Caching and Layered Disk Devices

CS 4411

Spring 2020

Announcements

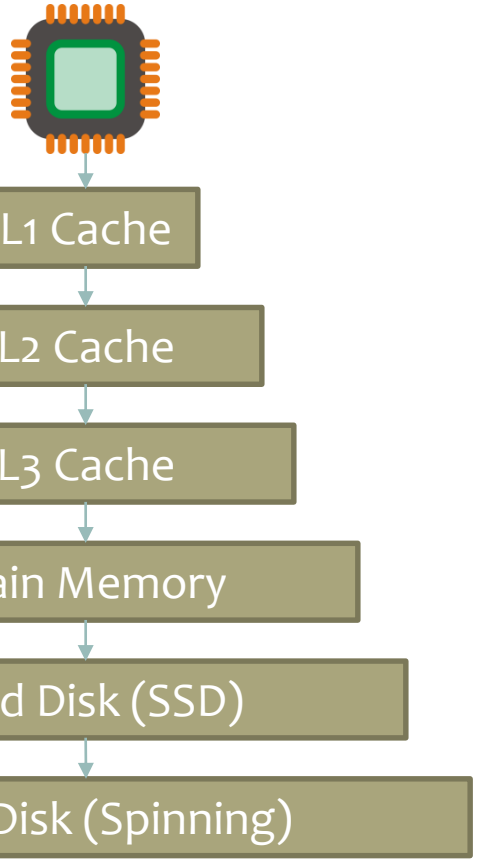
- EGOS Code Updated
- Impending Cornell shutdown
- Next lecture will be conducted via Zoom
- Office hours will be Zoom meetings hosted by TAs
- Links will be posted to Piazza

Outline for Today

- Block Cache Design
  - Memory hierarchy
  - Disk blocks and block cache
  - Write-Through vs. Write-Back
- The EGOS storage system
  - Block devices
  - Layering
  - Code details

Memory Hierarchy

Access Time	Size
4 cycles (1 ns)	64 KB
12 cycles (4 ns)	256 KB
42 cycles (14 ns)	9 MB
75 ns	16 GB
100 μs	256 GB
10 ms	1 TB

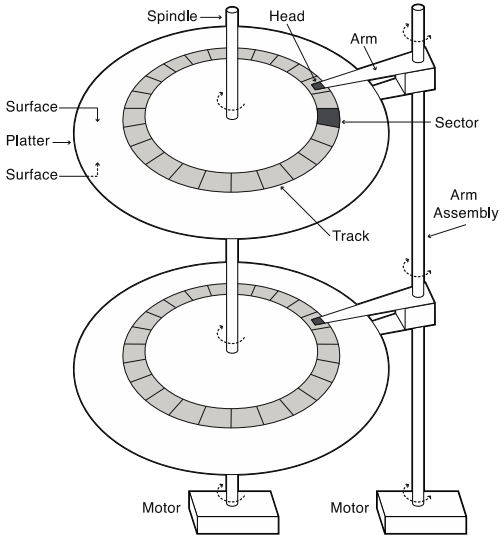


Outline

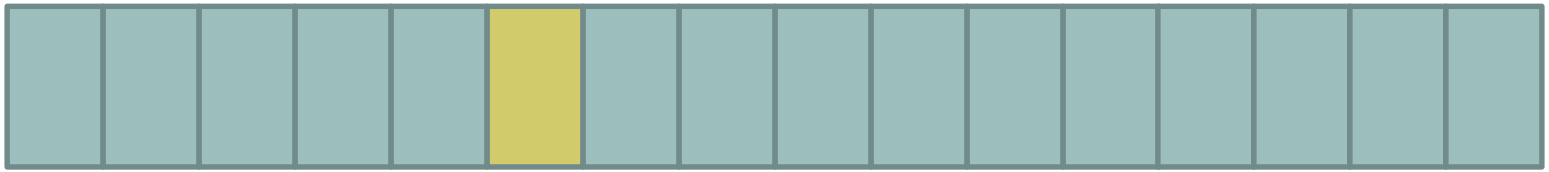
- Block Cache Design
  - Memory hierarchy
  - **Disk blocks and block cache**
  - Write-Through vs. Write-Back
- The EGOS storage system
  - Block devices
  - Layering
  - Code details

Hard Disk Abstraction

- Disk drivers provide read/write operations in units of **blocks**
- Usually 512 bytes, based on sector size of a spinning disk
- File system stores files in groups of blocks

