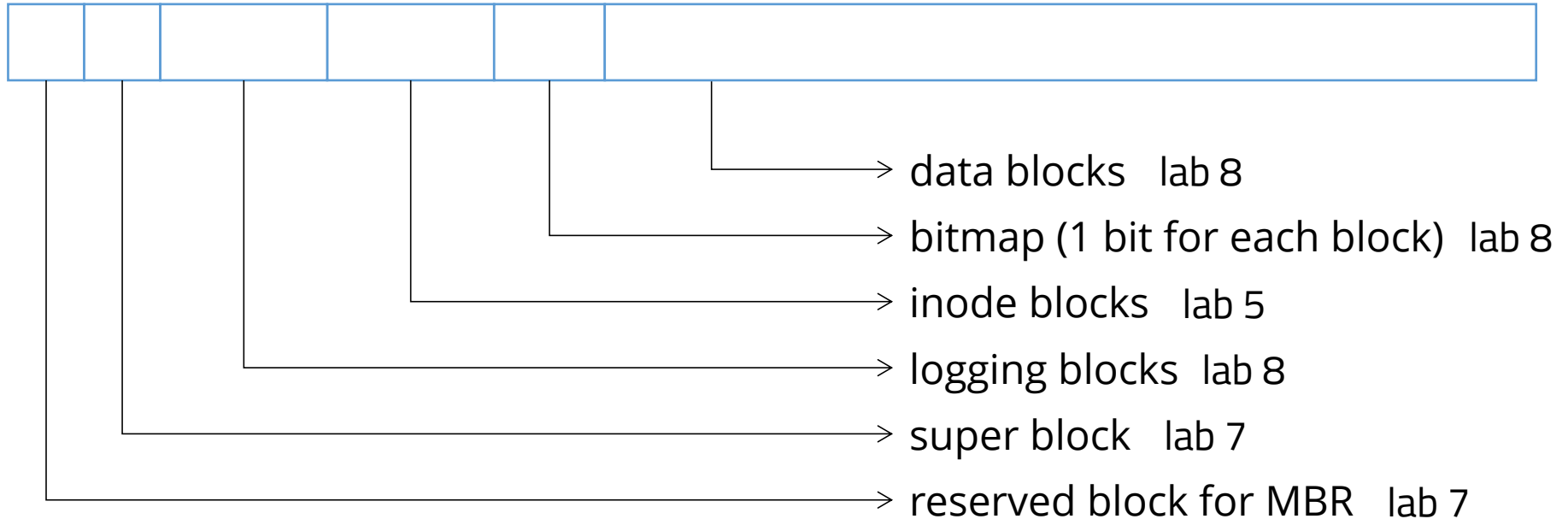
Sep 22, 2021 — Separately, 6.828 will be offered in future terms as a graduate-level seminar-style class focused on research in operating systems.

[References](#) · [Lab 1](#) · [Xv6](#) · [Lab 5](#)



On-Disk Layout

- fs/defines.h



```
typedef struct {  
    u32 num_blocks; // total number of blocks in filesystem.  
    u32 num_data_blocks;  
    u32 num_inodes;  
    u32 num_log_blocks; // number of blocks for logging, including log header.  
    u32 log_start; // the first block of logging area.  
    u32 inode_start; // the first block of inode area.  
    u32 bitmap_start; // the first block of bitmap area.  
} SuperBlock;
```

Inode Tree

- Metadata are stored in inodes
 - Directory
 - Regular file
 - Device file: `/dev/tty`, `/dev/null`
- Inodes are stored consecutively on disk
 - Each one has an inode number (`inode_no`)
 - Each block contains `INODE_PER_BLOCK` inodes
- Usually inodes form a tree
 - Leaves are files
 - Internal nodes are directories
 - It becomes a DAG if OS supports hard link
- Inode contains binary data
 - Directory: directory entries
 - Regular file: file content
 - Device file: ?

