cs5460/6460: Operating Systems

Lecture: File systems

Anton Burtsev

April, 2024

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

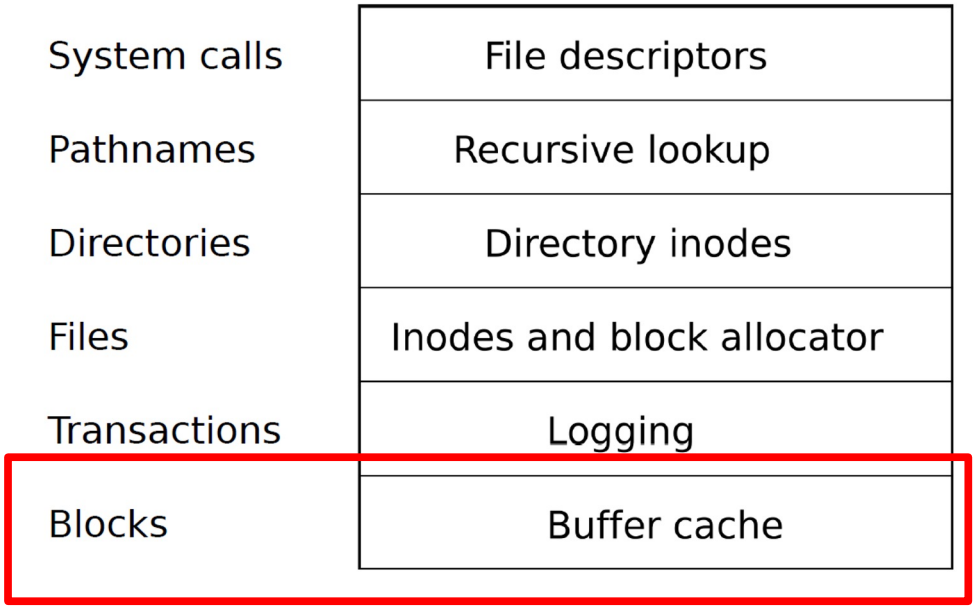
Crash recovery

- 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

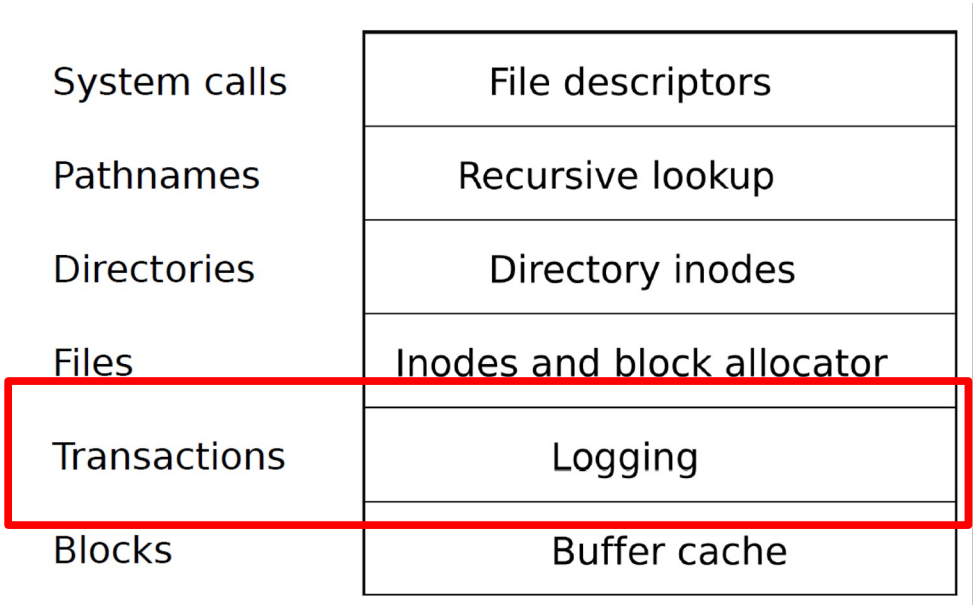
- Access to a block device is several orders of magnitude slower
  - Memory: 200 cycles
  - Disk: 20 000 000 cycles
- A file system must maintain a cache of disk blocks in memory