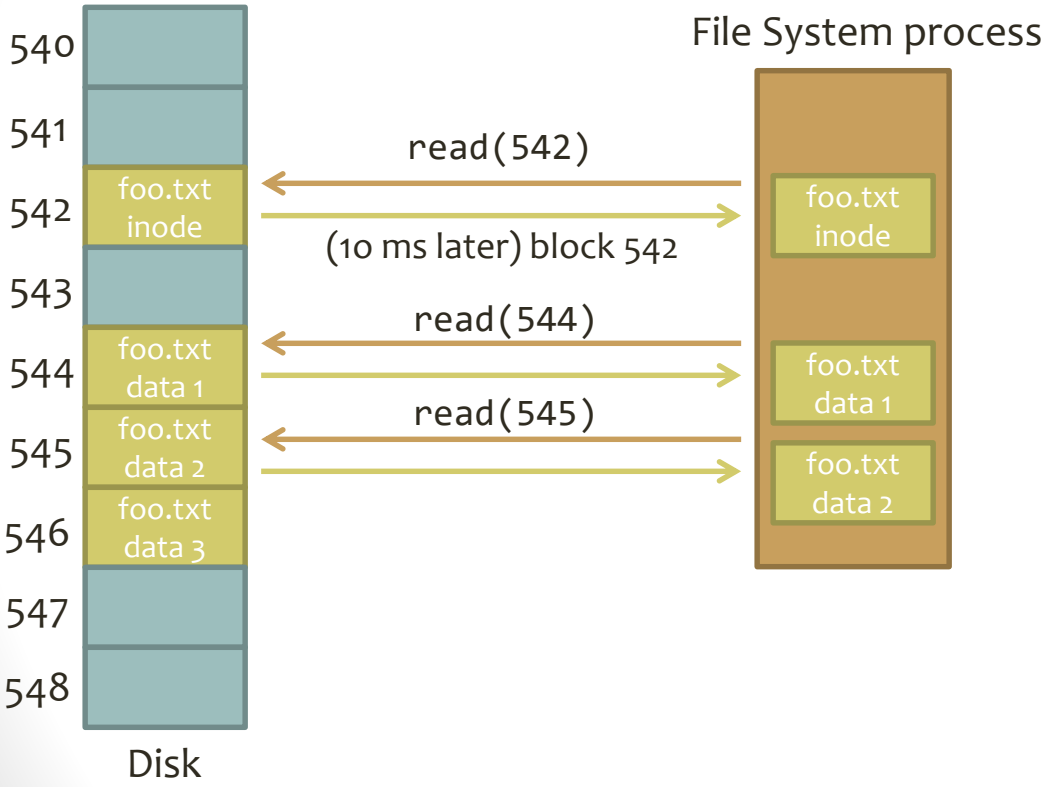



```
write(5, 128, &buf)
```



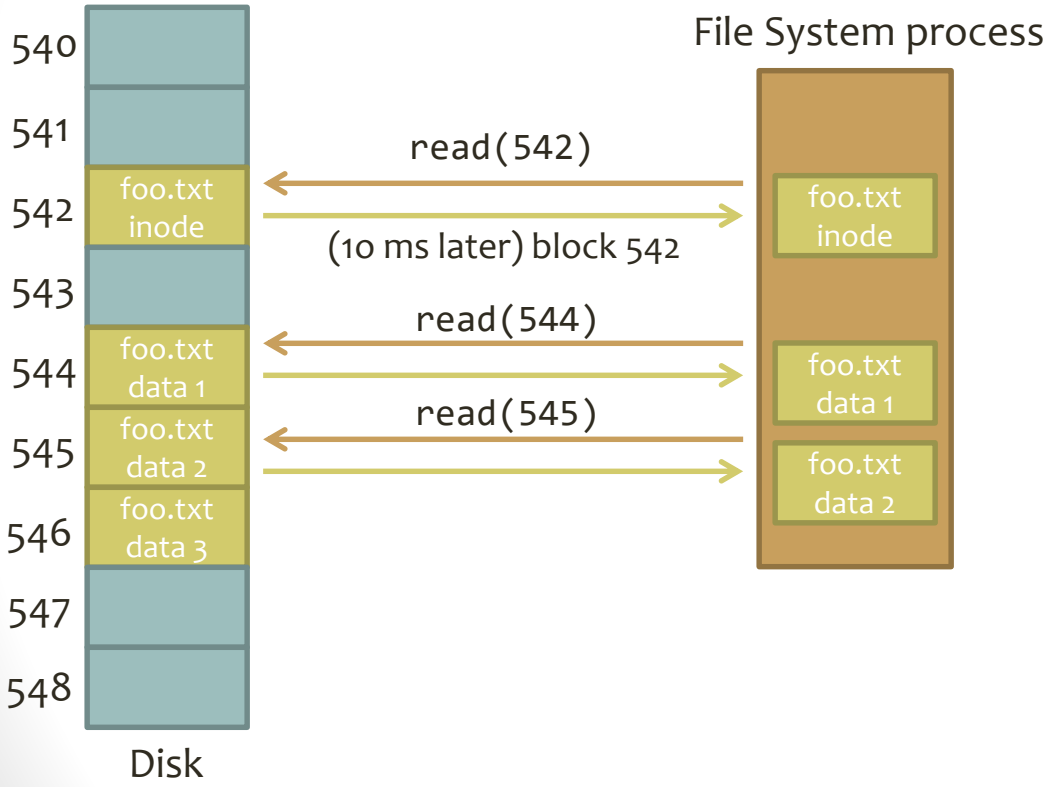
Block # 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...