Inode

```
// inode types:
#define INODE_INVALID    0
#define INODE_DIRECTORY 1
#define INODE_REGULAR    2 // regular file
#define INODE_DEVICE     3

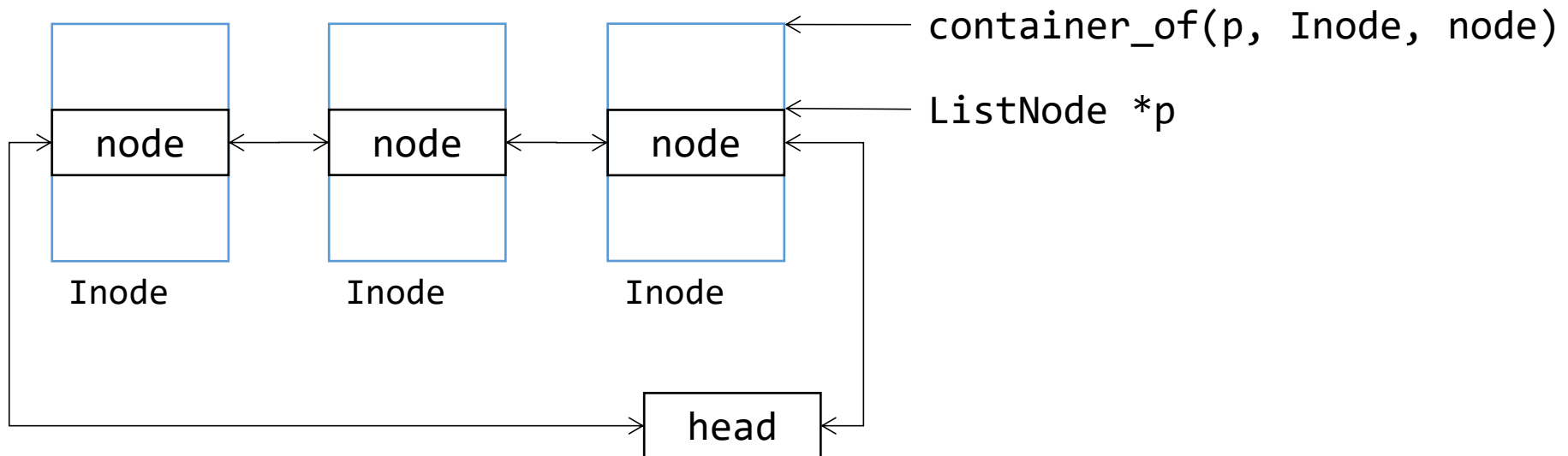
typedef struct {
    InodeType type;
    u16 major;
    u16 minor;
    u16 num_links;
    u16 num_bytes;
    u32 addrs[INODE_NUM_DIRECT];
    u32 indirect;
} InodeEntry;
```

Inode Lifetime

- `alloc`: the birth on disk
- `get`: in-memory pointer to inode
- `lock/unlock`: in-memory locking
- `put`: destroy in-memory pointer
 - If there's no pointer and no hard link to it, free it on disk
 - `rc.count == 0 && num_links == 0`
 - \Rightarrow `put` may write something to disk
- You need to maintain a pool of in-memory inodes
 - Use memory pool to allocate new inode
 - Use linked list to search through
 - Advanced: SLAB, hash map and binary search tree

Arena: Object Memory Pool

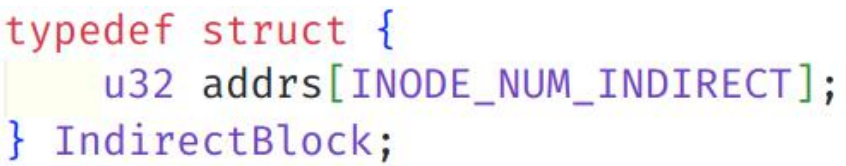
- `core/arena.h`
- A simple constant time object allocator
- See `arena_test` for usage



