

CS248P: Operating Systems

Lecture 12: File systems

Anton Burtsev
November, 2018

The role of file systems

The role of file systems

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

Architecture

- On-disk and in-memory data structures that 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
 - And leave the file system in an inconsistent state
 - E.g., a block both marked free and used

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

- Group multiple writes into an atomic transaction

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

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 |

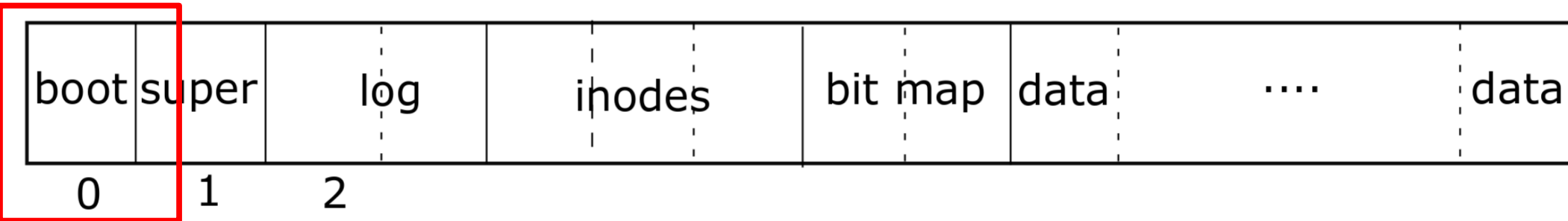
- Hierarchical path names
 - /usr/bin/sh
 - Recursive lookup

System call

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

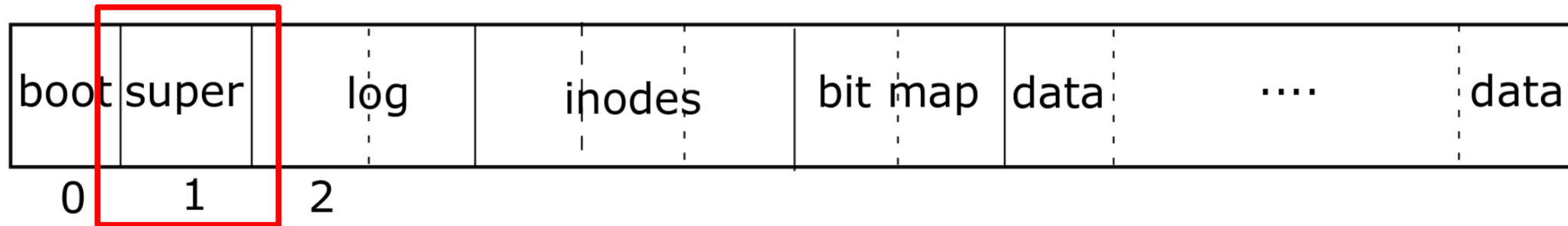
- Abstract UNIX resources as files
 - Files, sockets, devices, pipes, etc.
- Unified programming interface

File system layout on disk



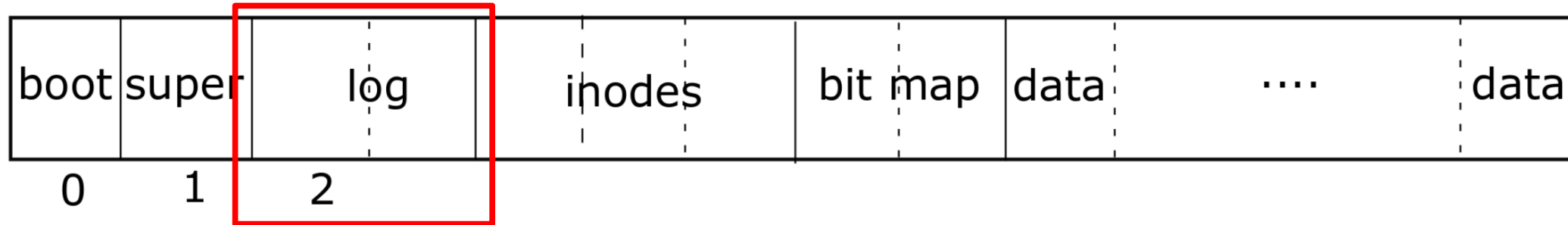
- Block #0: Boot code

File system layout on disk



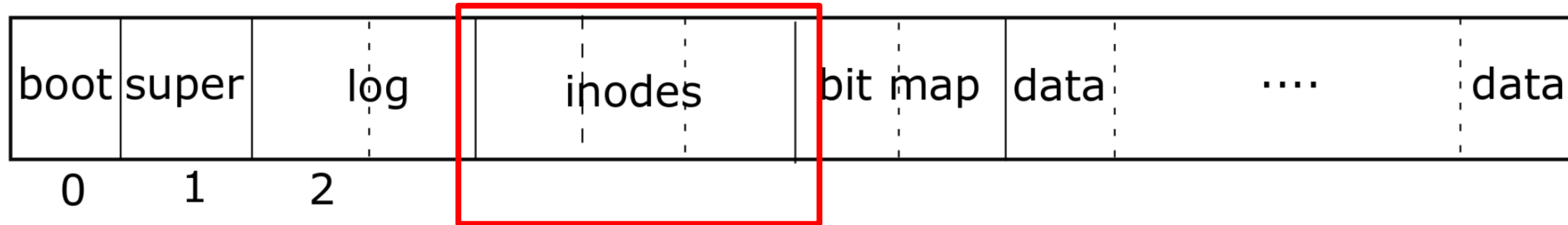
- Block #0: Boot code
- Block #1: (superblock) Metadata about the file system
 - Size (number of blocks)
 - Number of data blocks
 - Number of inodes
 - Number of blocks in log

File system layout on disk



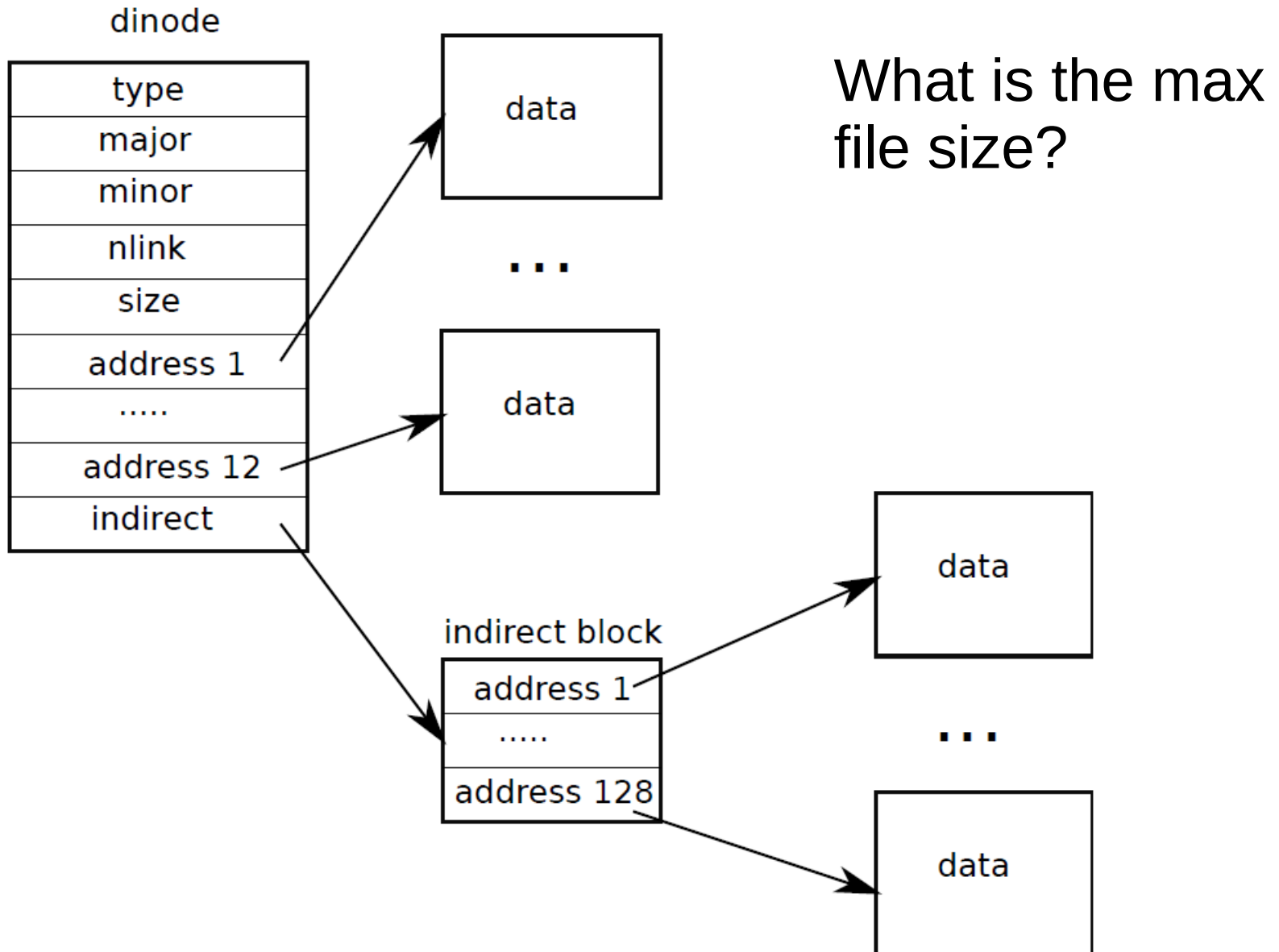
- Block #2: Log area: maintaining consistency in case of a power outage or system crash

File system layout on disk

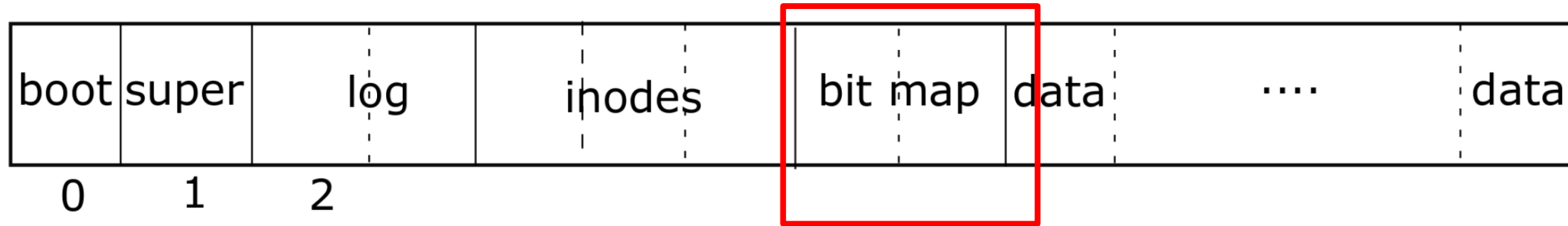


- Inode area
 - Unnamed files

Representing files on disk

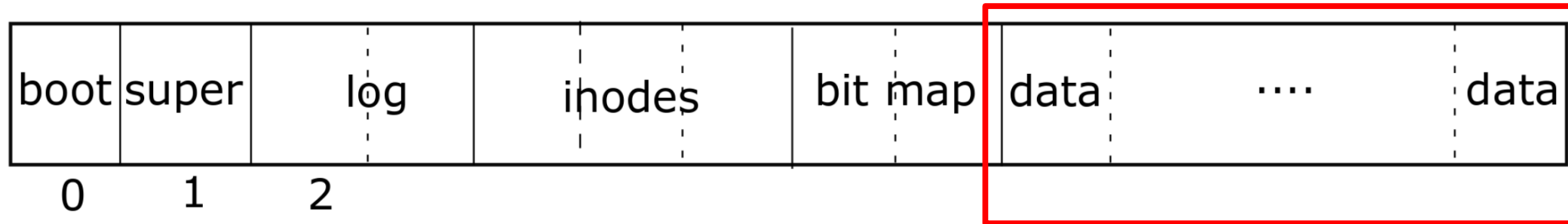


File system layout on disk



- Bit map area: track which blocks are in use

File system layout on disk



- 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

Buffer cache

```
3750 struct buf {
3751     int flags;
3752     uint dev;
3753     uint blockno;
3754     struct buf *prev; // LRU cache list
3755     struct buf *next;
3756     struct buf *qnext; // disk queue
3757     uchar data[BSIZE];
3758 };
3759 #define B_BUSY 0x1 // buffer is locked by some process
3760 #define B_VALID 0x2 // buffer has been read from disk
3761 #define B_DIRTY 0x4 // buffer needs to be written to disk
```