Operations to Read from a File



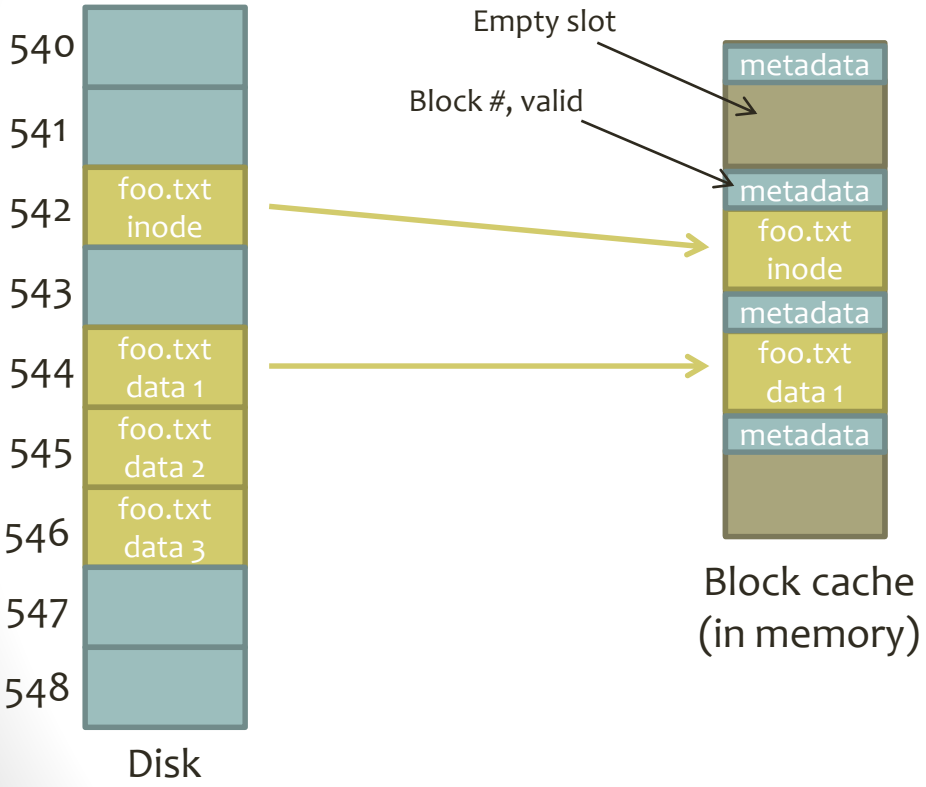
1. Read file's inode into memory
2. Get location of data blocks from inode
3. Read each data block in range of read request into memory
4. Respond to file read request

Operations to Read from a File



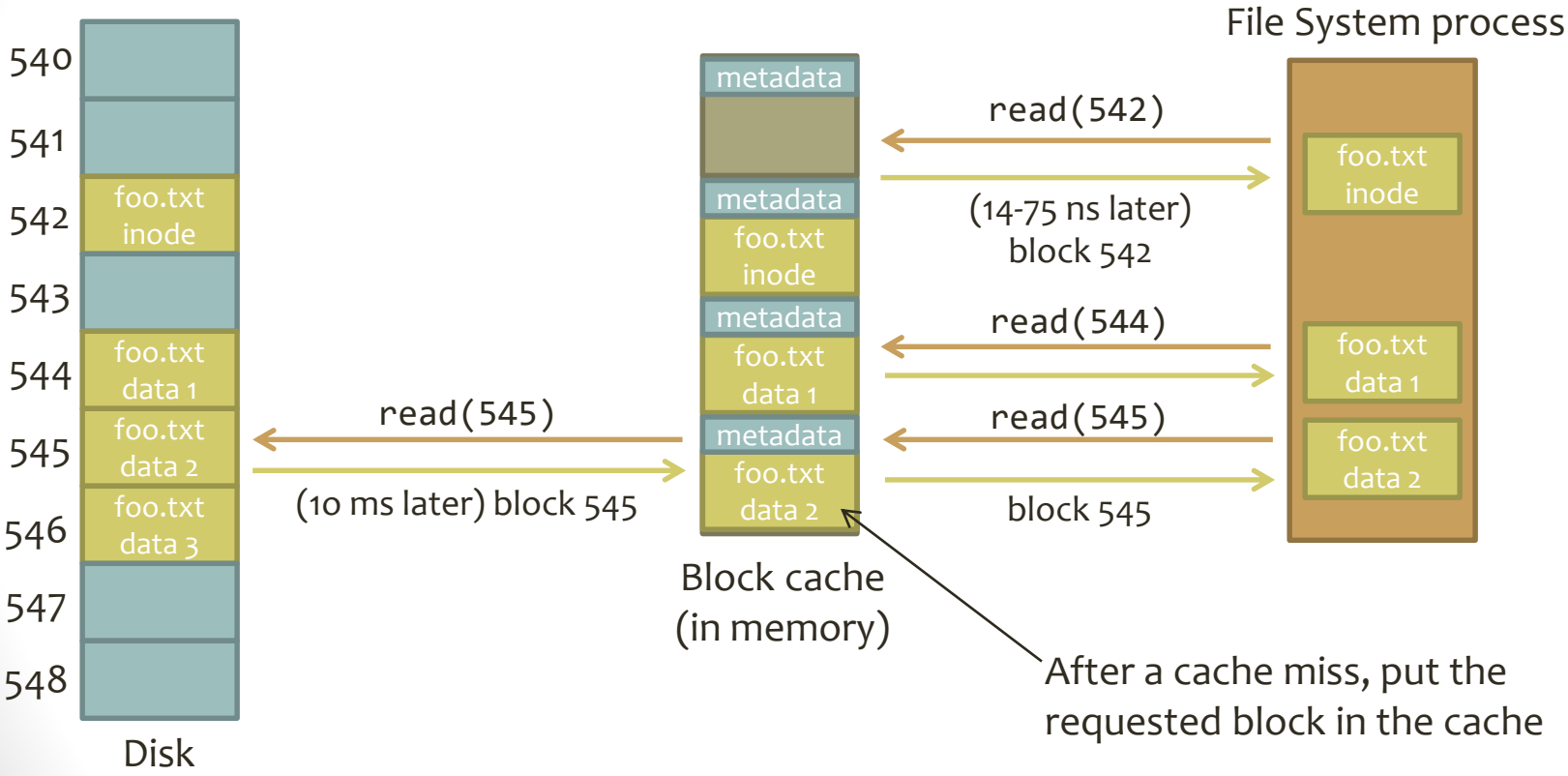
- How long does it take to read 1000 bytes from the file?
- What happens when we get another read request for the same file?
- Why is this inefficient?