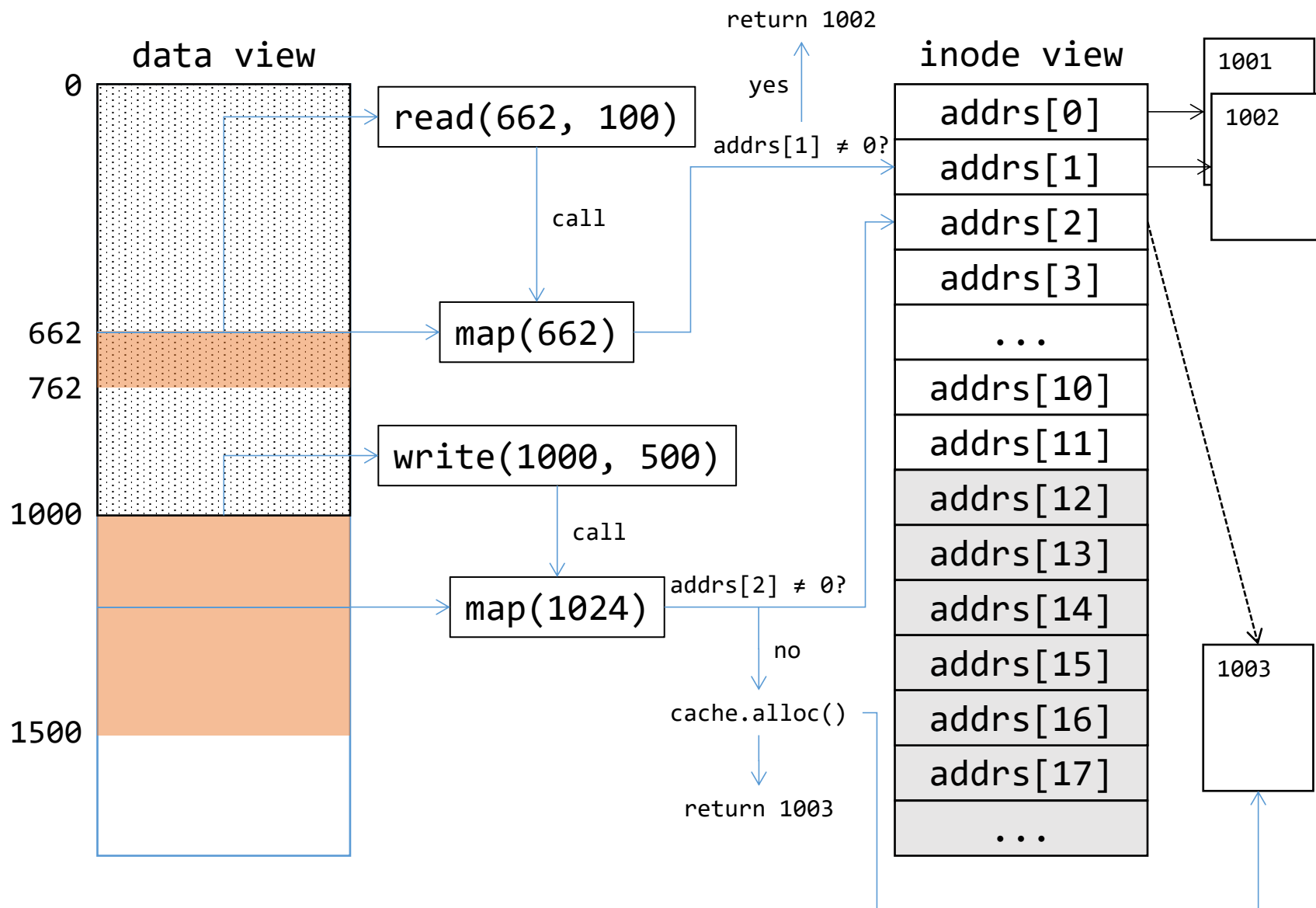
Inode Data Management

- Regular File:
 - sync: read data from disk or write dirty data to disk
 - clear: discard all binary data
 - read: read existing data
 - write: overwrite or append data
- Directory:
 - Directory is a special file: an array of entries
 - lookup: search for entry with specified name
 - insert: add a new entry
 - remove: erase entry at specified index
 - Just like `std::vector<DirEntry>`

```
// directory entry. `inode_no == 0` implies this entry is free.
typedef struct {
    u16 inode_no;
    char name[FILE_NAME_MAX_LENGTH];
} DirEntry;
```



inode_{read,write,map}



Block Cache Interface

- `fs/cache.h`
- Inodes read/write/allocate/free blocks via block cache
 - Block cache manages on-disk bitmap and in-memory block pool
- Before you operate on a block, you should lock it
 - `cache->acquire`: return a locked block
 - `cache->release`: unlock and return the block back to cache
- Whenever you need atomicity, `begin_op/end_op` can help
- Block cache does not have a “write” interface
 - It's sync
 - Actual write can be delayed, depending on your context