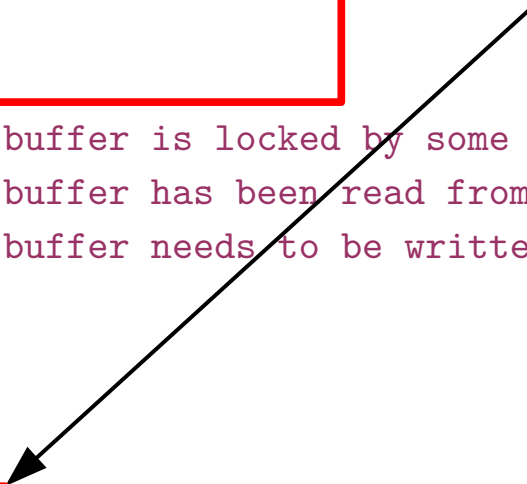
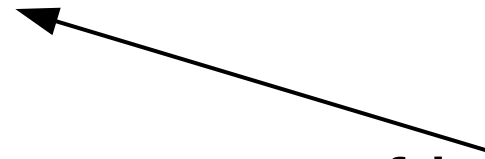
```
4329 struct {
4330     struct spinlock lock;
4331     struct buf buf[NBUF];
4332
4333     // Linked list of all buffers, through prev/next.
4334     // head.next is most recently used.
4335     struct buf head;
4336 } bcache;
```

Buffer cache

```
3750 struct buf {
3751     int flags;
3752     uint dev;
3753     uint blockno;
3754     struct buf *prev; // LRU cache list
3755     struct buf *next;
3756     struct buf *qnext; // disk queue
3757     uchar data[BSIZE];
3758 };
3759 #define B_BUSY 0x1 // buffer is locked by some process
3760 #define B_VALID 0x2 // buffer has been read from disk
3761 #define B_DIRTY 0x4 // buffer needs to be written to disk
```

```
4329 struct {
4330     struct spinlock lock;
4331     struct buf buf[NBUF];
4332
4333     // Linked list of all buffers, through prev/next.
4334     // head.next is most recently used.
4335     struct buf head;
4336 } bcache;
```

- Array of buffers



```
3750 struct buf {
3751     int flags;
3752     uint dev;
3753     uint blockno;
3754     struct buf *prev; // LRU cache list
3755     struct buf *next;
3756     struct buf *qnext; // disk queue
3757     uchar data[BSIZE];
3758 };
3759 #define B_BUSY 0x1 // buffer is locked by some process
3760 #define B_VALID 0x2 // buffer has been read from disk
3761 #define B_DIRTY 0x4 // buffer needs to be written to disk

4329 struct {
4330     struct spinlock lock;
4331     struct buf buf[NBUF];
4332
4333     // Linked list of all buffers, through prev/next.
4334     // head.next is most recently used.
4335     struct buf head;
4336 } bcache;
```

- Cached data
- 512 bytes

Buffer cache


```
3750 struct buf {
3751     int flags;
3752     uint dev;
3753     uint blockno;
3754     struct buf *prev; // LRU cache list
3755     struct buf *next;
3756     struct buf *qnext; // disk queue
3757     uchar data[BSIZE];
3758 };
3759 #define B_BUSY 0x1 // buffer is locked by some process
3760 #define B_VALID 0x2 // buffer has been read from disk
3761 #define B_DIRTY 0x4 // buffer needs to be written to disk
```

- Flags

```
4329 struct {
4330     struct spinlock lock;
4331     struct buf buf[NBUF];
4332
4333     // Linked list of all buffers, through prev/next.
4334     // head.next is most recently used.
4335     struct buf head;
4336 } bcache;
```

Buffer cache

```
3750 struct buf {
3751     int flags;
3752     uint dev;
3753     uint blockno;
3754     struct buf *prev; // LRU cache list
3755     struct buf *next;
3756     struct buf *qnext; // disk queue
3757     uchar data[BSIZE];
3758 };
3759 #define B_BUSY 0x1 // buffer is locked by some process
3760 #define B_VALID 0x2 // buffer has been read from disk
3761 #define B_DIRTY 0x4 // buffer needs to be written to disk

4329 struct {
4330     struct spinlock lock;
4331     struct buf buf[NBUF];
4332
4333     // Linked list of all buffers, through prev/next.
4334     // head.next is most recently used.
4335     struct buf head;
4336 } bcache;
```

- Device
- We might have multiple disks

Buffer cache

```
3750 struct buf {
3751     int flags;
3752     uint dev;
3753     uint blockno;
3754     struct buf *prev; // LRU cache list
3755     struct buf *next;
3756     struct buf *qnext; // disk queue
3757     uchar data[BSIZE];
3758 };
3759 #define B_BUSY 0x1 // buffer is locked by some process
3760 #define B_VALID 0x2 // buffer has been read from disk
3761 #define B_DIRTY 0x4 // buffer needs to be written to disk

4329 struct {
4330     struct spinlock lock;
4331     struct buf buf[NBUF];
4332
4333     // Linked list of all buffers, through prev/next.
4334     // head.next is most recently used.
4335     struct buf head;
4336 } bcache;
```

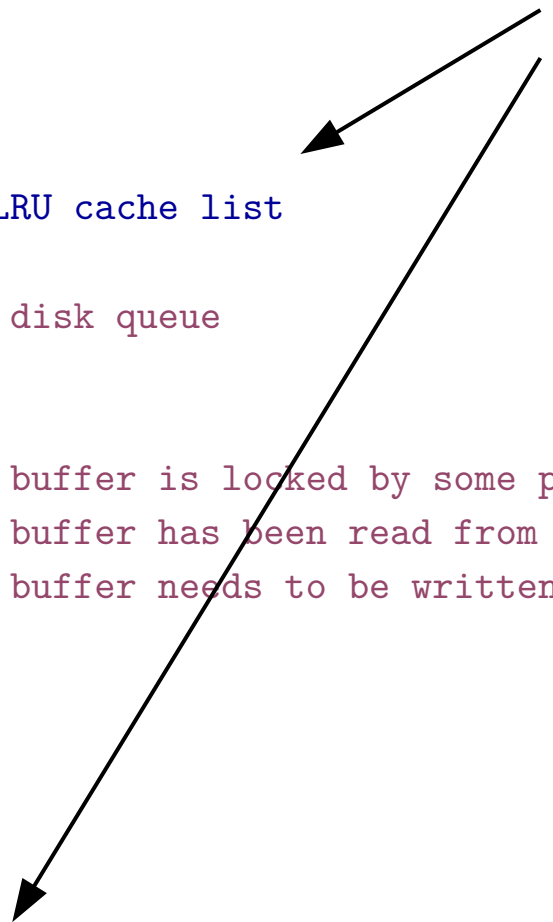
- Block number on disk

Buffer cache

```
3750 struct buf {
3751     int flags;
3752     uint dev;
3753     uint blockno;
3754     struct buf *prev; // LRU cache list
3755     struct buf *next;
3756     struct buf *qnext; // disk queue
3757     uchar data[BSIZE];
3758 };
3759 #define B_BUSY 0x1 // buffer is locked by some process
3760 #define B_VALID 0x2 // buffer has been read from disk
3761 #define B_DIRTY 0x4 // buffer needs to be written to disk

4329 struct {
4330     struct spinlock lock;
4331     struct buf buf[NBUF];
4332
4333     // Linked list of all buffers, through prev/next.
4334     // head.next is most recently used.
4335     struct buf head;
4336 } bcache;
```

• LRU list



Buffer cache

Buffer cache layer: interface

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

```
4401 struct buf*
4402 bread(uint dev, uint sector)
4403 {
4404     struct buf *b;
4405
4406     b = bget(dev, sector);
4407     if(!(b->flags & B_VALID)) {
4408         iderw(b);
4409     }
4410     return b;
4411 }
```

```
4415 bwrite(struct buf *b)
4416 {
4417     if((b->flags & B_BUSY) == 0)
4418         panic("bwrite");
4419     b->flags |= B_DIRTY;
4420     iderw(b);
4421 }
```

Block read and write operations

```
4365 static struct buf*
4366 bget(uint dev, uint blockno)
4367 {
4368     struct buf *b;
4369
4370     acquire(&bcache.lock);
4371
4372 loop:
4373     // Is the block already cached?
4374     for(b = bcache.head.next; b != &bcache.head; b = b->next){
4375         if(b->dev == dev && b->blockno == blockno){
4376             if(!(b->flags & B_BUSY)){
4377                 b->flags |= B_BUSY;
4378                 release(&bcache.lock);
4379                 return b;
4380             }
4381             sleep(b, &bcache.lock);
4382             goto loop;
4383     }
4384
```

Getting a block from a buffer cache (part 1)

```
4385
4386 // Not cached; recycle some non-busy and clean buffer.
4387 // "clean" because B_DIRTY and !B_BUSY means log.c
4388 // hasn't yet committed the changes to the buffer.
4389 for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4390     if((b->flags & B_BUSY)== 0 && (b->flags & B_DIRTY)== 0){
4391         b->dev = dev;
4392         b->blockno = blockno;
4393         b->flags = B_BUSY;
4394         release(&bcache.lock);
4395         return b;
4396     }
4397 }
4398 panic("bget: no buffers");
4399 }
```

**Getting a block
from a buffer
cache (part 2)**


```
4401 struct buf*
4402 bread(uint dev, uint sector)
4403 {
4404     struct buf *b;
4405
4406     b = bget(dev, sector);
4407     if(!(b->flags & B_VALID)) {
4408         iderw(b);
4409     }
4410     return b;
4411 }
```

```
4415 bwrite(struct buf *b)
4416 {
4417     if((b->flags & B_BUSY) == 0)
4418         panic("bwrite");
4419     b->flags |= B_DIRTY;
4420     iderw(b);
4421 }
```