The Block Cache

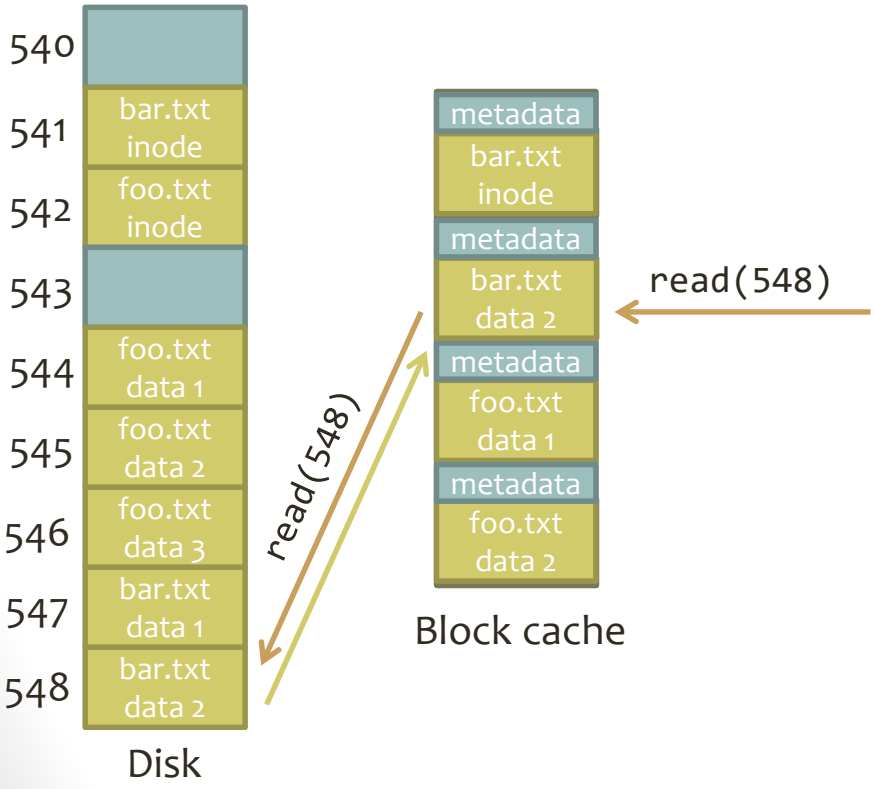


- Store recently-used disk blocks in memory
- Cache entry metadata indicates which block it caches (if any)
- Reading a cached block is a memory access, not a disk access

Using the Block Cache



Cache Eviction

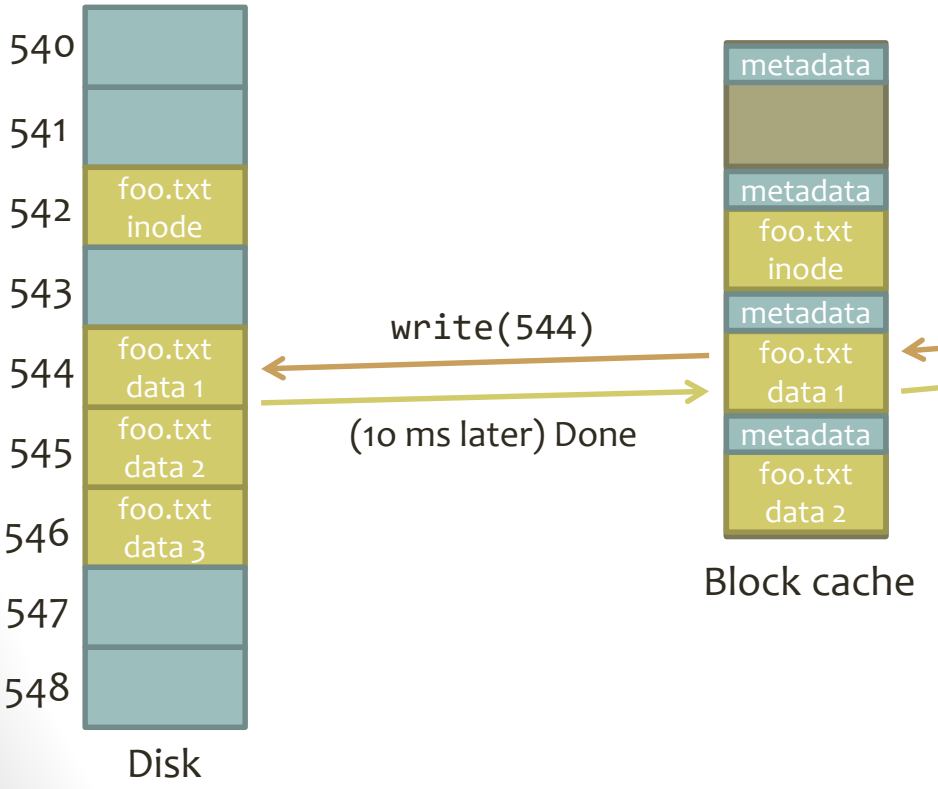


- What if the cache is full and a process needs to read a new block?
- Choose a block to **evict** based on an eviction algorithm
  - LRU, LFU, CLOCK, etc.
  - Block cache service must keep state for this algorithm
- This assignment: CLOCK