Inode Unit Testing

- fs/test
- Written in C++
 - Each test is running in a forked child: not friendly to debugger
 - Use C++ exceptions to report error
- Underlying block cache is mocked
 - Along with some OS functionalities, e.g. locks
- Run the test:
 - `cd src/fs/test`
 - `mkdir build; cd build`
 - `cmake ..`
 - `make && ./inode_test`
- Debug tools:
 - `coredumpctl gdb`: open coredump after test crashed
 - PAUSE: inject static breakpoint
 - `gdb -p [pid]`: attach to a running test process
 - `kill -CONT [pid]`: continue running after pause

```
inodes.lock(p);  
mock.begin_op(ctx);  
PAUSE;  
inodes.clear(ctx, p);  
inodes.unlock(p);  
mock.end_op(ctx);
```

Reference

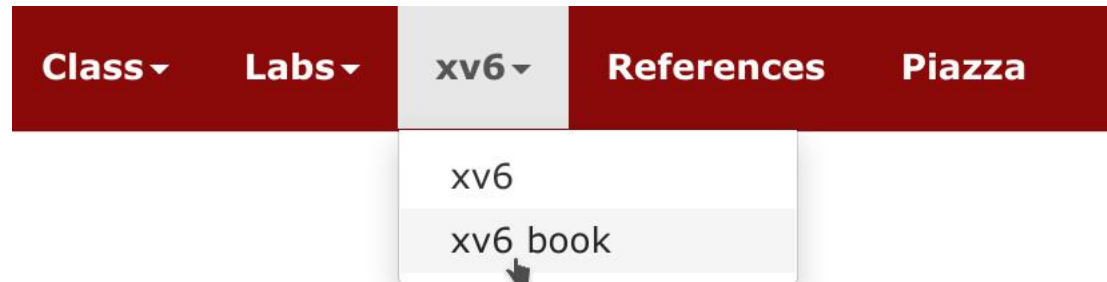
- The xv6 book
 - Chapter 8 “File system”
- <https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf>
- For latest version, google “mit xv6”

<https://pdos.csail.mit.edu> › 6.828 ⋮

[6.S081 / Fall 2021 - MIT PDOS](#)

Sep 22, 2021 — Separately, **6.828** will be offered in future terms as a graduate-level seminar-style class focused on research in operating systems.

[References](#) · [Lab 1](#) · [Xv6](#) · [Lab 5](#)



Naïve Block Layer

- API: read, write
- Slow
 - Need interrupts and large data copying
 - Batch? Asynchronous?
- Data race
 - Some are false data races: two inodes on the same block
- No atomicity guarantee
 - A database jargon
 - All or nothing
 - In our case, crash consistency
 - We use locks at higher layers to provide isolation & consistency
- Our solution: cache + lock + log

Block Cache

- API: read → get, write → sync, put
- Cache what you read in memory
- Directly modify in memory
- Delayed write back
 - Explicit write back with `sync(NULL, ...)`
- Try to keep cache small
 - Evict when `#cached` is larger than `EVICTIION_THRESHOLD`
 - It's a soft limit, not hard limit
 - LRU policy: move to front at get
- `get` returns a pointer to a block inside cache
- Implementation: organize blocks as a list

Per-Block Lock

- Use sleep lock
- API:
 - `acquire` \leftarrow `get` + `lock`
 - `release` \leftarrow `put` + `unlock`

Crash Consistency

- Writing a single block is safe
- Implement at inode layer: fsck
 - Full disk scan
 - Only checks metadata
- General solution: logging = journaling
 - Reserve a space on disk for log entries
 - Group multiple writes in one atomic operation
- Two-phase checkpoint
 - Write the entire operation to log first
 - Then copy to their original locations
- Crash at log write: this operation simply fails
- Crash at copying: contents still in log, replay