Block read and write operations

```
4423 // Release a B_BUSY buffer.
4424 // Move to the head of the MRU list.
4425 void
4426 brelse(struct buf *b)
4427 {
4428     if((b->flags & B_BUSY) == 0)
4429         panic("brelse");
4430
4431     acquire(&bcache.lock);
4432
4433     b->next->prev = b->prev;
4434     b->prev->next = b->next;
4435     b->next = bcache.head.next;
4436     b->prev = &bcache.head;
4437     bcache.head.next->prev = b;
4438     bcache.head.next = b;
4439
4440     b->flags &= ~B_BUSY;
4441     wakeup(b);
4442
4443     release(&bcache.lock);
4444 }
```

Release buffer

- Maintain least recently used list
 - Move to the head

Common pattern

`bread()`

`bwrite()`

`brelease()`

- Read
- Write
- Release

```
4570 // Copy committed blocks from log to their home location
4571 static void
4572 install_trans(void)
4573 {
4574     int tail;
4575
4576     for (tail = 0; tail < log.lh.n; tail++) {
4577         struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log
block
4578         struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4579         memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
4580         bwrite(dbuf); // write dst to disk
4581         brelse(lbuf);
4582         brelse(dbuf);
4583     }
4584 }
```

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. if crash happens during file delete operation it can leave the file system with:
 - Ex #1: Directory entry pointing to a free inode
 - Ex #2: 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)


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

Typical use of transactions

```
4532 struct logheader {
4533     int n;
4534     int block[LOGSIZE];
4535 };
4536
4537 struct log {
4538     struct spinlock lock;
4539     int start;
4540     int size;
4541     int outstanding; // how many FS sys calls are
                        // executing.
4542     int committing; // in commit(), please wait.
4543     int dev;
4544     struct logheader lh;
4545 };
```

Log (in memory)

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

Typical use of transactions

```
4626 // called at the start of each FS system call.
4627 void
4628 begin_op(void)
4629 {
4630     acquire(&log.lock);
4631     while(1){
4632         if(log.committing){
4633             sleep(&log, &log.lock);
4634         } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4635             // this op might exhaust log space; wait for commit.
4636             sleep(&log, &log.lock);
4637         } else {
4638             log.outstanding += 1;
4639             release(&log.lock);
4640             break;
4641         }
4642     }
4643 }
```

begin_op()

- Case #1
 - Log is being committed
 - Sleep

```
4626 // called at the start of each FS system call.
4627 void
4628 begin_op(void)
4629 {
4630     acquire(&log.lock);
4631     while(1){
4632         if(log.committing){
4633             sleep(&log, &log.lock);
4634         } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4635             // this op might exhaust log space; wait for commit.
4636             sleep(&log, &log.lock);
4637         } else {
4638             log.outstanding += 1;
4639             release(&log.lock);
4640             break;
4641         }
4642     }
4643 }
```

begin_op()

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

```
4626 // called at the start of each FS system call.
4627 void
4628 begin_op(void)
4629 {
4630     acquire(&log.lock);
4631     while(1){
4632         if(log.committing){
4633             sleep(&log, &log.lock);
4634         } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4635             // this op might exhaust log space; wait for commit.
4636             sleep(&log, &log.lock);
4637         } else {
4638             log.outstanding += 1;
4639             release(&log.lock);
4640             break;
4641         }
4642     }
4643 }
```

begin_op()

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

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

Typical use of transactions

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

```
4722 log_write(struct buf *b)
4723 {
4724     int i;
4725
4726     if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4727         panic("too big a transaction");
4728     if (log.outstanding < 1)
4729         panic("log_write outside of trans");
4730
4731     acquire(&log.lock);
4732     for (i = 0; i < log.lh.n; i++) {
4733         if (log.lh.block[i] == b->blockno) // log absorbtion
4734             break;
4735     }
4736     log.lh.block[i] = b->blockno;
4737     if (i == log.lh.n)
4738         log.lh.n++;
4739     b->flags |= B_DIRTY; // prevent eviction
4740     release(&log.lock);
4741 }
```

log_write

- Check if already in log


```
4722 log_write(struct buf *b)
4723 {
4724     int i;
4725
4726     if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4727         panic("too big a transaction");
4728     if (log.outstanding < 1)
4729         panic("log_write outside of trans");
4730
4731     acquire(&log.lock);
4732     for (i = 0; i < log.lh.n; i++) {
4733         if (log.lh.block[i] == b->blockno) // log absorption
4734             break;
4735     }
4736     log.lh.block[i] = b->blockno;
4737     if (i == log.lh.n)
4738         log.lh.n++;
4739     b->flags |= B_DIRTY; // prevent eviction
4740     release(&log.lock);
4741 }
```

log_write

- Add to the log
- Prevent eviction

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

Typical use of transactions

```
4653 end_op(void)
4654 {
4655     int do_commit = 0;
4656
4657     acquire(&log.lock);
4658     log.outstanding -= 1;
4661     if(log.outstanding == 0){
4662         do_commit = 1;
4663         log.committing = 1;
4664     } else {
4665         // begin_op() may be waiting for log space.
4666         wakeup(&log);
4667     }
4668     release(&log.lock);
4669
4670     if(do_commit){
4671         // call commit w/o holding locks, since not allowed
4672         // to sleep with locks.
4673         commit();
4674         acquire(&log.lock);
4675         log.committing = 0;
4676         wakeup(&log);
4677         release(&log.lock);
4678     }
4679 }
```

end_op()

```
4653 end_op(void)
4654 {
4655     int do_commit = 0;
4656
4657     acquire(&log.lock);
4658     log.outstanding -= 1;
4661     if(log.outstanding == 0){
4662         do_commit = 1;
4663         log.committing = 1;
4664     } else {
4665         // begin_op() may be waiting for log space.
4666         wakeup(&log);
4667     }
4668     release(&log.lock);
4669
4670     if(do_commit){
4671         // call commit w/o holding locks, since not allowed
4672         // to sleep with locks.
4673         commit();
4674         acquire(&log.lock);
4675         log.committing = 0;
4676         wakeup(&log);
4677         release(&log.lock);
4678     }
4679 }
```

end_op()

```
4701 commit()
4702 {
4703     if (log.lh.n > 0) {
4704         write_log(); // Write modified blocks
                        // from cache to log
4705         write_head(); // Write header to disk --
                        // the real commit
4706         install_trans(); // Now install writes
                            // to home locations
4707         log.lh.n = 0;
4708         write_head(); // Erase the transaction
                        // from the log
4709     }
4710 }
```

commit()

```
4681 // Copy modified blocks from cache to log.
4682 static void
4683 write_log(void)
4684 {
4685     int tail;
4686
4687     for (tail = 0; tail < log.lh.n; tail++) {
4688         struct buf *to = bread(log.dev,
4689                                 log.start+tail+1); // log block
4689         struct buf *from = bread(log.dev,
4690                                 log.lh.block[tail]); // cache block
4690         memmove(to->data, from->data, BSIZE);
4691         bwrite(to); // write the log
4692         brelse(from);
4693         brelse(to);
4694     }
4695 }
```

write_log()

- Loop through the entire log

```
4681 // Copy modified blocks from cache to log.
4682 static void
4683 write_log(void)
4684 {
4685     int tail;
4686
4687     for (tail = 0; tail < log.lh.n; tail++) {
4688         struct buf *to = bread(log.dev,
4689                                log.start+tail+1); // log block
4689         struct buf *from = bread(log.dev,
4690                                   log.lh.block[tail]); // cache block
4690         memmove(to->data, from->data, BSIZE);
4691         bwrite(to); // write the log
4692         brelse(from);
4693         brelse(to);
4694     }
4695 }
```

write_log()

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

```
4681 // Copy modified blocks from cache to log.
4682 static void
4683 write_log(void)
4684 {
4685     int tail;
4686
4687     for (tail = 0; tail < log.lh.n; tail++) {
4688         struct buf *to = bread(log.dev,
4689                                log.start+tail+1); // log block
4689         struct buf *from = bread(log.dev,
4690                                   log.lh.block[tail]); // cache block
4690         memmove(to->data, from->data, BSIZE);
4691         bwrite(to); // write the log
4692         brelse(from);
4693         brelse(to);
4694     }
4695 }
```

write_log()

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


```
4681 // Copy modified blocks from cache to log.
4682 static void
4683 write_log(void)
4684 {
4685     int tail;
4686
4687     for (tail = 0; tail < log.lh.n; tail++) {
4688         struct buf *to = bread(log.dev,
4689                                log.start+tail+1); // log block
4689         struct buf *from = bread(log.dev,
4690                                   log.lh.block[tail]); // cache block
4690         memmove(to->data, from->data, BSIZE);
4691         bwrite(to); // write the log
4692         brelse(from);
4693         brelse(to);
4694     }
4695 }
```

write_log()

- Copy block data into the log

```
4681 // Copy modified blocks from cache to log.
4682 static void
4683 write_log(void)
4684 {
4685     int tail;
4686
4687     for (tail = 0; tail < log.lh.n; tail++) {
4688         struct buf *to = bread(log.dev,
4689                                log.start+tail+1); // log block
4689         struct buf *from = bread(log.dev,
4690                                   log.lh.block[tail]); // cache block
4690         memmove(to->data, from->data, BSIZE);
4691         bwrite(to); // write the log
4692         brelse(from);
4693         brelse(to);
4694     }
4695 }
```

write_log()

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

```
4701 commit()
4702 {
4703     if (log.lh.n > 0) {
4704         write_log(); // Write modified blocks
                        // from cache to log
4705         write_head(); // Write header to disk --
                        // the real commit
4706         install_trans(); // Now install writes
                            // to home locations
4707         log.lh.n = 0;
4708         write_head(); // Erase the transaction
                        // from the log
4709     }
4710 }
```

commit()

```
4600 // Write in-memory log header to disk.
4601 // This is the true point at which the
4602 // current transaction commits.
4603 static void
4604 write_head(void)
4605 {
4606     struct buf *buf = bread(log.dev, log.start);
4607     struct logheader *hb = (struct logheader *)
                                (buf->data);
4608     int i;
4609     hb->n = log.lh.n;
4610     for (i = 0; i < log.lh.n; i++) {
4611         hb->block[i] = log.lh.block[i];
4612     }
4613     bwrite(buf);
4614     brelse(buf);
4615 }
```

write_head()

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

```

4600 // Write in-memory log header to disk.
4601 // This is the true point at which the
4602 // current transaction commits.
4603 static void
4604 write_head(void)
4605 {
4606     struct buf *buf = bread(log.dev, log.start);
4607     struct logheader *hb = (struct logheader *)
                                (buf->data);
4608     int i;
4609     hb->n = log.lh.n;
4610     for (i = 0; i < log.lh.n; i++) {
4611         hb->block[i] = log.lh.block[i];
4612     }
4613     bwrite(buf);
4614     brelse(buf);
4615 }

```

write_head()

