ICS143A: Principles of Operating Systems

Lecture 18: File systems

Anton Burtsev

November, 2017

The role of file systems

The role of file systems

- Sharing
  - Sharing of data across users and applications
- Persistence
  - Data is available after reboot

Architecture

- On-disk and in-memory data structures represent
  - The tree of named files and directories
  - Record identities of disk blocks which hold data for each file
  - Record which areas of the disk are free

Crash recovery

- File systems must support crash recovery
  - A power loss may interrupt a sequence of updates
  - Leave file system in inconsistent state
    - E.g. a block both marked free and used

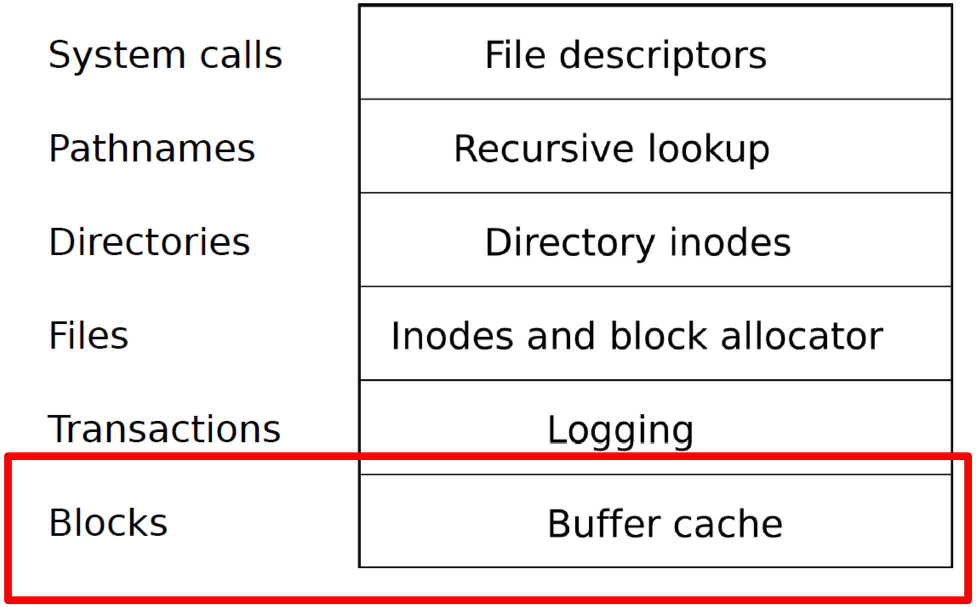
Multiple users

- Multiple users operate on a file system concurrently
  - File system must maintain invariants

Speed

- Access to a block device is several orders of magnitude slower
  - Memory: 200 cycles
  - Disk: 20 000 000 cycles
- A file system must maintain a cache of disk blocks in memory

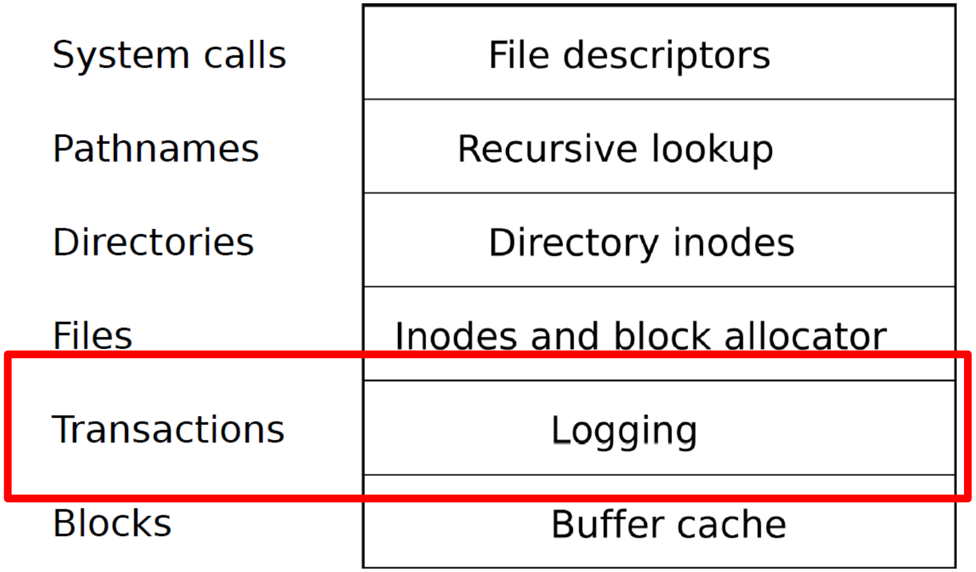
Block layer



System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Read and write data
  - From a block device
  - Into a buffer cache
- Synchronize across multiple readers and writers

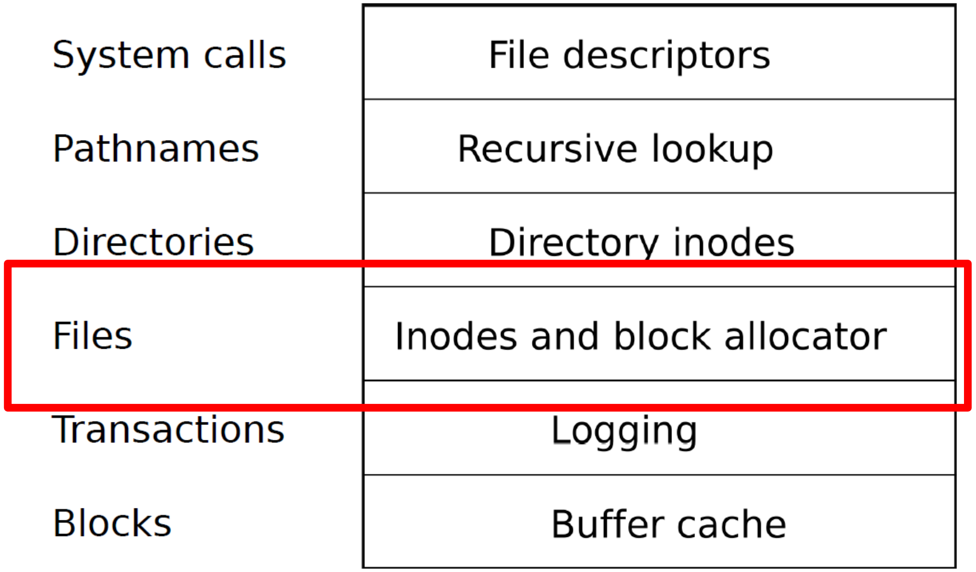
Transactions



System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Group multiple writes into an atomic transaction

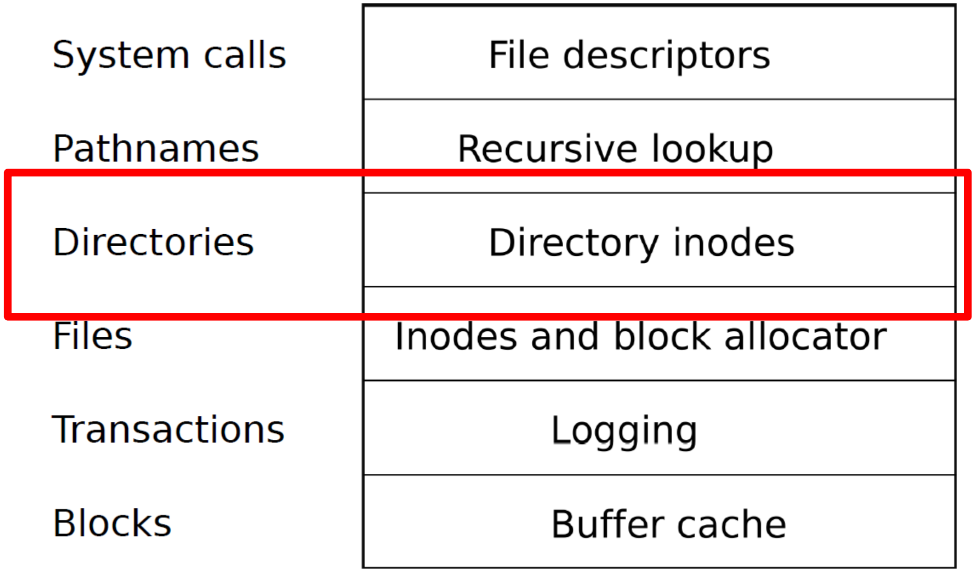
Files



System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Unnamed files
  - Represented as inodes
  - Sequence of blocks holding file's data