Outline

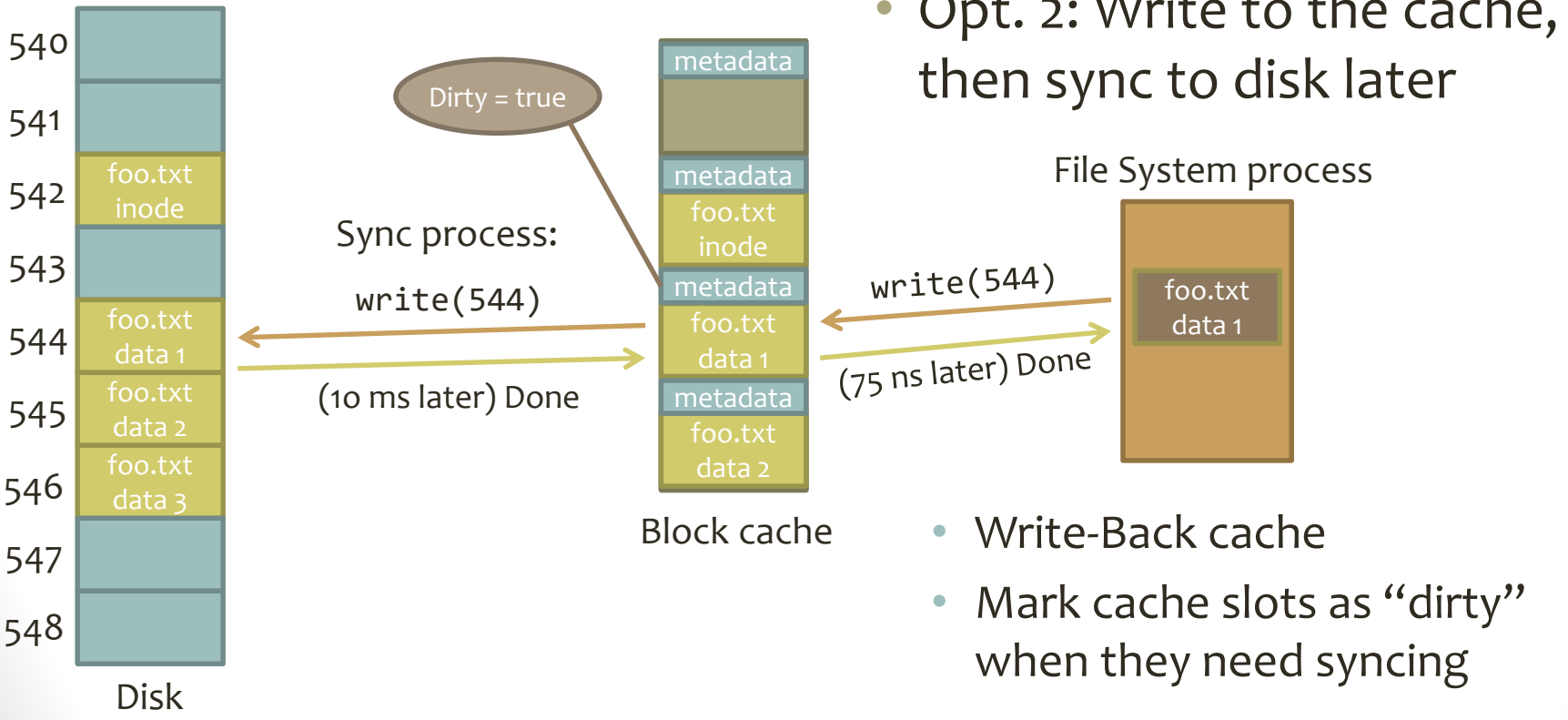
- Block Cache Design
  - Memory hierarchy
  - Disk blocks and block cache
  - **Write-Through vs. Write-Back**
- The EGOS storage system
  - Block devices
  - Layering
  - Code details

Handling Writes



- What if a process writes to a block that's in the cache?
- Opt. 1: Forward the write to the disk now
  - Write-Through cache