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

```
4600 // Write in-memory log header to disk.
4601 // This is the true point at which the
4602 // current transaction commits.
4603 static void
4604 write_head(void)
4605 {
4606     struct buf *buf = bread(log.dev, log.start);
4607     struct logheader *hb = (struct logheader *)
                                (buf->data);
4608     int i;
4609     hb->n = log.lh.n;
4610     for (i = 0; i < log.lh.n; i++) {
4611         hb->block[i] = log.lh.block[i];
4612     }
4613     bwrite(buf);
4614     brelse(buf);
4615 }
```

write_head()

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

```
4600 // Write in-memory log header to disk.
4601 // This is the true point at which the
4602 // current transaction commits.
4603 static void
4604 write_head(void)
4605 {
4606     struct buf *buf = bread(log.dev, log.start);
4607     struct logheader *hb = (struct logheader *)
                                (buf->data);
4608     int i;
4609     hb->n = log.lh.n;
4610     for (i = 0; i < log.lh.n; i++) {
4611         hb->block[i] = log.lh.block[i];
4612     }
4613     bwrite(buf);
4614     brelse(buf);
4615 }
```

write_head()

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

```
4600 // Write in-memory log header to disk.
4601 // This is the true point at which the
4602 // current transaction commits.
4603 static void
4604 write_head(void)
4605 {
4606     struct buf *buf = bread(log.dev, log.start);
4607     struct logheader *hb = (struct logheader *)
                                (buf->data);
4608     int i;
4609     hb->n = log.lh.n;
4610     for (i = 0; i < log.lh.n; i++) {
4611         hb->block[i] = log.lh.block[i];
4612     }
4613     bwrite(buf);
4614     brelse(buf);
4615 }
```

write_head()

- Write block to disk
- Release


```
4701 commit()
4702 {
4703     if (log.lh.n > 0) {
4704         write_log(); // Write modified blocks
                        // from cache to log
4705         write_head(); // Write header to disk --
                        // the real commit
4706         install_trans(); // Now install writes
                        // to home locations
4707         log.lh.n = 0;
4708         write_head(); // Erase the transaction
                        // from the log
4709     }
4710 }
```

commit()

```
4570 // Copy committed blocks from log to their home location
4571 static void
4572 install_trans(void)
4573 {
4574     int tail;
4575
4576     for (tail = 0; tail < log.lh.n; tail++) {
4577         struct buf *lbuf = bread(log.dev,
4578                                 log.start+tail+1); // read log block
4579         struct buf *dbuf = bread(log.dev,
4580                                 log.lh.block[tail]); // read dst
4581         memmove(dbuf->data, lbuf->data, BSIZE); // copy block
4582                                                // to dst
4583         bwrite(dbuf); // write dst to disk
4584         brelse(lbuf);
4585         brelse(dbuf);
4586     }
4587 }
```

install_trans()

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

```
4570 // Copy committed blocks from log to their home location
4571 static void
4572 install_trans(void)
4573 {
4574     int tail;
4575
4576     for (tail = 0; tail < log.lh.n; tail++) {
4577         struct buf *lbuf = bread(log.dev,
4578                                 log.start+tail+1); // read log block
4579         struct buf *dbuf = bread(log.dev,
4580                                 log.lh.block[tail]); // read dst
4581         memmove(dbuf->data, lbuf->data, BSIZE); // copy block
4582                                                // to dst
4583         bwrite(dbuf); // write dst to disk
4584         brelse(lbuf);
4585         brelse(dbuf);
4586     }
4587 }
```

install_trans()

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

```
4570 // Copy committed blocks from log to their home location
4571 static void
4572 install_trans(void)
4573 {
4574     int tail;
4575
4576     for (tail = 0; tail < log.lh.n; tail++) {
4577         struct buf *lbuf = bread(log.dev,
4578                                 log.start+tail+1); // read log block
4579         struct buf *dbuf = bread(log.dev,
4580                                 log.lh.block[tail]); // read dst
4581         memmove(dbuf->data, lbuf->data, BSIZE); // copy block
4582                                                // to dst
4583         bwrite(dbuf); // write dst to disk
4584         brelse(lbuf);
4585         brelse(dbuf);
4586     }
4587 }
```

install_trans()

- Copy data

```
4570 // Copy committed blocks from log to their home location
4571 static void
4572 install_trans(void)
4573 {
4574     int tail;
4575
4576     for (tail = 0; tail < log.lh.n; tail++) {
4577         struct buf *lbuf = bread(log.dev,
4578                                 log.start+tail+1); // read log block
4579         struct buf *dbuf = bread(log.dev,
4580                                 log.lh.block[tail]); // read dst
4581         memmove(dbuf->data, lbuf->data, BSIZE); // copy block
4582                                                // to dst
4583         bwrite(dbuf); // write dst to disk
4584         brelse(lbuf);
4585         brelse(dbuf);
4586     }
4587 }
```

install_trans()

- Write the block to disk
- Release both blocks

```
4701 commit()
4702 {
4703     if (log.lh.n > 0) {
4704         write_log(); // Write modified blocks
                        // from cache to log
4705         write_head(); // Write header to disk --
                        // the real commit
4706         install_trans(); // Now install writes
                        // to home locations
4707         log.lh.n = 0;
4708         write_head(); // Erase the transaction
                        // from the log
4709     }
4710 }
```

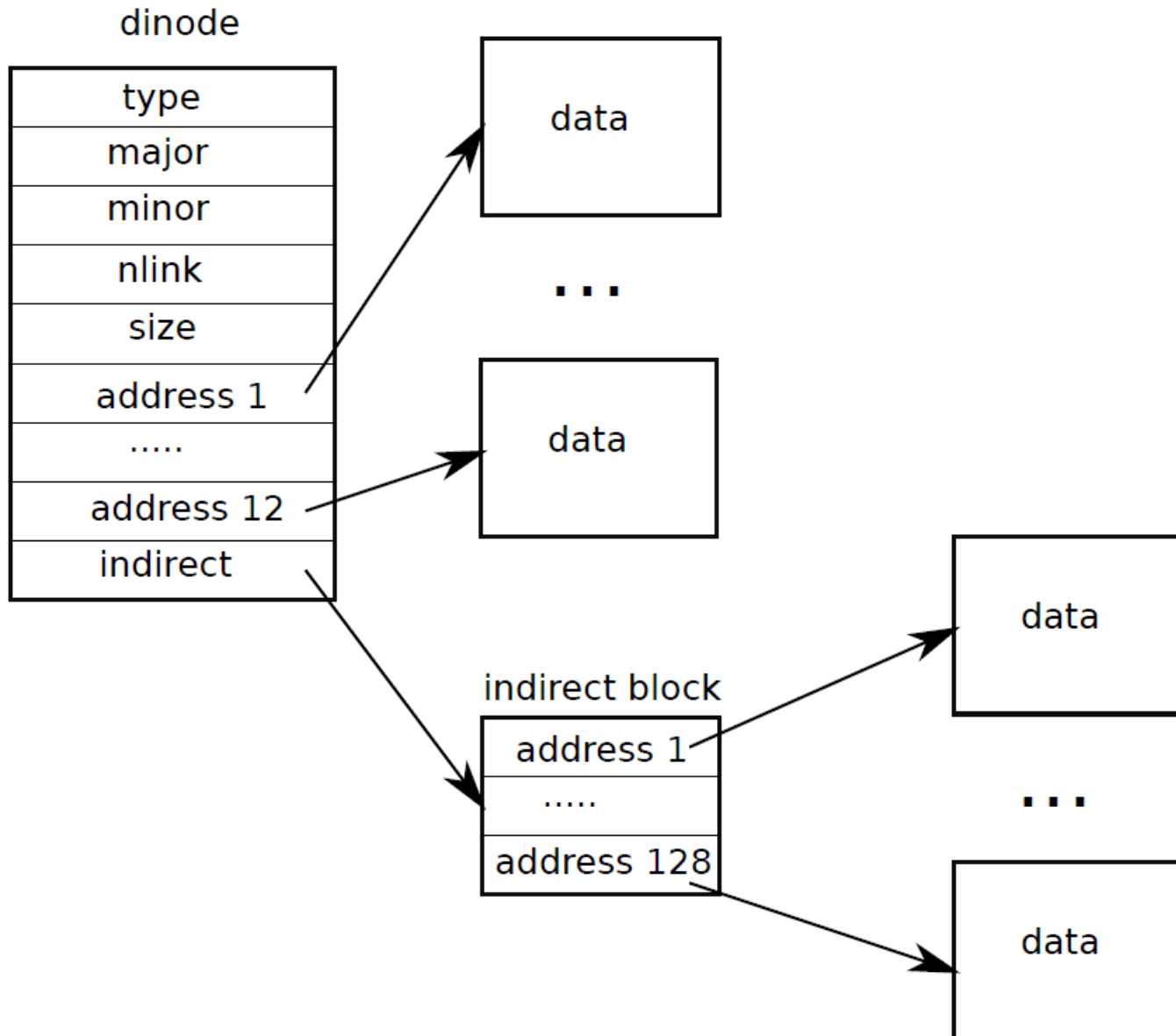
commit()