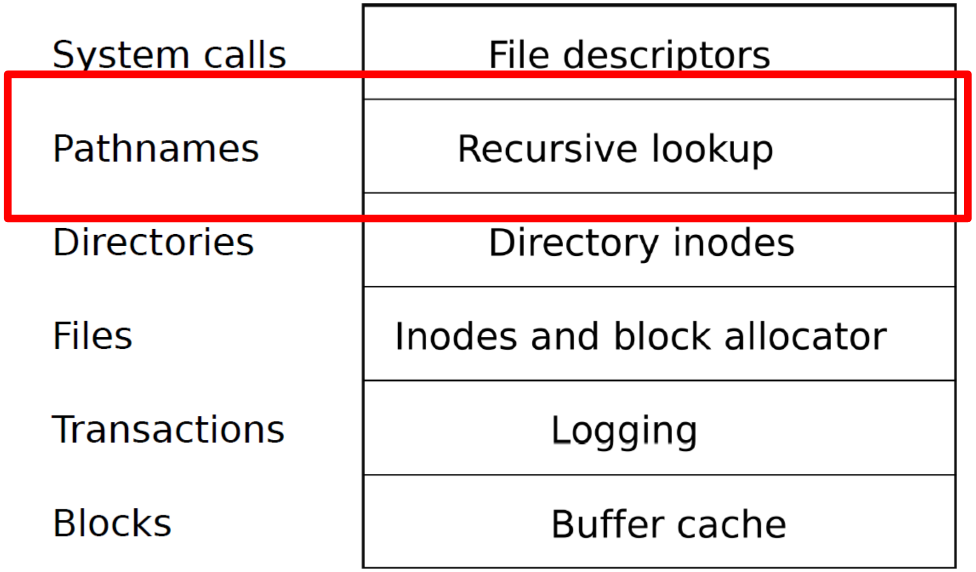
Directories



System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Special kind of inode
  - Sequence of directory entries
  - Each contains name and a pointer to an unnamed inode

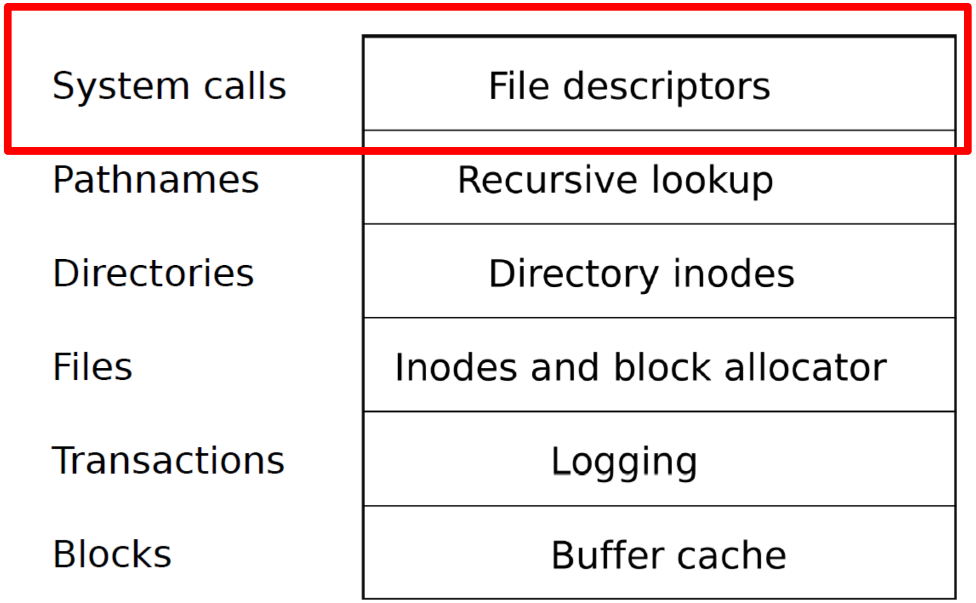
Pathnames



System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Hierarchical path names
  - /usr/bin/sh
  - Recursive lookup

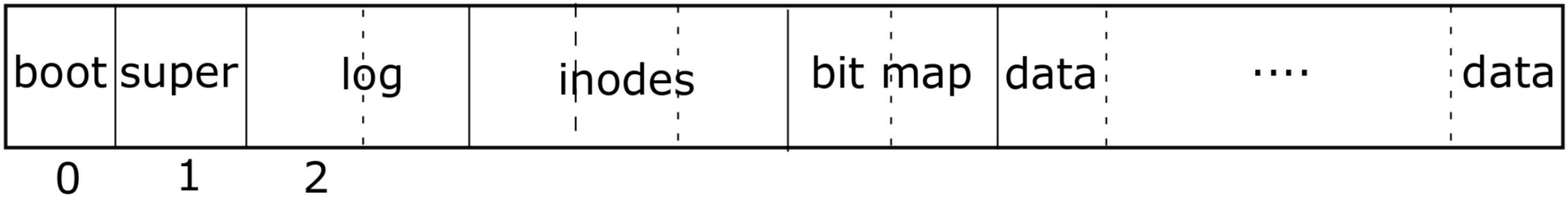
System call



System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

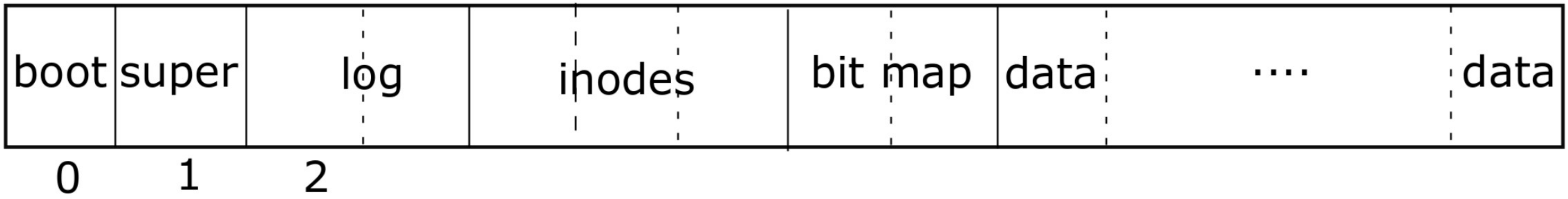
- Abstract UNIX resources as files
  - Files, sockets, devices, pipes, etc.
- Unified programming interface

File system layout on disk



- Block #0: Boot code
- Block #1: (superblock) Metadata about the file system
  - Size (number of blocks)
  - Number of data blocks
  - Number of inodes
  - Number of blocks in log