Block layer



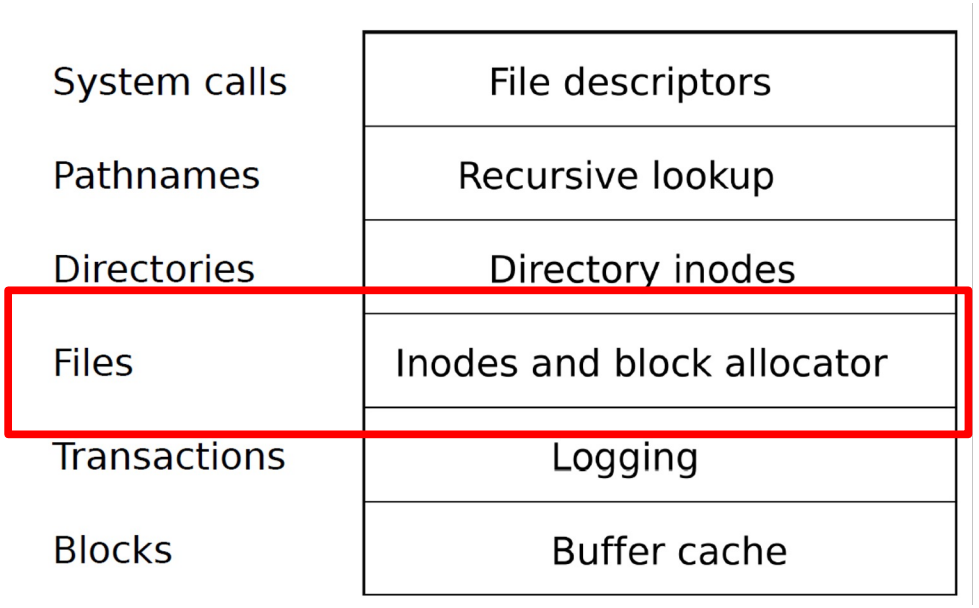
- Read and write data
- From a block device
- Into a buffer cache
- Synchronize across multiple readers and writers

Transactions



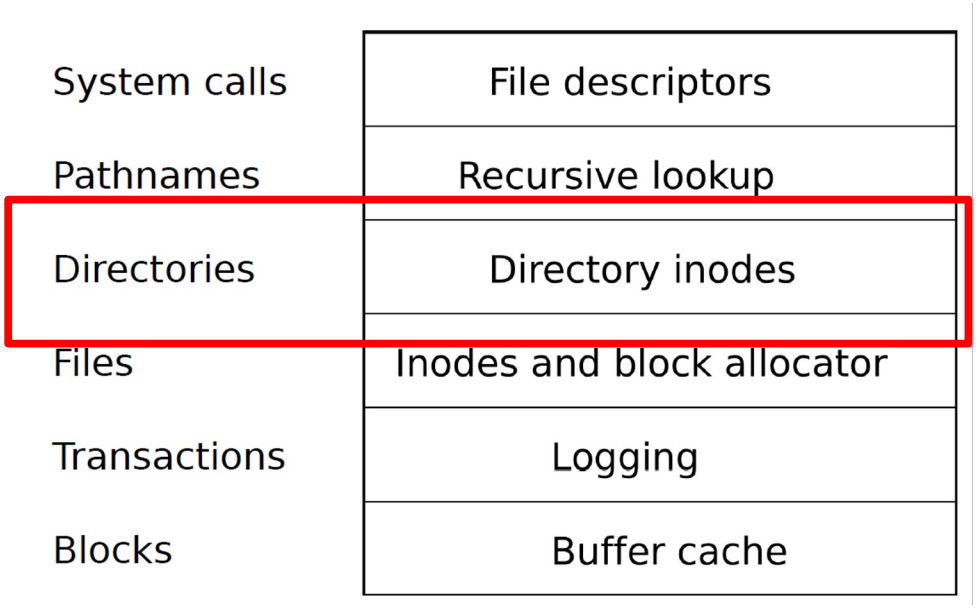
- Group multiple writes into an atomic transaction

Files



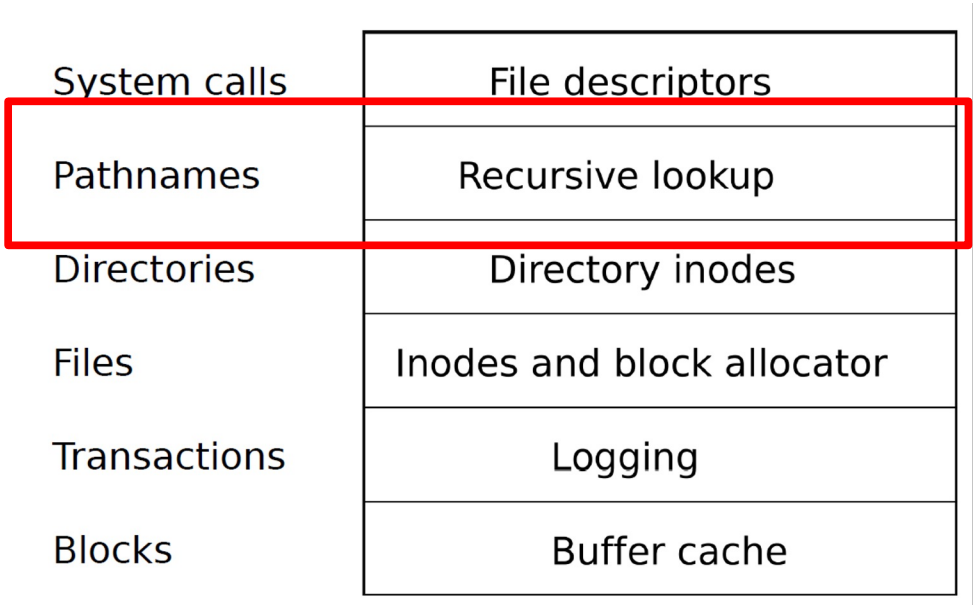
- Unnamed files
- Represented as inodes
- Sequence of blocks holding file's data