Inode layer

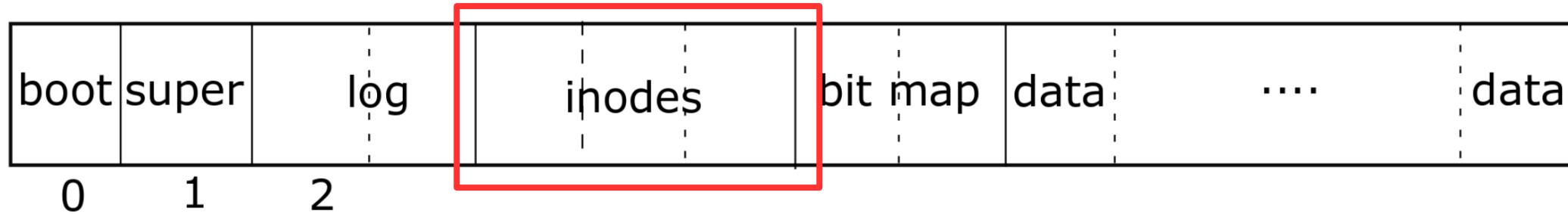
Inode

- 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

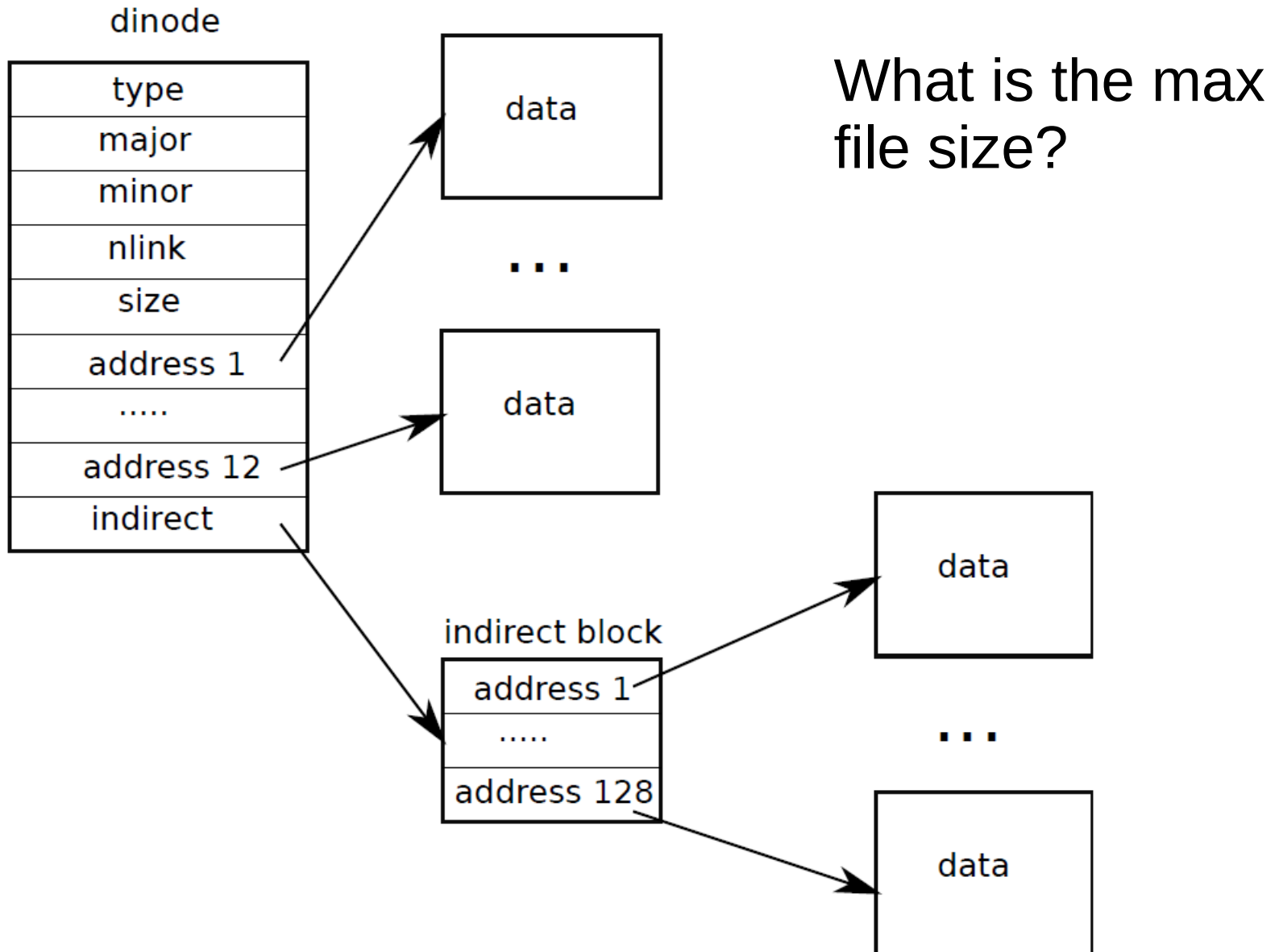


- Inodes are stored as an array on disk
 - `sb.startinode`
- Each inode has a number (indicating its position on disk)
- The kernel keeps a cache of inodes in memory
 - Synchronization

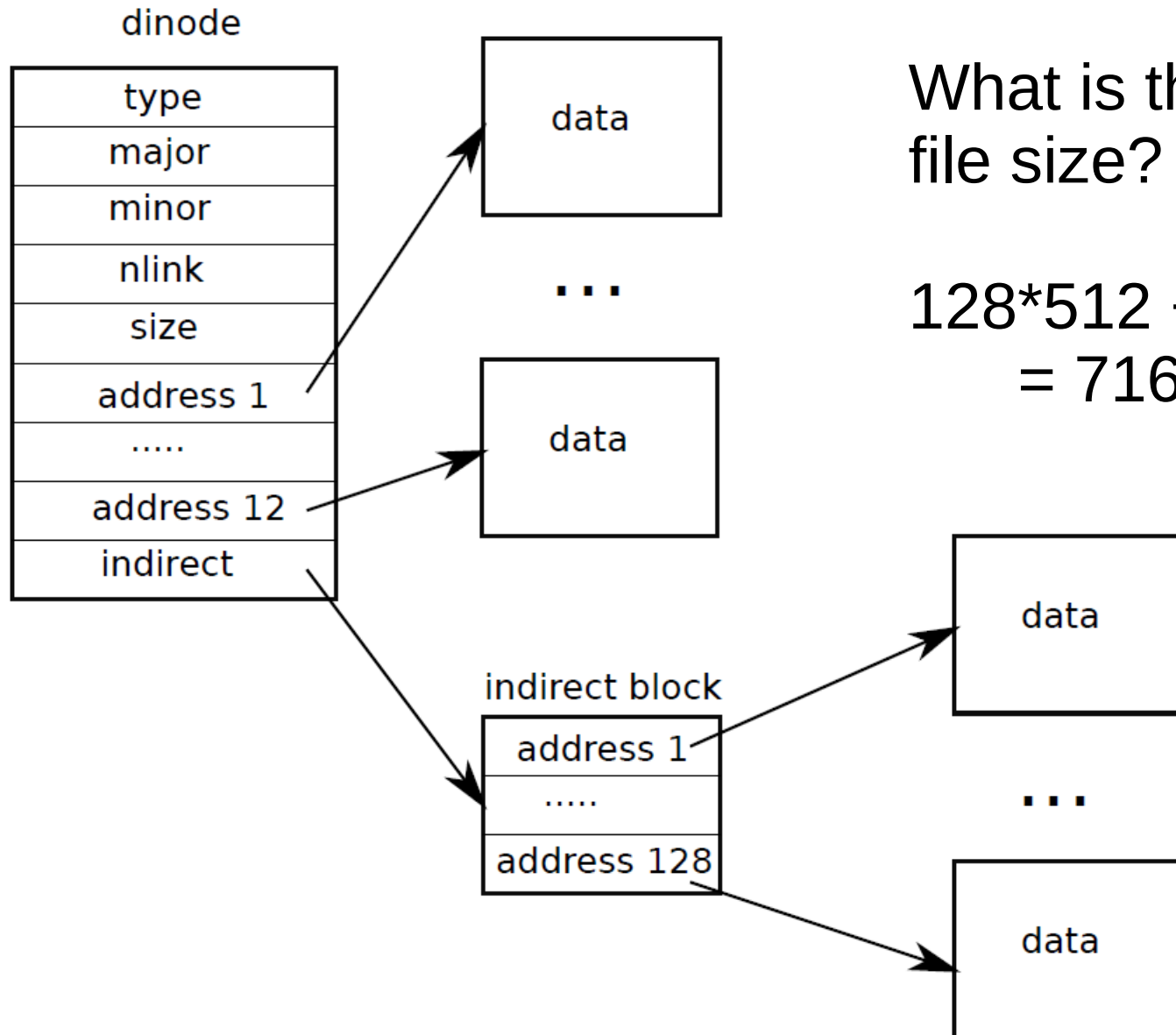
Inode on disk

```
3927 // On-disk inode structure
3928 struct dinode {
3929     short type; // File type
3930     short major; // Major device number (T_DEV
                    only)
3931     short minor; // Minor device number (T_DEV
                    only)
3932     short nlink; // Number of links to inode in
                    file system
3933     uint size; // Size of file (bytes)
3934     uint addrs[NDIRECT+1]; // Data block addresses
3935 };
```

Representing files on disk



Representing files on disk



What is the max
file size?

$$128 * 512 + 12 * 512 = 71680$$

Inode in memory

```
4011 // in-memory copy of an inode
4012 struct inode {
4013     uint dev; // Device number
4014     uint inum; // Inode number
4015     int ref; // Reference count
4016     int flags; // I_BUSY, I_INVALID
4017
4018     short type; // copy of disk inode
4019     short major;
4020     short minor;
4021     short nlink;
4022     uint size;
4023     uint addrs[NDIRECT+1];
4024 };
```

In-memory cache of inodes

```
4912 struct {  
4913     struct spinlock lock;  
4914     struct inode inode[NINODE];  
4915 } 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()

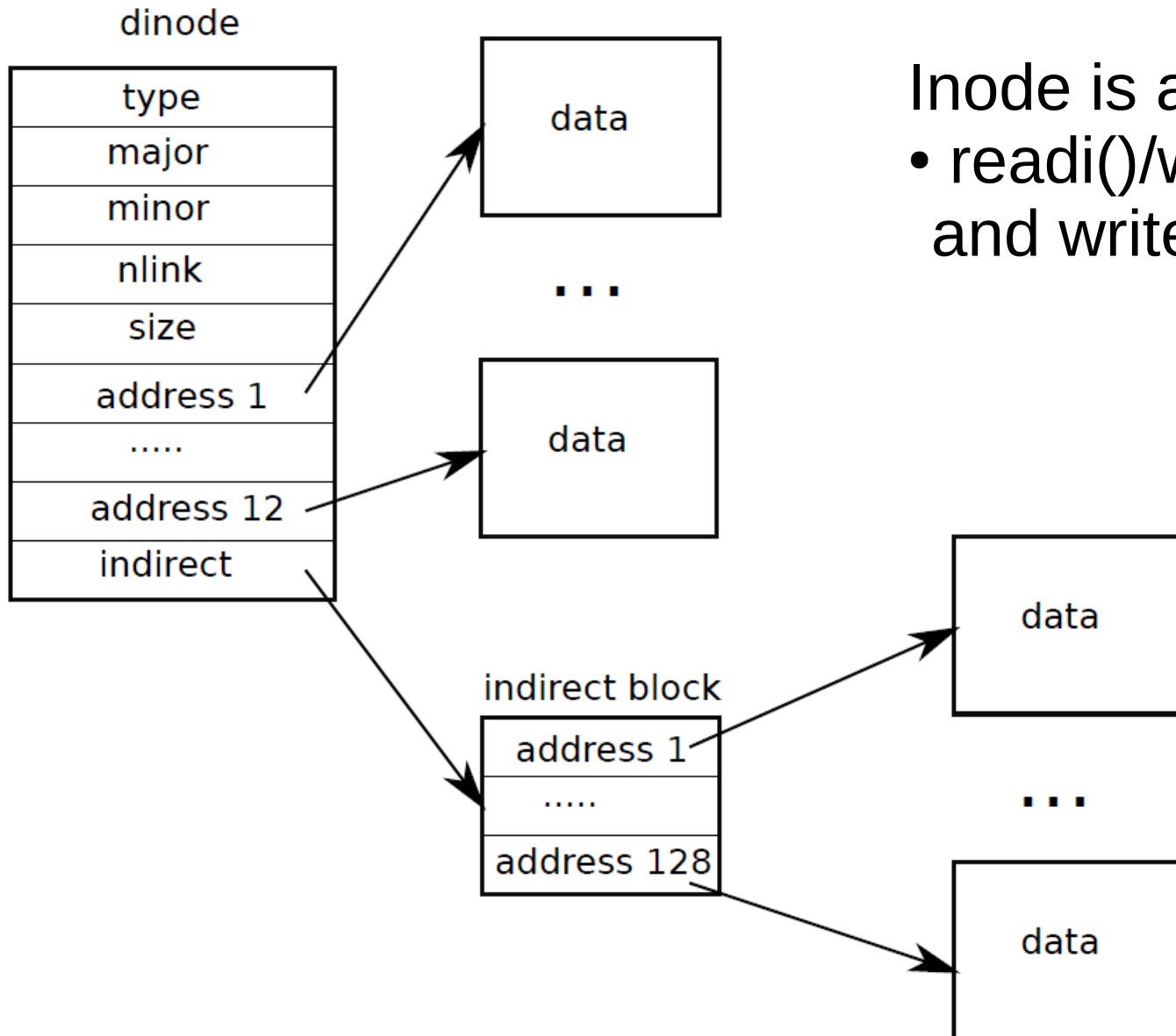
Accessing inodes

```
4894 // Thus a typical sequence is:
4895 // ip = iget(dev, inum)
4896 // ilock(ip)
4897 // ... examine and modify ip->xxx ...
4898 // iunlock(ip)
4899 // iput(ip)
```

iget()

```
5004 iget(uint dev, uint inum) {
...
5008     acquire(&icache.lock);
5010     // Is the inode already cached?
5011     empty = 0;
5012     for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5013         if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5014             ip->ref++;
5015             release(&icache.lock);
5016             return ip;
5017         }
5018         if(empty == 0 && ip->ref == 0) // Remember empty slot.
5019             empty = ip;
5020     }
...
5029     ip->ref = 1;
...
5031     release(&icache.lock);
5033     return ip;
5034 }
```

Reading and writing inodes



Inode is a file

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

```
5864 int
5865 sys_read(void)
5866 {
5867     struct file *f;
5868     int n;
5869     char *p;
5870
5871     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5872         return -1;
5873     return fileread(f, p, n);
5874 }
```

Example: sys_read()

- Question:
 - Where does f come from?