File system layout on disk

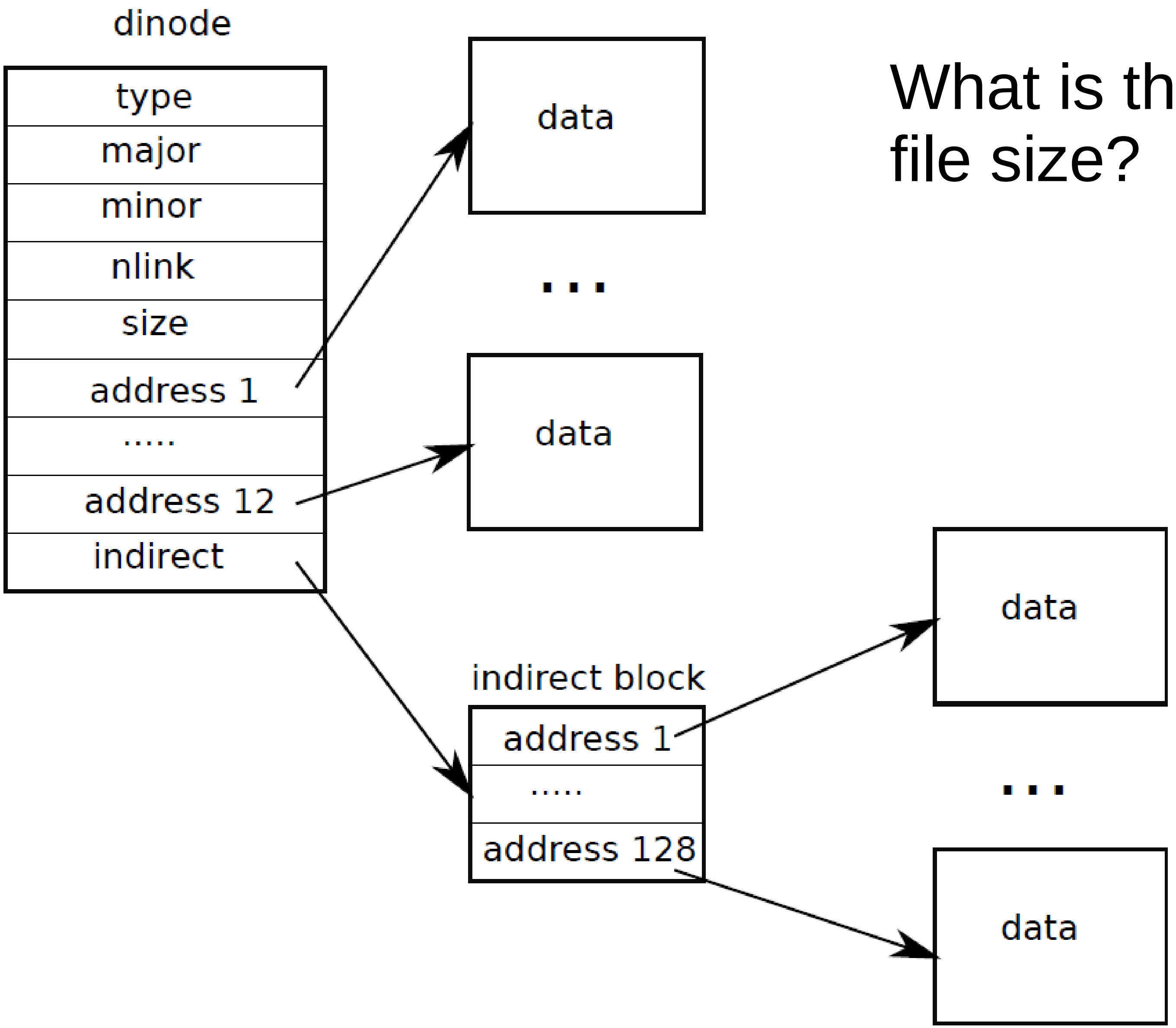


- Block #2: Log area: maintaining consistency in case of a power outage or system crash

thanksgiving

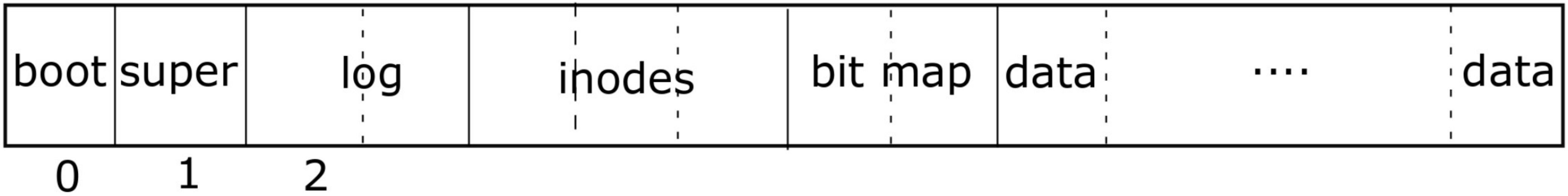
- Inode area
  - Unnamed files

Representing files on disk



What is the max file size?

File system layout on disk



- Block #2: Log area: maintaining consistency in case of a power outage or system crash

thanksgiving

- Inode area
  - Unnamed files
- Bit map area: track which blocks are in use
- Data area: actual file data

Buffer cache layer

Buffer cache layer

- Two goals:
  - Synchronization:
    - Only one copy of a data block exist in the kernel
    - Only one writer updates this copy at a time
  - Caching
    - Frequently used copies are cached for efficient reads and writes

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

- Flags

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

Buffer cache: linked list

Buffer cache: linked list

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

- Device
  - We might have multiple disks

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

- Block number on disk

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

Buffer cache: linked list

Buffer cache: linked list

- LRU list

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

Buffer cache: linked list

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

- Cached data
  - 512 bytes

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

Buffer cache layer: interface

- bread() and bwrite() - obtain a copy for reading or writing
  - Owned until brelse()
  - Locking with a flag (B_BUSY)
- Other threads will be blocked and wait until brelse()

Block read and write operations

```
struct buf*
bread(uint dev, uint sector)
{
  struct buf *b;

  b = bget(dev, sector);
  if(!(b->flags & B_VALID)) {
    iderw(b);
  }
  return b;
}

bwrite(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

Getting a block from a buffer cache (part 1)

```
bget(uint dev, uint sector)
{
  struct buf *b;
  acquire(&bcache.lock);
loop:
  // Is the sector already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->sector == sector){
      if(!(b->flags & B_BUSY)){
        b->flags |= B_BUSY;
        release(&bcache.lock);
        return b;
      }
      sleep(b, &bcache.lock);
      goto loop;
    }
  }

...
}
```

Getting a block from a buffer cache (part 2)

```
bget(uint dev, uint sector)
{
  struct buf *b;
  acquire(&bcache.lock);
loop:
...

// Not cached; recycle some non-busy and clean buffer.
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
      b->dev = dev;
      b->sector = sector;
      b->flags = B_BUSY;
      release(&bcache.lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

Block read and write operations

```
struct buf*
bread(uint dev, uint sector)
{
  struct buf *b;

  b = bget(dev, sector);
  if(!(b->flags & B_VALID)) {
    iderw(b);
  }
  return b;
}
```

```
bwrite(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

Release buffer

```
// Release a B_BUSY buffer.
// Move to the head of the MRU list.
void
brelse(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("brelse");

  acquire(&bcache.lock);

  b->next->prev = b->prev;
  b->prev->next = b->next;
  b->next = bcache.head.next;
  b->prev = &bcache.head;
  bcache.head.next->prev = b;
  bcache.head.next = b;

  b->flags &= ~B_BUSY;
  wakeup(b);

  release(&bcache.lock);
}
```

- Maintain least recently used list
  - Move to the head

Common pattern

```
bread()
bwrite()
brelse()
```

- Read
- Write
- Release

```c
// Copy committed blocks from log to their home location
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
    bwrite(dbuf);  // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

Example

Logging layer

Logging layer

- Consistency
  - File system operations involve multiple writes to disk
  - During the crash, subset of writes might leave the file system in an inconsistent state
  - E.g. file delete can crash leaving:
    - Directory entry pointing to a free inode
    - Allocated but unlinked inode

Logging

- Writes don't directly go to disk
  - Instead they are logged in a journal
  - Once all writes are logged, the system writes a special commit record
    - Indicating that log contains a complete operation
- At this point file system copies writes to the on-disk data structures
  - After copy completes, log record is erased

Recovery

- After reboot, copy the log
  - For operations marked as complete
    - Copy blocks to disk
  - For operations partially complete
    - Discard all writes
    - Information might be lost (output consistency, e.g. can launch the rocket twice)

Typical use of transactions

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

Log (in memory)

```
struct logheader {
  int n;
  int block[LOGSIZE];
};

struct log {
  struct spinlock lock;
  int start;
  int size;
  int outstanding; // how many FS sys calls are
                      executing.
  int committing; // in commit(), please wait.
  int dev;
  struct logheader lh;
};
```

Typical use of transactions

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

begin_op()

```
// called at the start of each FS system call.
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

- Case #1
  - Log is being committed
  - Sleep

begin_op()

```
// called at the start of each FS system call.
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

- Case #2
  - Log doesn't have enough space for the new transaction

begin_op()

```
// called at the start of each FS system call.
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

- Case #3
  - All ok, reserve space in the log for the new transaction

Typical use of transactions

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

- `log_write()` replaces `bwrite(); brelse()`

log_write

```
log_write(struct buf *b)
{
  int i;

  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (log.outstanding < 1)
    panic("log_write outside of trans");

  acquire(&log.lock);
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno) // log absorbtion
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // prevent eviction
  release(&log.lock);
}
```

- Check if already in log

log_write

```
log_write(struct buf *b)
{
  int i;

  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (log.outstanding < 1)
    panic("log_write outside of trans");

  acquire(&log.lock);
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno) // log absorbtion
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // prevent eviction
  release(&log.lock);
}
```

- Add to the log
- Prevent eviction

Typical use of transactions

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

end_op()