Handling Writes



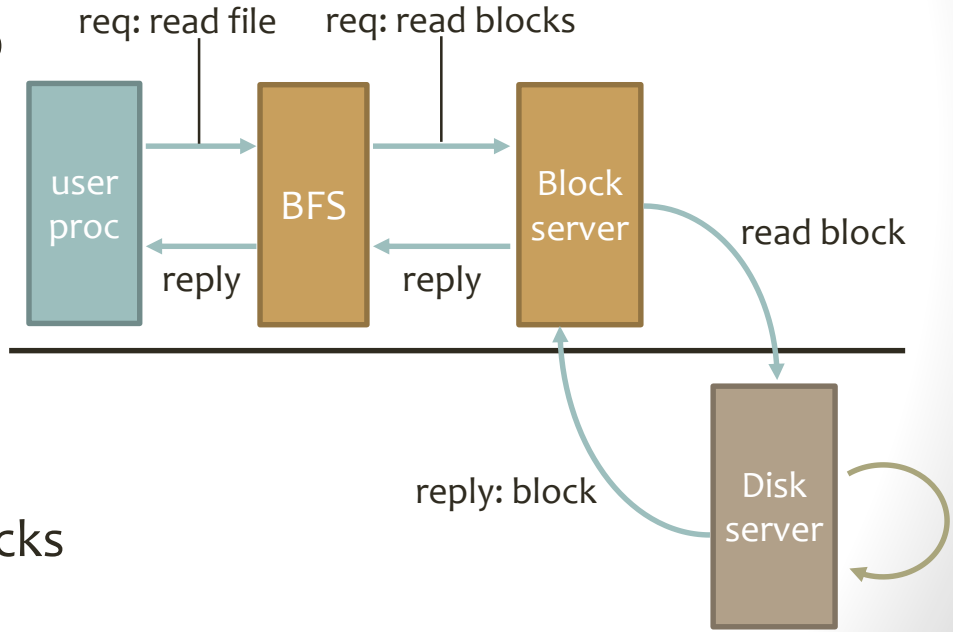
- Opt. 2: Write to the cache, then sync to disk later
  - Write-Back cache
  - Mark cache slots as "dirty" when they need syncing

Outline

- Block Cache Design
  - Memory hierarchy
  - Disk blocks and block cache
  - Write-Through vs. Write-Back
- **The EGOS storage system**
  - Block devices
  - Layering
  - Code details

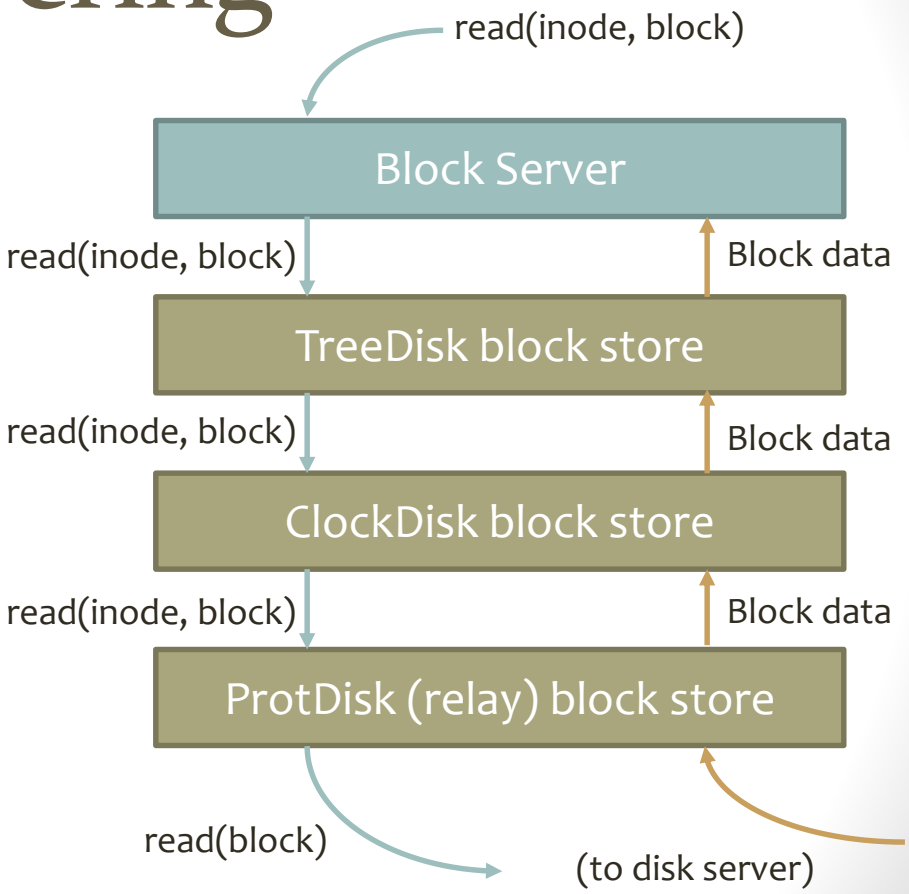
Storage in EGOS

- Disk server (disksvr.c) reads and writes blocks to HW
- Block server (blocksvr.c) also reads and writes blocks
  - Forwards requests to disk server (eventually)
  - Blocks are grouped by **inode #**
- File server (bfs.c) stores files in sequences of blocks
  - Each file has an inode for its blocks