```
5816 // Fetch the nth word-sized system call argument as a file descriptor
5817 // and return both the descriptor and the corresponding struct file.
5818 static int
5819 argfd(int n, int *pfd, struct file **pf)
5820 {
5821     int fd;
5822     struct file *f;
5823
5824     if(argint(n, &fd) < 0)
5825         return -1;
5826     if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5827         return -1;
5828     if(pfd)
5829         *pfd = fd;
5830     if(pf)
5831         *pf = f;
5832     return 0;
5833 }
```

argfd()

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

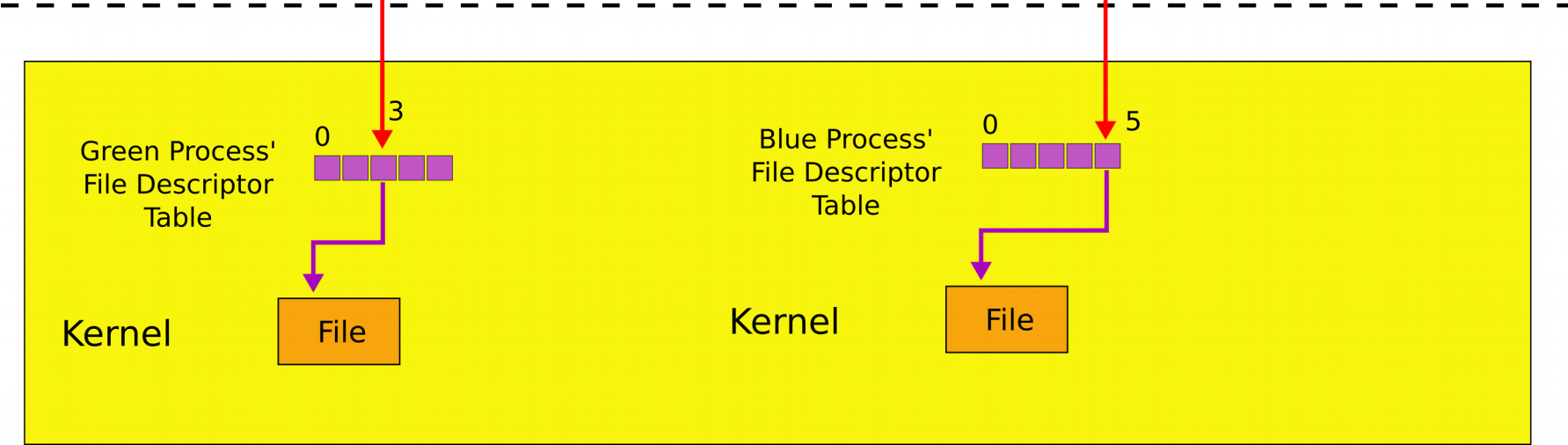
File descriptors: two processes

Process (e.g., Apache, shell)

```
read(3, buf, size);
```

Process (e.g., Apache, shell)

```
read(5, buf, size);
```



```

2353 struct proc {
2354     uint sz;                // Size of process memory (bytes)
2355     pde_t* pgdir;          // Page table
2356     char *kstack;         // Bottom of kernel stack for this
process
2357     enum procstate state;  // Process state
2358     int pid;               // Process ID
2359     struct proc *parent;   // Parent process
2360     struct trapframe *tf;  // Trap frame for current syscall
2361     struct context *context; // swtch() here to run process
2362     void *chan;           // If non-zero, sleeping on chan
2363     int killed;           // If non-zero, have been killed
2364     struct file *ofile[NOFILE]; // Open files
2365     struct inode *cwd;    // Current directory
2366     char name[16];       // Process name (debugging)
2367 };

```

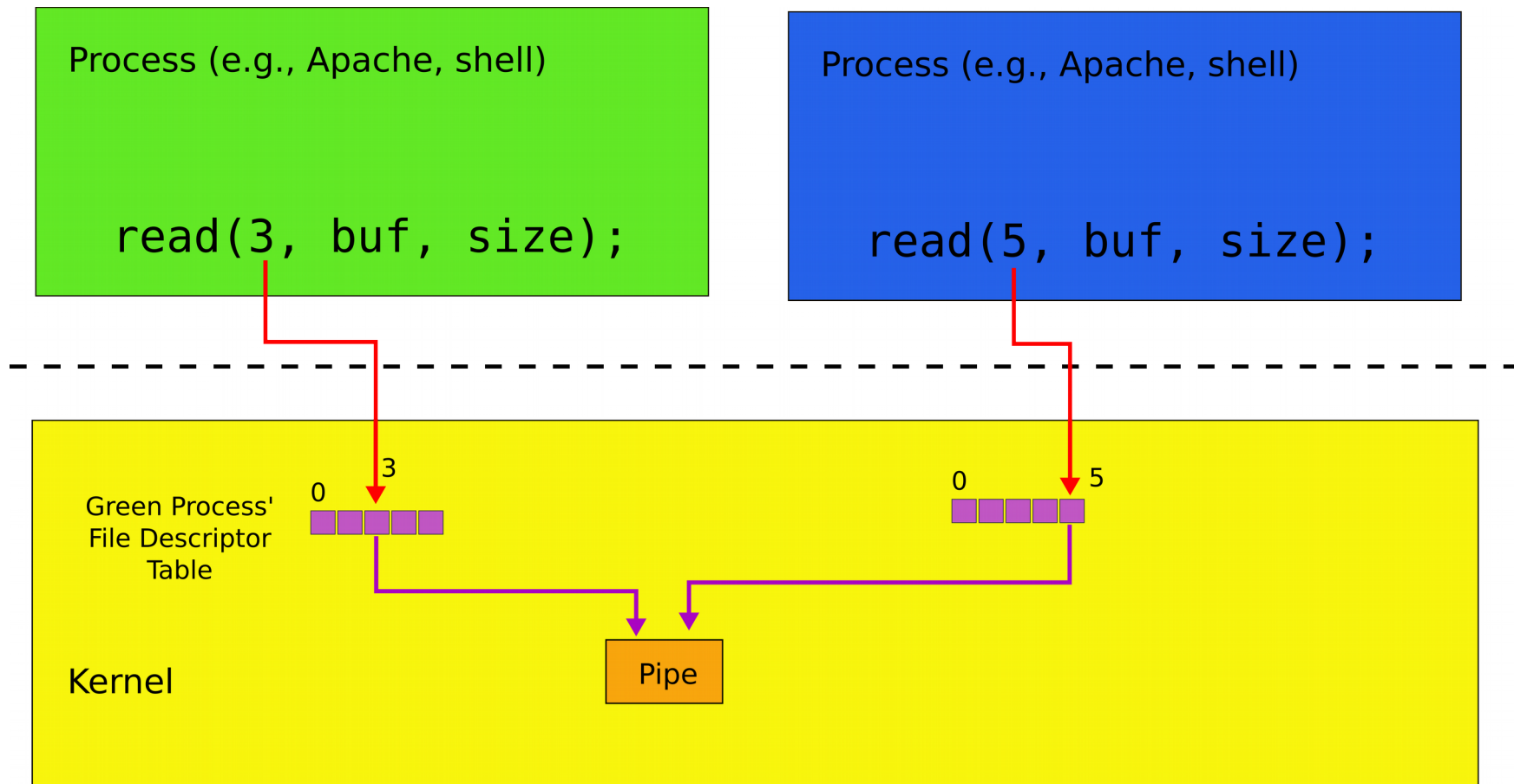
- struct proc has an array of struct file pointers
 - Each element is a “file descriptor”

```
4000 struct file {
4001     enum { FD_NONE, FD_PIPE, FD_INODE } type;
4002     int ref; // reference count
4003     char readable;
4004     char writable;
4005     struct pipe *pipe;
4006     struct inode *ip;
4007     uint off;
4008 };
```

struct file

- 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



```
5714 int
5715 fileread(struct file *f, char *addr, int n)
5716 {
5717     int r;
5718
5719     if(f->readable == 0)
5720         return -1;
5721     if(f->type == FD_PIPE)
5722         return piperead(f->pipe, addr, n);
5723     if(f->type == FD_INODE){
5724         ilock(f->ip);
5725         if((r = readi(f->ip, addr, f->off, n)) > 0)
5726             f->off += r;
5727         iunlock(f->ip);
5728         return r;
5729     }
5730     panic("fileread");
5731 }
```

readi()

```
5714 int
5715 fileread(struct file *f, char *addr, int n)
5716 {
5717     int r;
5718
5719     if(f->readable == 0)
5720         return -1;
5721     if(f->type == FD_PIPE)
5722         return piperead(f->pipe, addr, n);
5723     if(f->type == FD_INODE){
5724         ilock(f->ip);
5725         if((r = readi(f->ip, addr, f->off, n)) > 0)
5726             f->off += r;
5727         iunlock(f->ip);
5728         return r;
5729     }
5730     panic("fileread");
5731 }
```

readi()

```
5714 int
5715 fileread(struct file *f, char *addr, int n)
5716 {
5717     int r;
5718
5719     if(f->readable == 0)
5720         return -1;
5721     if(f->type == FD_PIPE)
5722         return piperead(f->pipe, addr, n);
5723     if(f->type == FD_INODE){
5724         ilock(f->ip);
5725         if((r = readi(f->ip, addr, f->off, n)) > 0)
5726             f->off += r;
5727         iunlock(f->ip);
5728         return r;
5729     }
5730     panic("fileread");
5731 }
```

readi()

```
5714 int
5715 fileread(struct file *f, char *addr, int n)
5716 {
5717     int r;
5718
5719     if(f->readable == 0)
5720         return -1;
5721     if(f->type == FD_PIPE)
5722         return piperead(f->pipe, addr, n);
5723     if(f->type == FD_INODE){
5724         ilock(f->ip);
5725         if((r = readi(f->ip, addr, f->off, n)) > 0)
5726             f->off += r;
5727         iunlock(f->ip);
5728         return r;
5729     }
5730     panic("fileread");
5731 }
```

readi()

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

```
5252 readi(struct inode *ip, char *dst, uint off, uint n)
5253 {
5254     uint tot, m;
5255     struct buf *bp;
5256
5257     ...
5263     if(off > ip->size || off + n < off)
5264         return -1;
5265     if(off + n > ip->size)
5266         n = ip->size - off;
5267
5268     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5269         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5270         m = min(n - tot, BSIZE - off%BSIZE);
5271         memmove(dst, bp->data + off%BSIZE, m);
5272         brelse(bp);
5273     }
5274     return n;
5275 }
```

readi()

- What is this check for?