```
end_op(void)
{
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
  } else {
    // begin_op() may be waiting for log space.
    wakeup(&log);
  }
  release(&log.lock);

  if(do_commit){
    // call commit w/o holding locks, since not allowed
    // to sleep with locks.
    commit();
    acquire(&log.lock);
    log.committing = 0;
    wakeup(&log);
    release(&log.lock);
  }
}
```

end_op()

```
end_op(void)
{
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
  } else {
    // begin_op() may be waiting for log space.
    wakeup(&log);
  }
  release(&log.lock);

  if(do_commit){
    // call commit w/o holding locks, since not allowed
    // to sleep with locks.
    commit();
    acquire(&log.lock);
    log.committing = 0;
    wakeup(&log);
    release(&log.lock);
  }
}
```

commit()

```
commit()
{
  if (log.lh.n > 0) {
    write_log(); // Write modified blocks
                     from cache to log
    write_head(); // Write header to disk --
                     the real commit
    install_trans(); // Now install writes
                        to home locations
    log.lh.n = 0;
    write_head(); // Erase the transaction
                     from the log
  }
}
```

write_log()

```
// Copy modified blocks from cache to log.
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev,
                      log.start+tail+1); // log block
    struct buf *from = bread(log.dev,
                      log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to); // write the log
    brelse(from);
    brelse(to);
  }
}
```

- Loop through the entire log

write_log()

```
// Copy modified blocks from cache to log.
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev,
                      log.start+tail+1); // log block
    struct buf *from = bread(log.dev,
                      log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to); // write the log
    brelse(from);
    brelse(to);
  }
}
```

- Read the log block
  - Log goes to `log.start+tail+1`

write_log()

```
// Copy modified blocks from cache to log.
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev,
                      log.start+tail+1); // log block
    struct buf *from = bread(log.dev,
                      log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to); // write the log
    brelse(from);
    brelse(to);
  }
}
```

- Read the actual block
  - It's in the buffer cache
  - Block number is in `log.lh.block[tail]`

write_log()

```
// Copy modified blocks from cache to log.
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev,
                      log.start+tail+1); // log block
    struct buf *from = bread(log.dev,
                      log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to); // write the log
    brelse(from);
    brelse(to);
  }
}
```

- Copy block data into the log

write_log()

```
// Copy modified blocks from cache to log.
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev,
                      log.start+tail+1); // log block
    struct buf *from = bread(log.dev,
                      log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to); // write the log
    brelse(from);
    brelse(to);
  }
}
```

- Write the log block (to)
- Release both blocks

commit()

```
commit()
{
  if (log.lh.n > 0) {
    write_log(); // Write modified blocks
                     from cache to log
    write_head(); // Write header to disk --
                     the real commit
    install_trans(); // Now install writes
                        to home locations
    log.lh.n = 0;
    write_head(); // Erase the transaction
                     from the log
  }
}
```

write_head()

```
// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *)
                                     (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}
```

- Read the log header block
  - It's in `log.start`

write_head()

```
// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *)
                                     (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}
```

- Interpret `buf->data` as log header
    - See how type casts work in C

write_head()

```
// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *)
                                     (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}
```

- Write log size (`log.lh.n`) into block of the logheader

write_head()

```
// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *)
                                     (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}
```

- Write the entire log (numbers of blocks in the log) into log header

write_head()

```
// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *)
                                     (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}
```

- Write block to disk
- Release

commit()

```
commit()
{
  if (log.lh.n > 0) {
    write_log(); // Write modified blocks
                     from cache to log
    write_head(); // Write header to disk --
                     the real commit
    install_trans(); // Now install writes
                        to home locations
    log.lh.n = 0;
    write_head(); // Erase the transaction
                     from the log
  }
}
```

install_trans()

```
// Copy committed blocks from log to their home location
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev,
                   log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev,
                   log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE); // copy block
                                        //  to dst
    bwrite(dbuf); // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

- Read the block from the log area (`log.start+tail+1`)

install_trans()

```
// Copy committed blocks from log to their home location
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev,
                   log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev,
                   log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE); // copy block
                                        //  to dst
    bwrite(dbuf); // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

- Read the block where data should go on disk
  - It's a block number in `log.lh.block[tail]`

install_trans()

```
// Copy committed blocks from log to their home location
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev,
                   log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev,
                   log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE); // copy block
                                        //  to dst
    bwrite(dbuf); // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