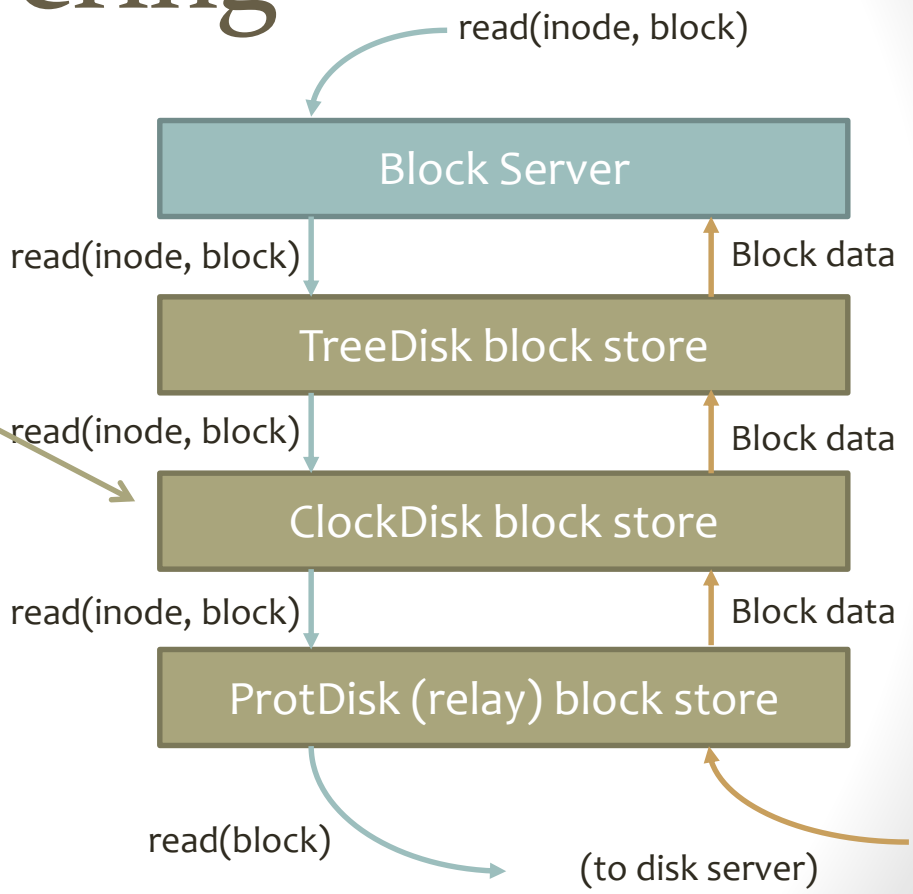
Block Service Layering

- Within the block server, a **stack** of block stores
- Each block store has the same interface
- Block server sends requests to top of stack
- Each block store knows the block store below it, can "pass through" read and write operations



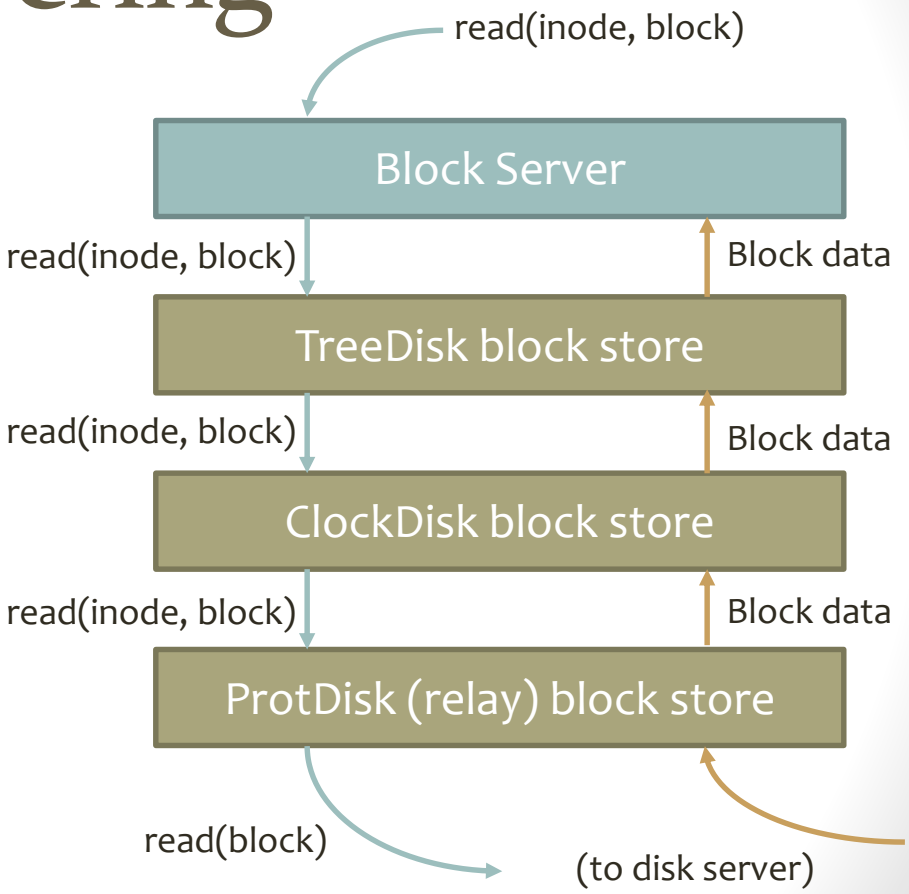
Block Service Layering

- This is how EGOS adds a block cache – it's a block store layer!
- Reads **don't** have to be forwarded if the block is in the cache



Block Service Layering

- Important: Each block store can have its own interpretation of inode numbers
- In **TreeDisk**, inodes track groups of blocks belonging to the same file
- In **ProtDisk**, inodes represent disk partitions on the underlying disk server
  - Right now there's only one, so all ProtDisk ops have inode = 0