Atomic Operation

- New API: `begin_op`, `end_op`
 - `end_op` returns when entire operation is persisted to disk
 - **NOTE:** this is different from xv6!
- Track write set in `OpContext`
 - `sync(ctx, ...)`
- Multiple writes to the same block as a single write
 - We call it *local absorption*
- All writes must be delayed until `end_op`
 - Pinned blocks: you must not evict them in cache
- Concurrent operations?
 - Log entries are not unlimited
 - Different operations may touch the same block
 - Naïve way: one by one
 - Our solution: log reservation + group checkpoint

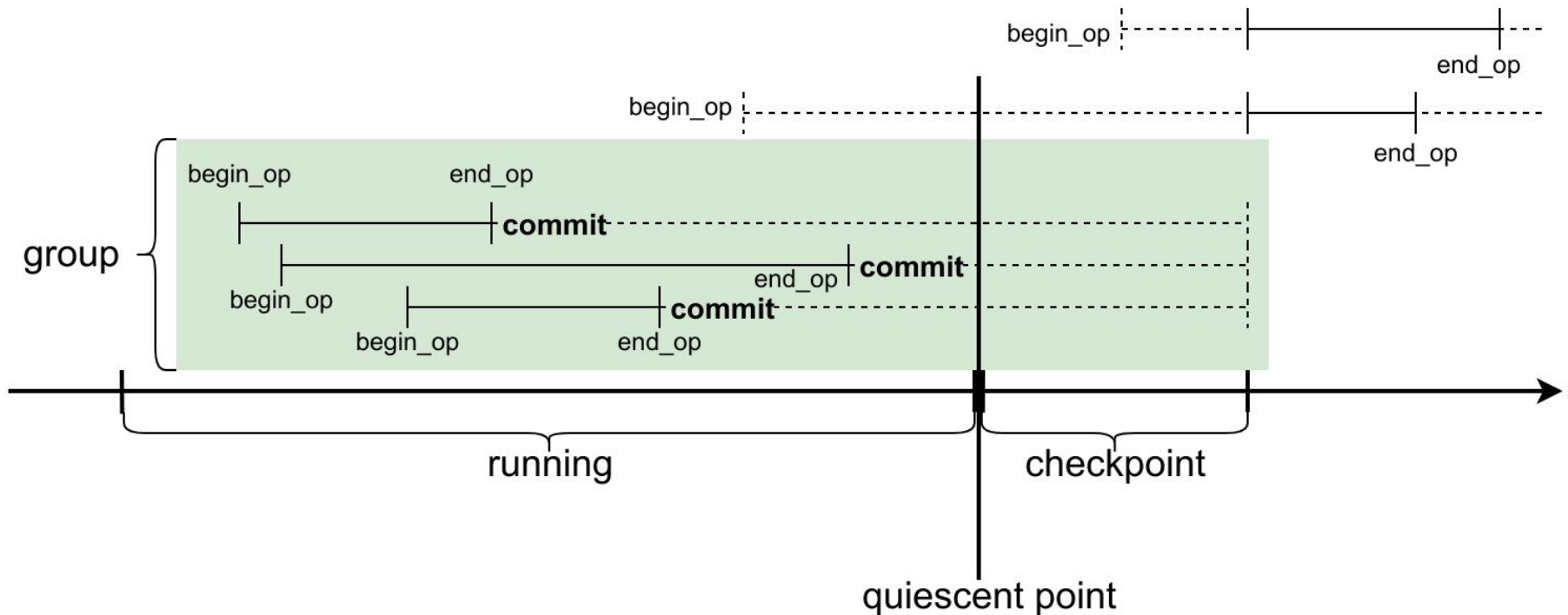
Log Reservation

- We assume an atomic operation can track at most n blocks
 - $n = \text{OP_MAX_NUM_BLOCKS}$
- So reserve at least n log entries before start
- In `end_op`, release log entries that are not used
- If there's no enough log entries to reserve: sleep
 - wakeup when `log_used` changes

```
while (log_used + OP_MAX_NUM_BLOCKS > log_size) {  
    sleep(&log_used, &lock);  
}
```