- Copy data

install_trans()

```
// Copy committed blocks from log to their home location
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev,
                   log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev,
                   log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE); // copy block
                                        //  to dst
    bwrite(dbuf); // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

- Write the block to disk
- Release both blocks

commit()

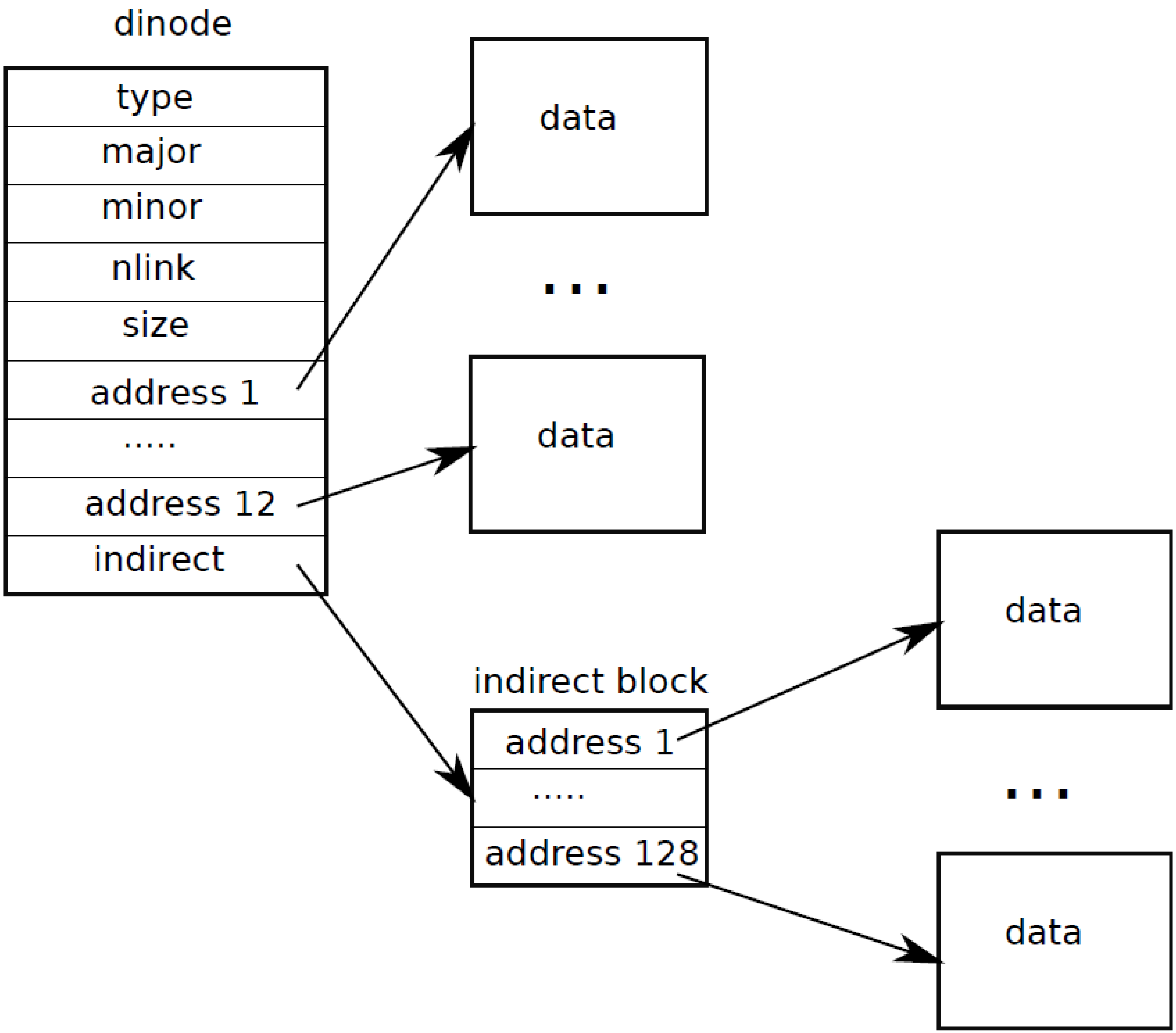
```
commit()
{
  if (log.lh.n > 0) {
    write_log(); // Write modified blocks
                     from cache to log
    write_head(); // Write header to disk --
                     the real commit
    install_trans(); // Now install writes
                        to home locations
    log.lh.n = 0;
    write_head(); // Erase the transaction
                     from the log
  }
}
```

Inode layer

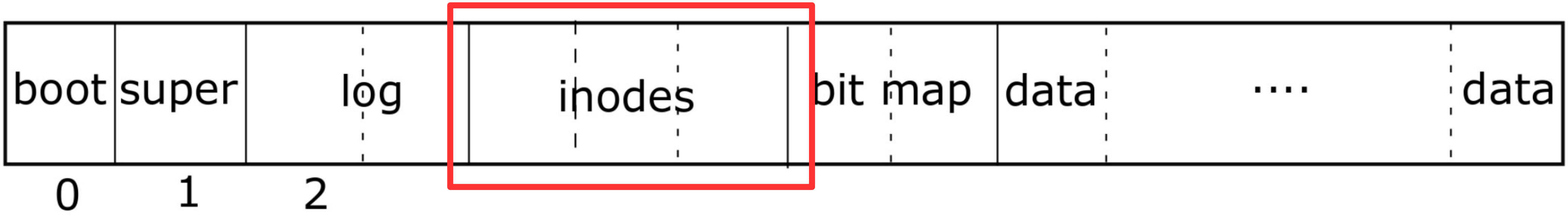
Inode

- Describes a single unnamed file
- The inode on disk holds metadata
    - File type, size, # of links referring to it, list of blocks with data
- In memory
    - A copy of an on-disk inode + some additional kernel information
        - Reference counter (ip->ref)
        - Synchronization flags (ip->flags)

Representing files on disk



File system layout on disk

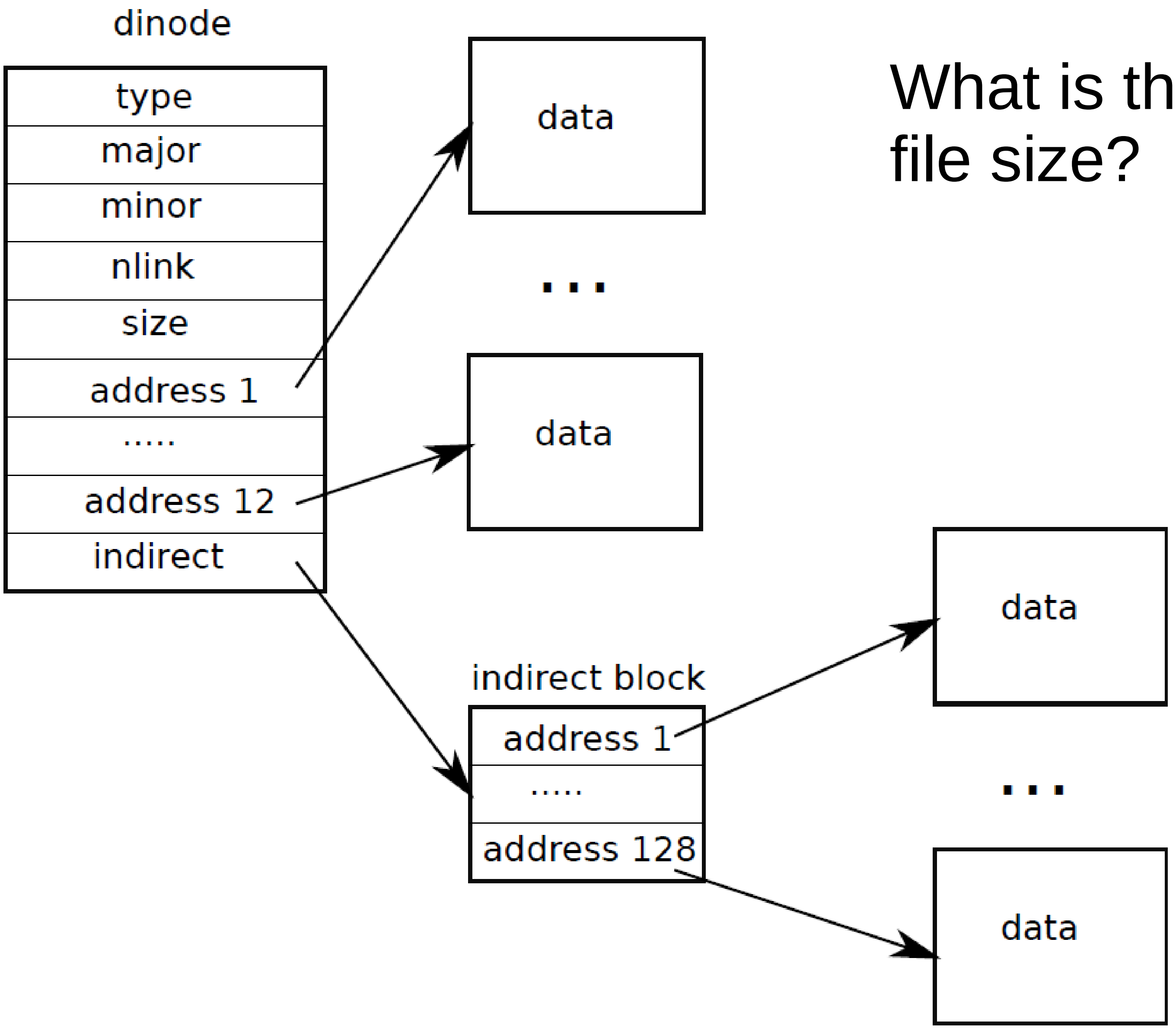


- Inodes are stored as an array on disk
  - sb.startinode
- Each inode has a number (indicating its position on disk)
- The kernel keeps a cache of inodes in memory
  - Synchronization

Inode on disk

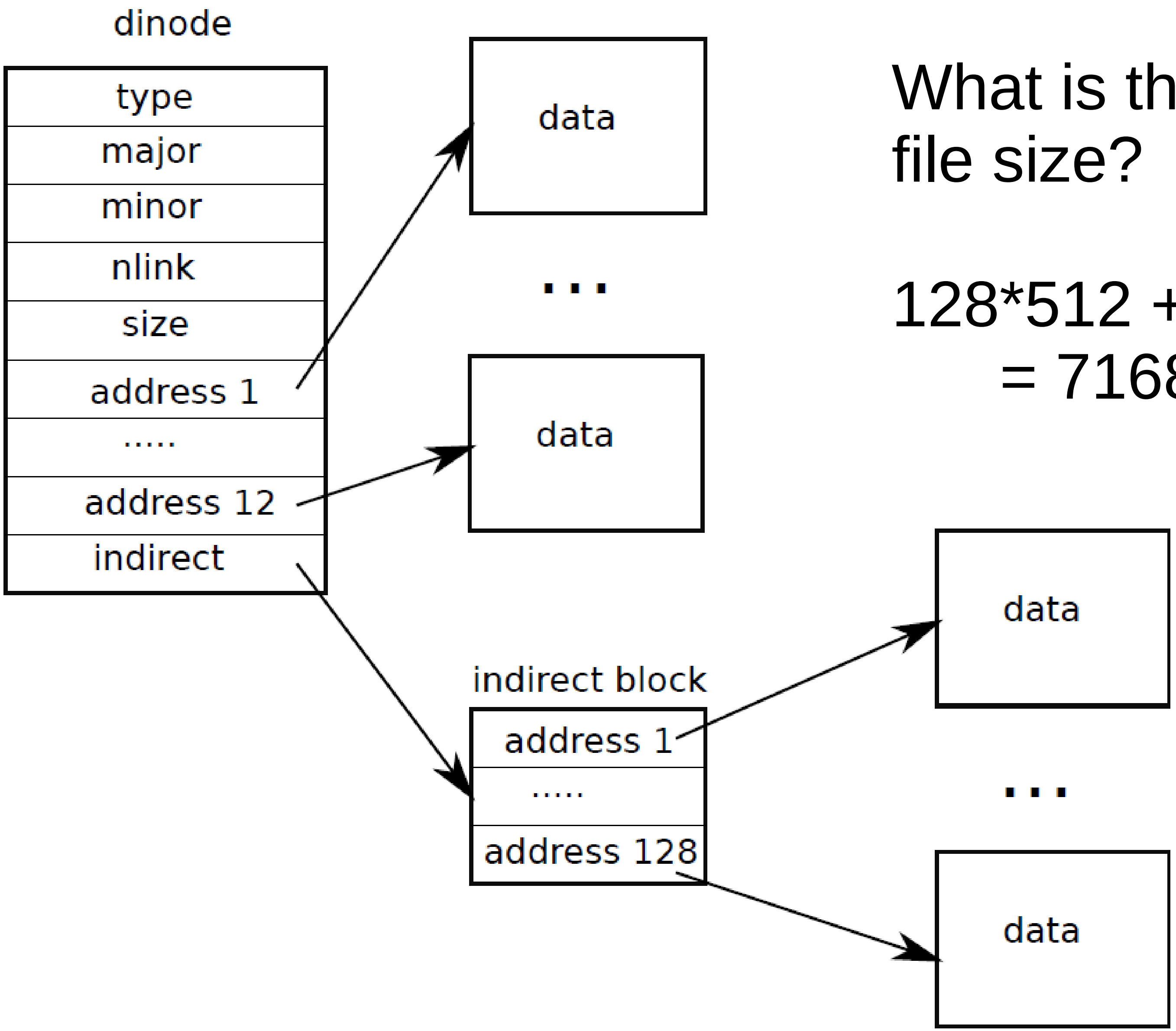
```
// On-disk inode structure
struct dinode {
  short type; // File type
  short major; // Major device number (T_DEV only)
  short minor; // Minor device number (T_DEV only)
  short nlink; // Number of links to inode in file system
  uint size; // Size of file (bytes)
  uint addrs[NDIRECT+1]; // Data block addresses
};
```

Representing files on disk



What is the max file size?

Representing files on disk



What is the max file size?

128*512 + 12*512
= 71680

Inode in memory