```
5252 readi(struct inode *ip, char *dst, uint off, uint n)
5253 {
5254     uint tot, m;
5255     struct buf *bp;
5256
5257     ...
5263     if(off > ip->size || off + n < off)
5264         return -1;
5265     if(off + n > ip->size)
5266         n = ip->size - off;
5267
5268     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5269         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5270         m = min(n - tot, BSIZE - off%BSIZE);
5271         memmove(dst, bp->data + off%BSIZE, m);
5272         brelse(bp);
5273     }
5274     return n;
5275 }
```

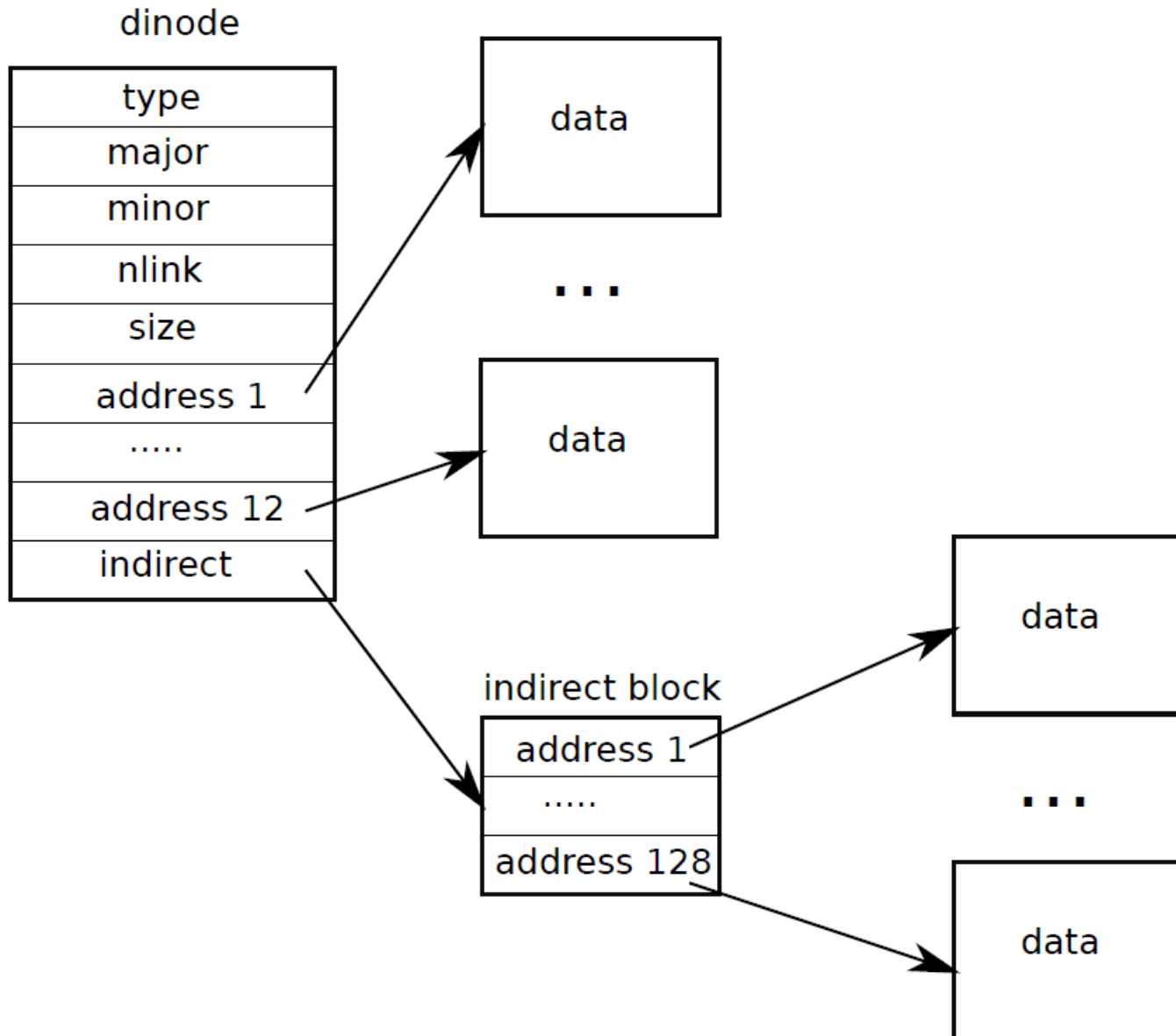
readi()

```
5252 readi(struct inode *ip, char *dst, uint off, uint n)
5253 {
5254     uint tot, m;
5255     struct buf *bp;
5256
...
5263     if(off > ip->size || off + n < off)
5264         return -1;
5265     if(off + n > ip->size)
5266         n = ip->size - off;
5267
5268     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5269         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5270         m = min(n - tot, BSIZE - off%BSIZE);
5271         memmove(dst, bp->data + off%BSIZE, m);
5272         brelse(bp);
5273     }
5274     return n;
5275 }
```

readi()

- What is this bmap() function?

Representing files on disk



bmap()

```
5159 static uint
5160 bmap(struct inode *ip, uint bn)
5161 {
...
5165     if(bn < NDIRECT){
5166         if((addr = ip->addrs[bn]) == 0)
5167             ip->addrs[bn] = addr = balloc(ip->dev);
5168         return addr;
5169     }
5170     bn -= NDIRECT;
5171
5172     if(bn < NINDIRECT){
5173         // Load indirect block, allocating if necessary.
5174         if((addr = ip->addrs[NDIRECT]) == 0)
5175             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176         bp = bread(ip->dev, addr);
5177         a = (uint*)bp->data;
5178         if((addr = a[bn]) == 0){
5179             a[bn] = addr = balloc(ip->dev);
5180             log_write(bp);
5181         }
5182         brelse(bp);
5183         return addr;
5184     }
...
5187 }
```

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

bmap()

```
5159 static uint
5160 bmap(struct inode *ip, uint bn)
5161 {
...
5165     if(bn < NDIRECT){
5166         if((addr = ip->addrs[bn]) == 0)
5167             ip->addrs[bn] = addr = balloc(ip->dev);
5168         return addr;
5169     }
5170     bn -= NDIRECT;
5171
5172     if(bn < NINDIRECT){
5173         // Load indirect block, allocating if necessary.
5174         if((addr = ip->addrs[NDIRECT]) == 0)
5175             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176         bp = bread(ip->dev, addr);
5177         a = (uint*)bp->data;
5178         if((addr = a[bn]) == 0){
5179             a[bn] = addr = balloc(ip->dev);
5180             log_write(bp);
5181         }
5182         brelse(bp);
5183         return addr;
5184     }
...
5187 }
```

- No it's beyond NDIRECT

bmap()

```
5159 static uint
5160 bmap(struct inode *ip, uint bn)
5161 {
...
5165     if(bn < NDIRECT){
5166         if((addr = ip->addrs[bn]) == 0)
5167             ip->addrs[bn] = addr = balloc(ip->dev);
5168         return addr;
5169     }
5170     bn -= NDIRECT;
5171
5172     if(bn < NINDIRECT){
5173         // Load indirect block, allocating if necessary.
5174         if((addr = ip->addrs[NDIRECT]) == 0)
5175             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176         bp = bread(ip->dev, addr);
5177         a = (uint*)bp->data;
5178         if((addr = a[bn]) == 0){
5179             a[bn] = addr = balloc(ip->dev);
5180             log_write(bp);
5181         }
5182         brelse(bp);
5183         return addr;
5184     }
...
5187 }
```

- Read an indirect block

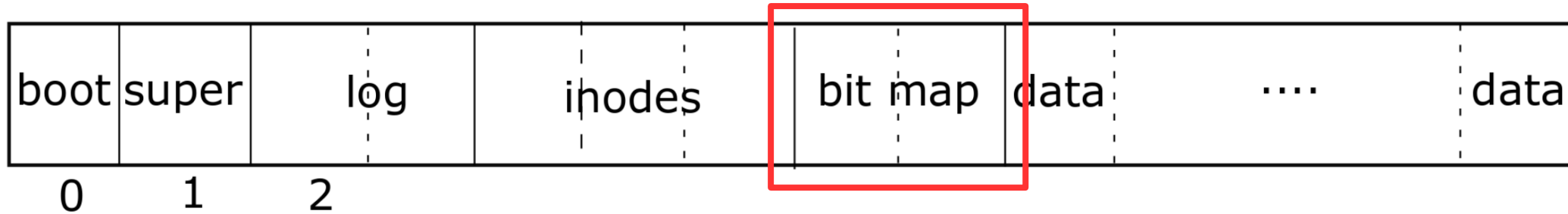
bmap()

```
5159 static uint
5160 bmap(struct inode *ip, uint bn)
5161 {
...
5165     if(bn < NDIRECT){
5166         if((addr = ip->addrs[bn]) == 0)
5167             ip->addrs[bn] = addr = balloc(ip->dev);
5168         return addr;
5169     }
5170     bn -= NDIRECT;
5171
5172     if(bn < NINDIRECT){
5173         // Load indirect block, allocating if necessary.
5174         if((addr = ip->addrs[NDIRECT]) == 0)
5175             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176         bp = bread(ip->dev, addr);
5177         a = (uint*)bp->data;
5178         if((addr = a[bn]) == 0){
5179             a[bn] = addr = balloc(ip->dev);
5180             log_write(bp);
5181         }
5182         brelse(bp);
5183         return addr;
5184     }
...
5187 }
```

- 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

```
4802 // Allocate a zeroed disk block.
4803 static uint
4804 balloc(uint dev)
4805 {
4806     int b, bi, m;
4807     struct buf *bp;
4808
4809     bp = 0;
4810     for(b = 0; b < sb.size; b += BPB){
4811         bp = bread(dev, BBLOCK(b, sb));
4812         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4813             m = 1 << (bi % 8);
4814             if((bp->data[bi/8] & m) == 0){ // Is block free?
4815                 bp->data[bi/8] |= m; // Mark block in use.
4816                 log_write(bp);
4817                 brelse(bp);
4818                 bzero(dev, b + bi);
4819                 return b + bi;
4820             }
4821         }
4822         brelse(bp);
4823     }
4824     panic("balloc: out of blocks");
4825 }
```

balloc()