Group Checkpoint

- Different operations may touch the same block
 - Write them together
- When: no outstanding operation
 - At *quiescent point*
- Who: the last one who commits



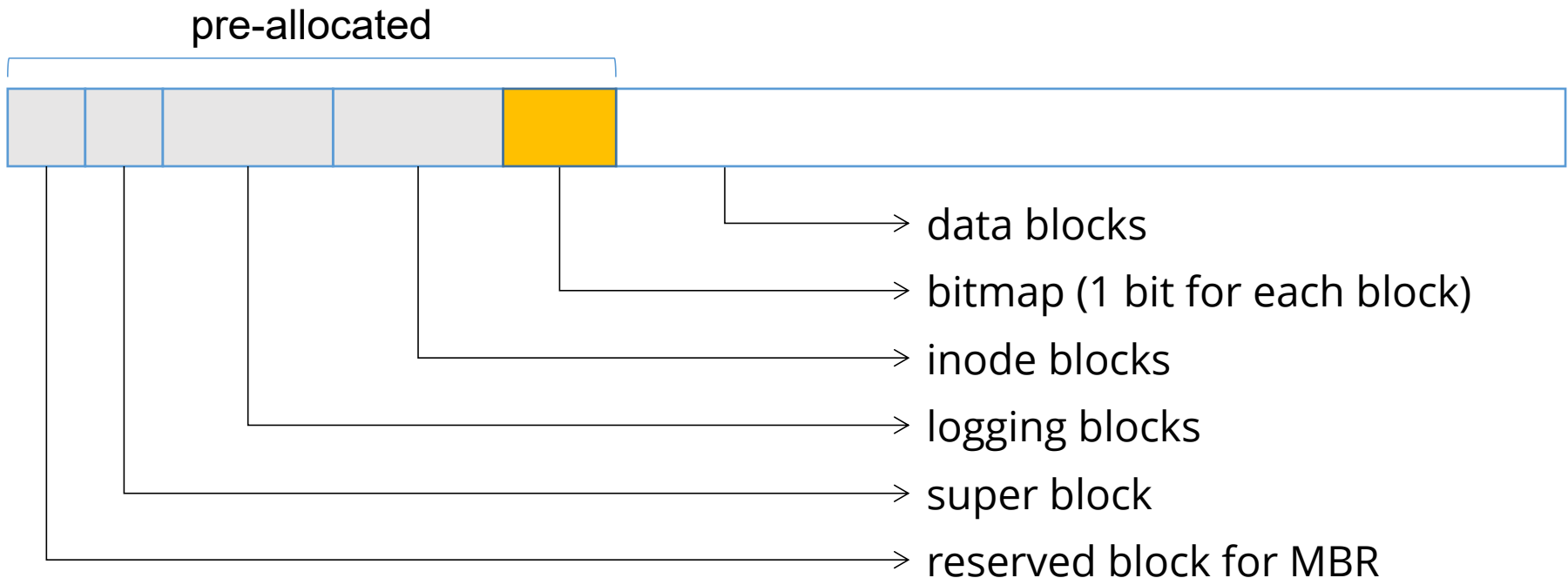
Lifecycle of Atomic Operation

- Running: after `begin_op`, before calling `end_op`
 - You can call `sync` in this state
- Committed: already called `end_op`, but `end_op` still waits
 - Associated blocks have been tracked in global log header
- Checkpointed: all blocks have been persisted
- `begin_op(reserve)[→sync]*→end_op(commit→checkpoint)`
- `commit`: set block numbers in log header
 - Global absorption: multiple same block numbers can be merged
- `checkpoint`: 4 steps
 - Write blocks to log area
 - Write log header
 - Copy blocks to original locations
 - Reset log header

```
typedef struct {  
    usize num_blocks;  
    usize block_no[LOG_MAX_SIZE];  
} LogHeader;
```


Bitmap

- Every block has a bit in bitmap area
 - Even for bitmap itself
- Some blocks are *pre-allocated*
 - mkfs will set their bits for us
- We only need to manage data blocks
 - API: alloc, free
 - A good exercise to use block cache interface :)



SD Read