```
// in-memory copy of an inode
struct inode {
  uint dev; // Device number
  uint inum; // Inode number
  int ref; // Reference count
  int flags; // I_BUSY, I_VALID

  short type; // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

In-memory cache of inodes

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

Lifecycle of inode

- Allocation (on disk)
  - ialloc()
  - iput() -- deallocates
- Referencing in cache
  - ip->ref tracks the number of active pointers to an inode in memory
  - iget()/iput()

Accessing inodes

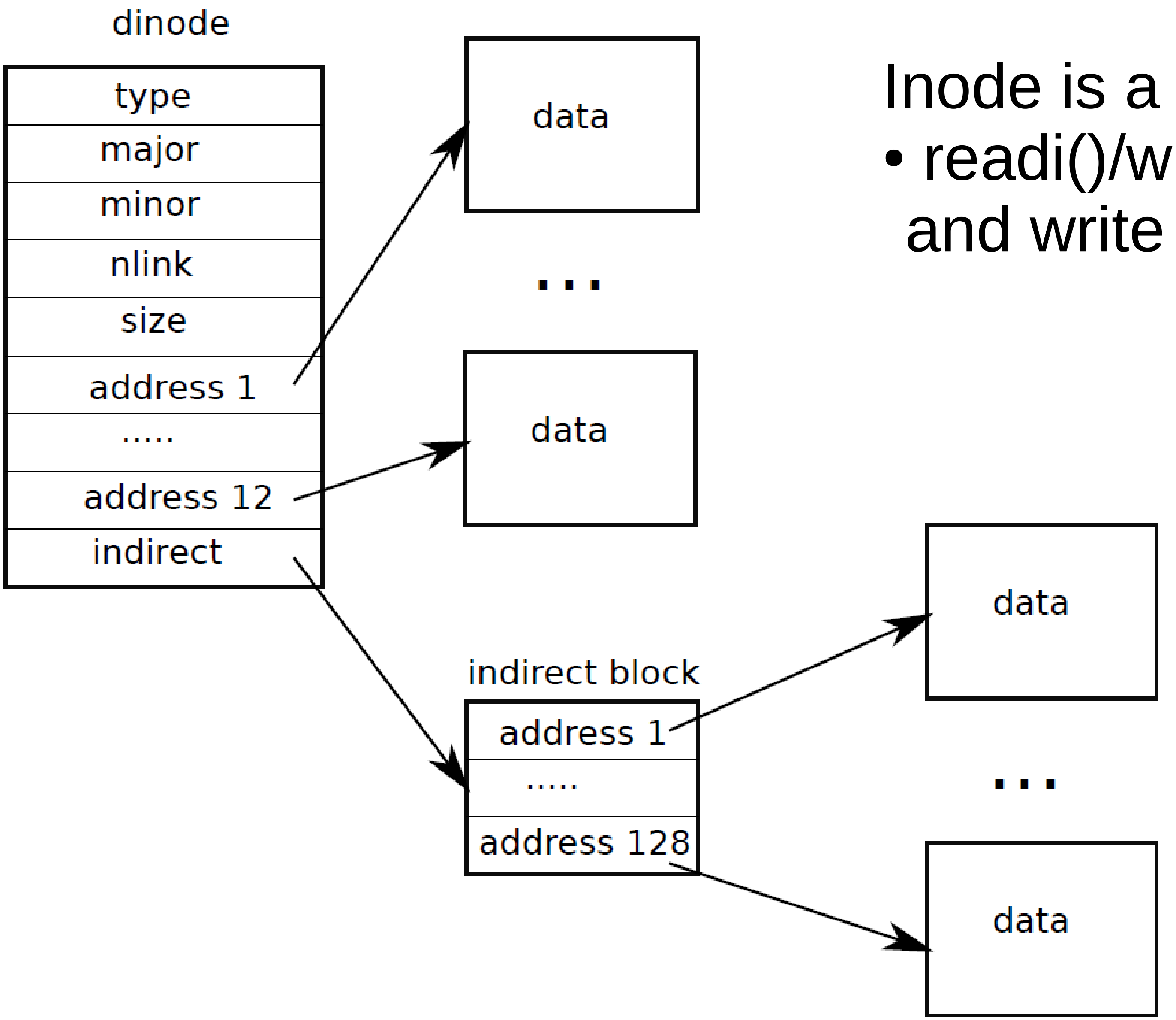
```
// Thus a typical sequence is:
// ip = iget(dev, inum)
// ilock(ip)
// ... examine and modify ip->xxx ...
// iunlock(ip)
// iput(ip)
```

iget()

```
iget(uint dev, uint inum) {
...
  acquire(&icache.lock);
  // Is the inode already cached?
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      ip->ref++;
      release(&icache.lock);
      return ip;
    }
    if(empty == 0 && ip->ref == 0) // Remember empty slot.
    empty = ip;
  }
...

  ip->ref = 1;
...
  release(&icache.lock);
  return ip;
}
```