Outline

- Block Cache Design
  - Memory hierarchy
  - Disk blocks and block cache
  - Write-Through vs. Write-Back
- **The EGOS storage system**
  - Block devices
  - Layering
  - **Code details**

Adding "Objects" to C

- A block store is a struct full of function pointers
- Each FP is a "member function" whose first argument is "this"
- Also a pointer to some other struct containing the block store's state – private member variables

```
typedef struct block_store {
  void *state;
  int (*getninodes)(struct block_store *this_bs);
  int (*getsize)(struct block_store *this_bs,
    unsigned int ino);
  int (*setsize)(struct block_store *this_bs,
    unsigned int ino, block_no newsize);
  int (*read)(struct block_store *this_bs,
    unsigned int ino, block_no offset, block_t *block);
  int (*write)(struct block_store *this_bs,
    unsigned int ino, block_no offset, block_t *block);
  void (*release)(struct block_store *this_bs);
  int (*sync)(struct block_store *this_bs,
    unsigned int ino);
} block_store_t;

typedef block_store_t *block_if;
```

Adding "Objects" to C

- Each block store "class" can inherit this "interface" by providing functions matching the FP types
- Each can define its own state struct

```
int clockdisk_getninodes(block_store_t *this_bs);
int clockdisk_getsize(block_store_t *this_bs,
    unsigned int ino);
int clockdisk_setsize(block_store_t *this_bs,
    unsigned int ino, block_no newsize);
int clockdisk_read(block_store_t *this_bs,
    unsigned int ino, block_no offset, block_t *block);
int clockdisk_write(block_store_t *this_bs,
    unsigned int ino, block_no offset, block_t *block);
void clockdisk_release(block_store_t *this_bs);
int clockdisk_sync(block_store_t *this_bs,
    unsigned int ino);

struct clockdisk_state {
  block_if below;
  block_t* blocks;
  block_no nblocks;
};
```

Adding "Objects" to C

```
block_if clockdisk_init(block_if below,
    block_t *blocks, block_no nblocks) {
  struct clockdisk_state *cs = new_alloc(
    struct clockdisk_state);
  cs->below = below;
  cs->blocks = blocks;
  cs->nblocks = nblocks;
  block_if this_bs = new_alloc(block_store_t);
  this_bs->state = cs;
  this_bs->getninodes = clockdisk_getninodes;
  this_bs->getsize = clockdisk_getsize;
  this_bs->setsize = clockdisk_setsize;
  this_bs->read = clockdisk_read;
  this_bs->write = clockdisk_write;
  this_bs->release = clockdisk_release;
  this_bs->sync = clockdisk_sync;
  return this_bs;
}
```

Initialize this object's state

Assign the function pointers in `block_store_t` to this class's implementation of those functions

- Each block store class has a "constructor" that returns a `block_if`

How Do I Use This?

- Within a "member function," you can safely cast `this->state` to your own state struct:

```
static int clockdisk_read(block_if this_bs, unsigned int ino,
    block_no offset, block_t *block) {
  struct clockdisk_state* cs = this_bs->state;
```

- You can call a function on another block store through its `block_store_t` interface:

```
int r = (*cs->below->read)(cs->below, ino, offset, block);
```

Call the `read` function of `cs->below`

Pass `cs->below` as the `this` parameter

Block Store Functions

- `int` **`getninodes`**`(this)`: Returns the total number of inodes this block store supports – for the "disk" block store, this will be 1
- `int` **`getsize`**`(this, inode)`: Returns the number of blocks associated with the given inode
- `int` **`setsize`**`(this, inode, nblocks)`: Resizes an inode to include a specified number of blocks; returns the old size. Note that this can implicitly free blocks if the new size is smaller.

Block Store Functions

- `int` **`read`**`(this, inode, offset, *block)`: Reads a single block at the given offset (in blocks) within an inode, returns it in `*block`. Returns -1 on error, 0 on success.
- `int` **`write`**`(this, inode, offset, *block)`: Writes the data in `*block` to the specified inode and offset (in blocks). Returns -1 on error, 0 on success.
- `int` **`sync`**`(this, inode)`: Syncs all data within the specified inode to the underlying layer, if this block store is a cache.
- `void` **`release`**`(this)`: Frees the block store, inverse of "init"

Your Task for Project 3

- Implement a write-through block cache in wtclockdisk.c
  - wtclockdisk_read, wtclockdisk_write, wtclockdisk_setsize
- Implement a write-back block cache in clockdisk.c
  - clockdisk_read, clockdisk_write, clockdisk_setsize, clockdisk_sync
- In both files, you'll want to implement this helper function:
  ```
  static void cache_update(struct [wt]clockdisk_state *cs,
  unsigned int ino, block_no offset, block_t *block [, bool
  dirty])
  ```
  - Call when you just had a cache miss
  - Use CLOCK to choose a block to evict, put `*block` in the cache
  - Only clockdisk keeps track of whether blocks are dirty

Keeping Track of Statistics

- `struct clockdisk_state` and `struct wtclockdisk_state` already have some members defined for you, including:
  - `read_hit`, `read_miss`: Number of cache hits vs. cache misses for read operations
  - `write_hit`, `write_miss`: Number of cache hits vs. cache misses for write operations
- Update these every time your cache handles a read or write