```

4802 // Allocate a zeroed disk block.
4803 static uint
4804 balloc(uint dev)
4805 {
4806     int b, bi, m;
4807     struct buf *bp;
4808
4809     bp = 0;
4810     for(b = 0; b < sb.size; b += BPB){
4811         bp = bread(dev, BBLOCK(b, sb));
4812         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4813             m = 1 << (bi % 8);
4814             if((bp->data[bi/8] & m) == 0){ // Is block free?
4815                 bp->data[bi/8] |= m; // Mark block in use.
4816                 log_write(bp);
4817                 brelse(bp);
4818                 bzero(dev, b + bi);
4819                 return b + bi;
4820             }
4821         }
4822         brelse(bp);
4823     }
4824     panic("balloc: out of blocks");
4825 }

```

balloc()

- Check every bit (**bi**) of a block
 - BPB – bits per block

```
4802 // Allocate a zeroed disk block.
4803 static uint
4804 balloc(uint dev)
4805 {
4806     int b, bi, m;
4807     struct buf *bp;
4808
4809     bp = 0;
4810     for(b = 0; b < sb.size; b += BPB){
4811         bp = bread(dev, BBLOCK(b, sb));
4812         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4813             m = 1 << (bi % 8);
4814             if((bp->data[bi/8] & m) == 0){ // Is block free?
4815                 bp->data[bi/8] |= m; // Mark block in use.
4816                 log_write(bp);
4817                 brelse(bp);
4818                 bzero(dev, b + bi);
4819                 return b + bi;
4820             }
4821         }
4822         brelse(bp);
4823     }
4824     panic("balloc: out of blocks");
4825 }
```

balloc()

```

4802 // Allocate a zeroed disk block.
4803 static uint
4804 balloc(uint dev)
4805 {
4806     int b, bi, m;
4807     struct buf *bp;
4808
4809     bp = 0;
4810     for(b = 0; b < sb.size; b += BPB){
4811         bp = bread(dev, BBLOCK(b, sb));
4812         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4813             m = 1 << (bi % 8);
4814             if((bp->data[bi/8] & m) == 0){ // Is block free?
4815                 bp->data[bi/8] |= m; // Mark block in use.
4816                 log_write(bp);
4817                 brelse(bp);
4818                 bzero(dev, b + bi);
4819                 return b + bi;
4820             }
4821         }
4822         brelse(bp);
4823     }
4824     panic("balloc: out of blocks");
4825 }

```

balloc()

- 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

```
3965 struct dirent {  
3966     ushort inum;  
3967     char name[DIRSIZ];  
3968 };
```

```
5360 struct inode*
5361 dirlookup(struct inode *dp, char *name, uint *poff)
5362 {
    ...
5366     if(dp->type != T_DIR)
5367         panic("dirlookup not DIR");
5368
5369     for(off = 0; off < dp->size; off += sizeof(de)){
5370         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5371             panic("dirlink read");
5372         if(de.inum == 0)
5373             continue;
5374         if(namecmp(name, de.name) == 0){
5375             // entry matches path element
5376             if(poff)
5377                 *poff = off;
5378             inum = de.inum;
5379             return iget(dp->dev, inum);
5380         }
5381     }
5382
5383     return 0;
5384 }
```

dirlookup()

- Inode is a directory

```
5360 struct inode*
5361 dirlookup(struct inode *dp, char *name, uint *poff)
5362 {
    ...
5366     if(dp->type != T_DIR)
5367         panic("dirlookup not DIR");
5368
5369     for(off = 0; off < dp->size; off += sizeof(de)){
5370         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5371             panic("dirlink read");
5372         if(de.inum == 0)
5373             continue;
5374         if(namecmp(name, de.name) == 0){
5375             // entry matches path element
5376             if(poff)
5377                 *poff = off;
5378             inum = de.inum;
5379             return iget(dp->dev, inum);
5380         }
5381     }
5382
5383     return 0;
5384 }
```

dirlookup()

- Iterate through all entries?


```
5360 struct inode*
5361 dirlookup(struct inode *dp, char *name, uint *poff)
5362 {
    ...
5366     if(dp->type != T_DIR)
5367         panic("dirlookup not DIR");
5368
5369     for(off = 0; off < dp->size; off += sizeof(de)){
5370         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5371             panic("dirlink read");
5372         if(de.inum == 0)
5373             continue;
5374         if(namecmp(name, de.name) == 0){
5375             // entry matches path element
5376             if(poff)
5377                 *poff = off;
5378             inum = de.inum;
5379             return iget(dp->dev, inum);
5380         }
5381     }
5382
5383     return 0;
5384 }
```

dirlookup()

- Read the inode
- Compare names

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()

```
5539 struct inode*
5540 namei(char *path)
5541 {
5542     char name[DIRSIZ];
5543     return namex(path, 0, name);
5544 }
```

```

5505 namex(char *path, int nameiparent, char *name)
5506 {
...
5509     if(*path == '/')
5510         ip = iget(ROOTDEV, ROOTINO);
5511     else
5512         ip = idup(proc->cwd);
5513     // skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5514     while((path = skipelem(path, name)) != 0){
5515         ilock(ip);
5516         if(ip->type != T_DIR){
5517             iunlockput(ip);
5518             return 0;
5519         }
...
5525         if((next = dirlookup(ip, name, 0)) == 0){
5526             iunlockput(ip);
5527             return 0;
5528         }
5529         iunlockput(ip);
5530         ip = next;
5531     }
5532     if(nameiparent){
5533         iput(ip);
5534         return 0;
5535     }
5536     return ip;
5537 }

```

namex()

- If path == "/" start with the inode number of the root

```

5505 namex(char *path, int nameiparent, char *name)
5506 {
...
5509     if(*path == '/')
5510         ip = iget(ROOTDEV, ROOTINO);
5511     else
5512         ip = idup(proc->cwd);
5513     // skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5514     while((path = skipelem(path, name)) != 0){
5515         ilock(ip);
5516         if(ip->type != T_DIR){
5517             iunlockput(ip);
5518             return 0;
5519         }
...
5525         if((next = dirlookup(ip, name, 0)) == 0){
5526             iunlockput(ip);
5527             return 0;
5528         }
5529         iunlockput(ip);
5530         ip = next;
5531     }
5532     if(nameiparent){
5533         iput(ip);
5534         return 0;
5535     }
5536     return ip;
5537 }

```

namex()

- Get the prefix
 - “a/bb/c”
 - name = a
 - path = “bb/c”

```

5505 namex(char *path, int nameiparent, char *name)
5506 {
...
5509     if(*path == '/')
5510         ip = iget(ROOTDEV, ROOTINO);
5511     else
5512         ip = idup(proc->cwd);
5513     // skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5514     while((path = skipelem(path, name)) != 0){
5515         ilock(ip);
5516         if(ip->type != T_DIR){
5517             iunlockput(ip);
5518             return 0;
5519         }
...
5525         if((next = dirlookup(ip, name, 0)) == 0){
5526             iunlockput(ip);
5527             return 0;
5528         }
5529         iunlockput(ip);
5530         ip = next;
5531     }
5532     if(nameiparent){
5533         iput(ip);
5534         return 0;
5535     }
5536     return ip;
5537 }

```

namex()

- Lookup that name in the directory

```
6101 sys_open(void)
6102 {
...
6108     if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6109         return -1;
6110
6111     begin_op();
6112
...
6120     if((ip = namei(path)) == 0){
6121         end_op();
6122         return -1;
6123     }
...
6132     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6133         if(f)
6134             fileclose(f);
6135         iunlockput(ip);
6136         end_op();
6137         return -1;
6138     }
6139     iunlock(ip);
6140     end_op();
6141
6142     f->type = FD_INODE;
6143     f->ip = ip;
...
6147     return fd;
6148 }
```

Example: sys_open

File descriptor layer

File descriptors

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

```
4000 struct file {
4001     enum { FD_NONE, FD_PIPE, FD_INODE } type;
4002     int ref; // reference count
4003     char readable;
4004     char writable;
4005     struct pipe *pipe;
4006     struct inode *ip;
4007     uint off;
4008 };
```

Example: sys_open

```
6101 sys_open(void)
6102 {
...
6108     if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6109         return -1;
6110
6111     begin_op();
6112
...
6120     if((ip = namei(path)) == 0){
6121         end_op();
6122         return -1;
6123     }
...
6132     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6133         if(f)
6134             fileclose(f);
6135         iunlockput(ip);
6136         end_op();
6137         return -1;
6138     }
6139     iunlock(ip);
6140     end_op();
6141
6142     f->type = FD_INODE;
6143     f->ip = ip;
...
6147     return fd;
6148 }
```

- Allocate new file data structure

Files and filealloc()

```
5612 struct {
5613     struct spinlock lock;
5614     struct file file[NFILE];
5615 } ftable;
...
5624 struct file*
5625 filealloc(void)
5626 {
5627     struct file *f;
5628
5629     acquire(&ftable.lock);
5630     for(f = ftable.file; f < ftable.file + NFILE; f++){
5631         if(f->ref == 0){
5632             f->ref = 1;
5633             release(&ftable.lock);
5634             return f;
5635         }
5636     }
5637     release(&ftable.lock);
5638     return 0;
5639 }
```

- Linear search for an available element of the ftable array
 - f->ref == 0

Example: sys_open

```
6101 sys_open(void)
6102 {
...
6108     if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6109         return -1;
6110
6111     begin_op();
6112
...
6120     if((ip = namei(path)) == 0){
6121         end_op();
6122         return -1;
6123     }
...
6132     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0) [
6133         if(f)
6134             fileclose(f);
6135         iunlockput(ip);
6136         end_op();
6137         return -1;
6138     }
6139     iunlock(ip);
6140     end_op();
6141
6142     f->type = FD_INODE;
6143     f->ip = ip;
...
6147     return fd;
6148 }
```

- Allocate a new file descriptor

```
5835 // Allocate a file descriptor for the given file.  
5836 // Takes over file reference from caller on  
success.
```

```
5837 static int  
5838 fdalloc(struct file *f)  
5839 {  
5840     int fd;  
5841  
5842     for(fd = 0; fd < NOFILE; fd++){  
5843         if(proc->ofile[fd] == 0){  
5844             proc->ofile[fd] = f;  
5845             return fd;  
5846         }  
5847     }  
5848     return -1;  
5849 }
```

File descriptors and fdalloc()

- Allocate a file descriptor

Thank you!

Example: write system call

Write() syscall

```
5476 int
5477 sys_write(void)
5478 {
5479     struct file *f;
5480     int n;
5481     char *p;
5482
5483     if(argfd(0, 0, &f) < 0
        || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5484         return -1;
5485     return filewrite(f, p, n);
5486 }
```



```
5352 fwrite(struct file *f, char *addr, int n)
5353 {
5360     if(f->type == FD_INODE){
...
5368         int i = 0;
5369         while(i < n){
...
5373
5374             begin_trans();
5375             ilock(f->ip);
5376             if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5377                 f->off += r;
5378             iunlock(f->ip);
5379             commit_trans();
5386     }
5390 }
```

**Write several
blocks at a time**