Reading and writing inodes



Inode is a file

- readi()/writei() read and write it

Example: sys_read()

```c
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```

- Question:
  - Where does f come from?

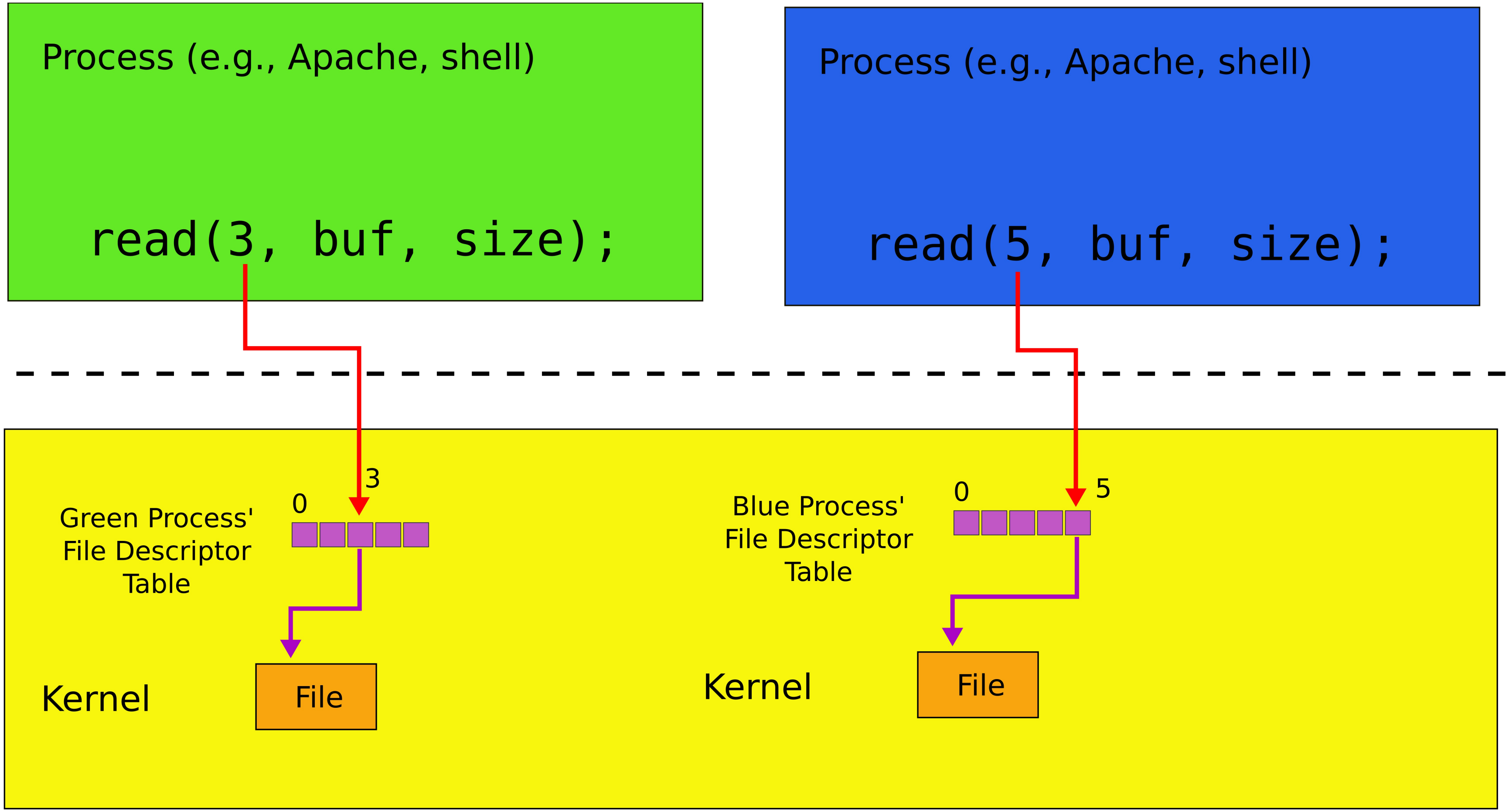
argfd()

```
// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
static int
argfd(int n, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;

  if(argint(n, &fd) < 0)
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
    return -1;
  if(pfd)
    *pfd = fd;
  if(pf)
    *pf = f;
  return 0;
}
```

- Remember file descriptors?
  - Each process has a table
  - proc->ofile[]

File descriptors: two processes


Process (e.g., Apache, shell)
read(3, buf, size);
Process (e.g., Apache, shell)
read(5, buf, size);
3
0
Green Process' File Descriptor Table
Kernel
File
0
5
Blue Process' File Descriptor Table
Kernel
File

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

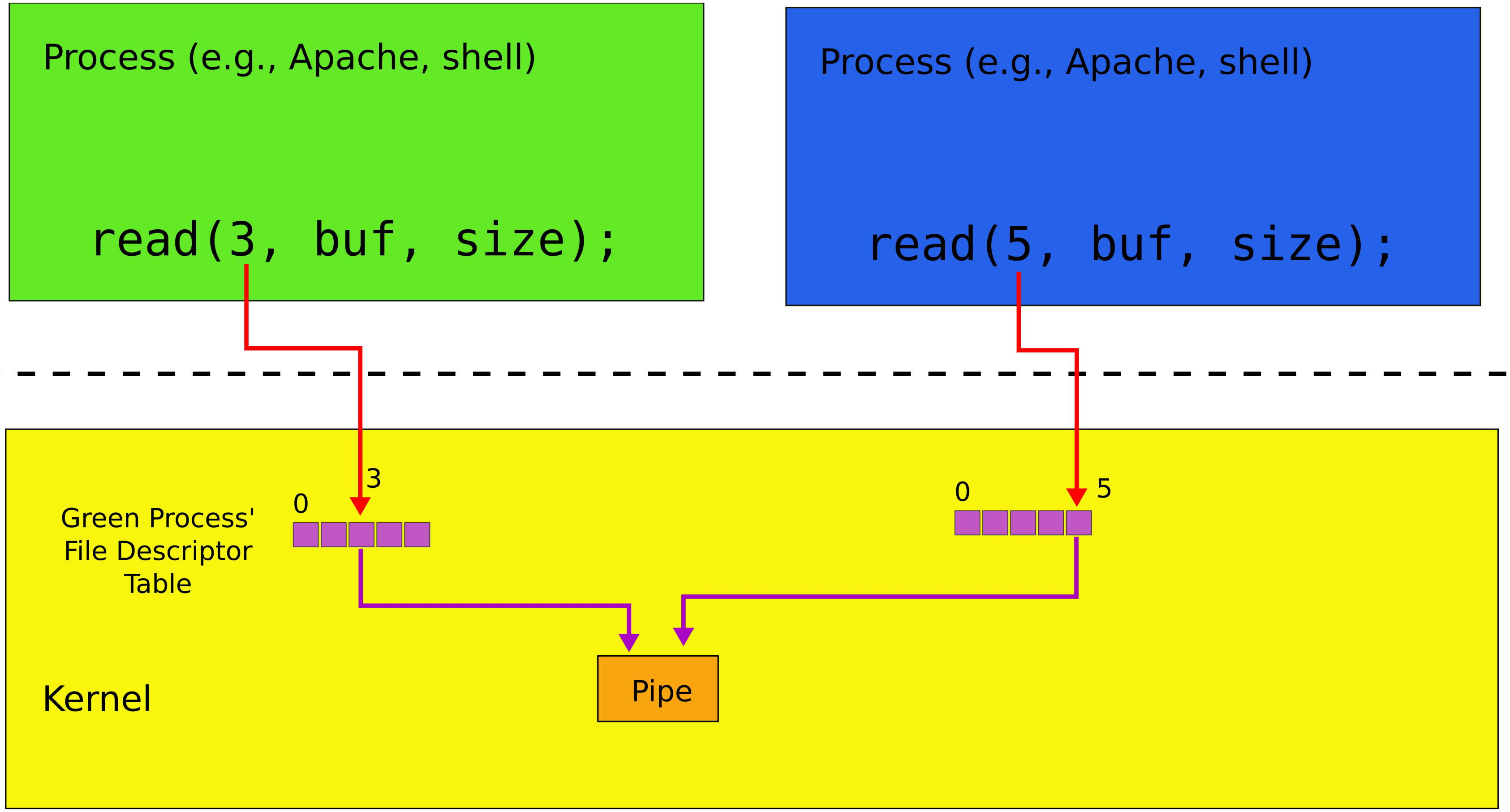
- struct proc has an array of struct file pointers
  - Each element is a “file descriptor”

struct file

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

- A file can be a pipe or an inode
  - It can be readable and/or writable
  - Each file has current offset (off)

Two file descriptors pointing to a pipe



readi()

```c
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

readi()

```c
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

readi()

```
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

readi()

```
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

- Note
  - Read starts with the current offset (`f->off`)

readi()

```
readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

...
  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

readi()

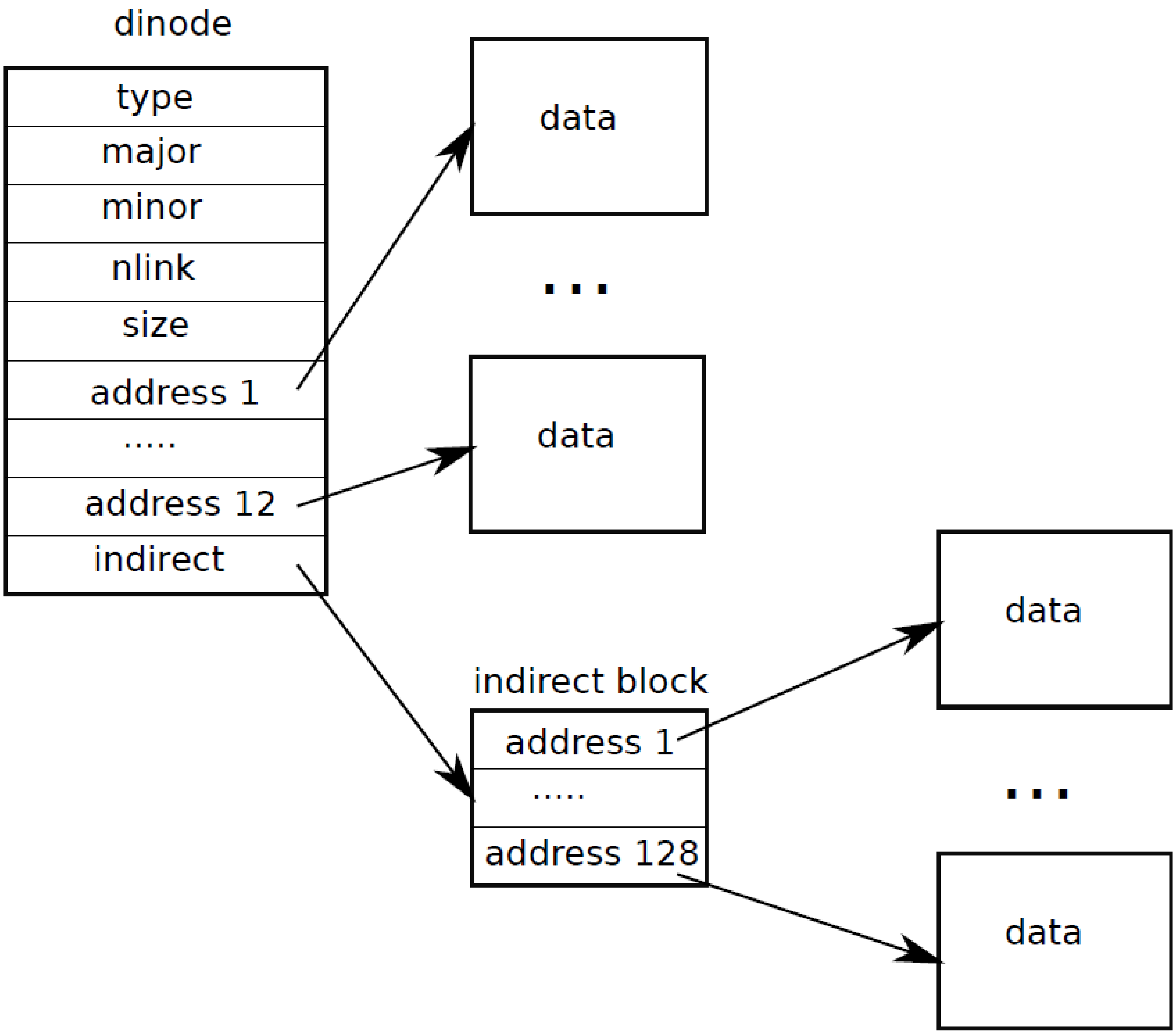
```
readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

...
  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

- What is this bmap() function?

Representing files on disk



bmap()

```
static uint
bmap(struct inode *ip, uint bn)
{
...
  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
...
}
```

- Each inode has some number (NDIRECT) of direct pointers

bmap()

```
static uint
bmap(struct inode *ip, uint bn)
{
...
  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
...
}
```

- No it's beyond NDIRECT

bmap()

```
static uint
bmap(struct inode *ip, uint bn)
{
...
  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
...
}
```

- Read an indirect block

bmap()

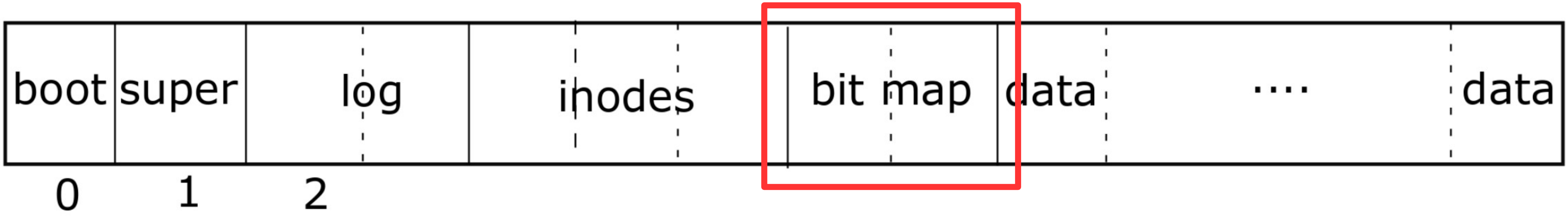
```
static uint
bmap(struct inode *ip, uint bn)
{
...
  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
...
}
```

- Check if a pointer in the indirect block is already allocated

Block allocator

Block allocator



- Bitmap of free blocks
  - balloc()/bfree()
- Read the bitmap block by block
  - Scan for a “free” bit
- Access to the bitmap is synchronized with bread()/bwrite()/brelse() operations

balloc()

```c
// Allocate a zeroed disk block.
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){ // Is block free?
        bp->data[bi/8] |= m; // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}
```

balloc()

```
// Allocate a zeroed disk block.
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){ // Is block free?
        bp->data[bi/8] |= m; // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}
```

balloc()

```
// Allocate a zeroed disk block.
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){ // Is block free?
        bp->data[bi/8] |= m; // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}
```

balloc()

```
// Allocate a zeroed disk block.
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){ // Is block free?
        bp->data[bi/8] |= m; // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}
```

- Why do we need `log_write()` instead of `bwrite()`?

Directory layer

Directory inodes

- A directory inode is a sequence of directory entries and inode numbers
  - Each name is max of 14 characters
  - Has a special inode type T_DIR
- dirlookup() - searches for a directory with a given name
- dirlink() - adds new file to a directory

Directory entry

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

dirlookup()

```
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
...
  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
        inum = de.inum;
        return iget(dp->dev, inum);
      }
  }

  return 0;
}
```

Path names layer

- Series of directory lookups to resolve a path
  - E.g. /usr/bin/sh
- Namei() - resolves a path into an inode
  - If path starts with “/” evaluation starts at the root
  - Otherwise current directory

namei()

```
struct inode*
namei(char *path)
{
  char name[DIRSIZ];
  return namex(path, 0, name);
}
```

namex()

```
namex(char *path, int nameiparent, char *name)
{
...
  if(*path == ’/’)
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(proc->cwd);

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
...
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}
```

```
namex(char *path, int nameiparent, char *name)
{
...
  if(*path == ’/’)
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(proc->cwd);
  // skipelem("a/bb/c", name) = "bb/c", setting name = "a"
  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
...
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}
```

namex()

Eaxmple: sys_open

```
sys_open(void)
{
...
  if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
    return -1;

  begin_op();

...
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
...
  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

  f->type = FD_INODE;
  f->ip = ip;
...
  return fd;
}
```

File descriptor layer

File descriptors

- Uniform access to
  - Files
  - Devices, e.g., console
  - Pipes

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

Eaxmple: sys_open

```
sys_open(void)
{
...
  if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
    return -1;

  begin_op();

...
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
...
  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

  f->type = FD_INODE;
  f->ip = ip;
...
  return fd;
}
```

Files and filealloc()

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
...
struct file*
filealloc(void)
{
  struct file *f;

  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    if(f->ref == 0){
      f->ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}
```

File descriptors and fdalloc()

```
// Allocate a file descriptor for the given file.
// Takes over file reference from caller on success.
static int
fdalloc(struct file *f)
{
  int fd;

  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd] == 0){
      proc->ofile[fd] = f;
      return fd;
    }
  }
  return -1;
}
```

Thank you!

ialloc()

```
struct inode*
ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;

  for(inum = 1; inum < sb.ninodes; inum++) {
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){ // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      log_write(bp); // mark it allocated on the disk
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

bmap()

```
bmap(struct inode *ip, uint bn)
...
  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
    ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  panic("bmap: out of range");
}
```

Example: write system call

Write() syscall

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0
  || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

```
filewrite(struct file *f, char *addr, int n)
{
  if(f->type == FD_INODE){
...
    int i = 0;
    while(i < n){
...

      begin_trans();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      commit_trans();
  }
}
```

Write several blocks at a time

dirlookup()

```
static struct inode*
create(char *path, short type, short major, short minor)
{
...
  if((ip = dirlookup(dp, name, &off)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(type == T_FILE && ip->type == T_FILE)
      return ip;
    iunlockput(ip);
    return 0;
  }

  if((ip = ialloc(dp->dev, type)) == 0)
    panic("create: ialloc");

...
  if(type == T_DIR){ // Create . and .. entries.
    dp->nlink++; // for ".."
    iupdate(dp);
    // No ip->nlink++ for ".": avoid cyclic ref count.
    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
      panic("create dots");
  }

  if(dirlink(dp, name, ip->inum) < 0)
    panic("create: dirlink");
...
  return ip;
}
```