Directories



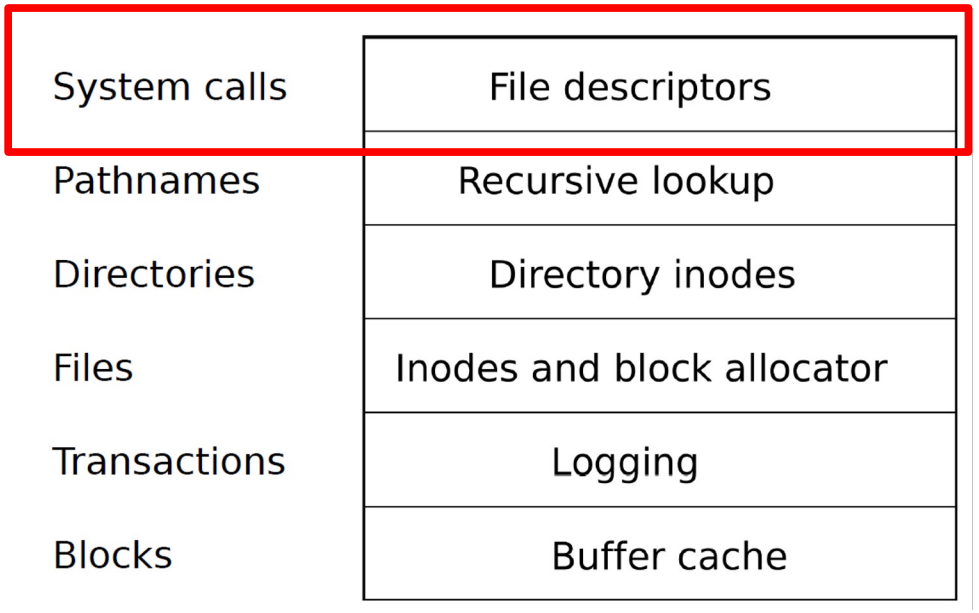
- Special kind of inode
- Sequence of directory entries
- Each contains name and a pointer to an unnamed inode

Pathnames



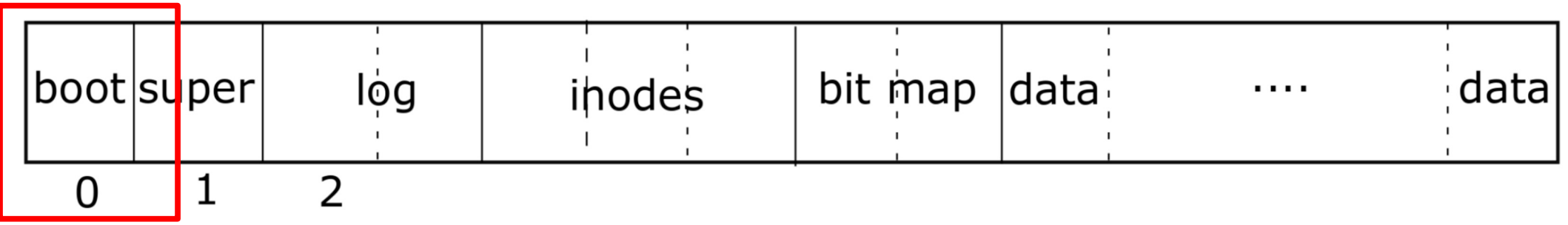
- Hierarchical path names
- /usr/bin/sh
- Recursive lookup

System call



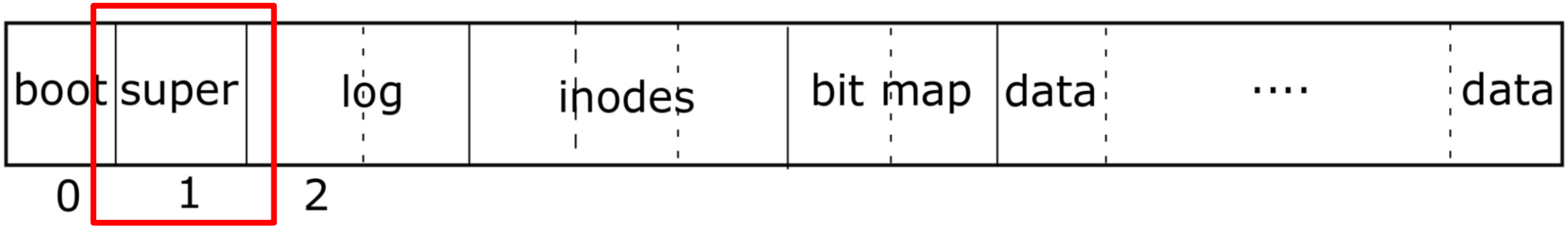
- Abstract UNIX resources as files
- Files, sockets, devices, pipes, etc.
- Unified programming interface

File system layout on disk



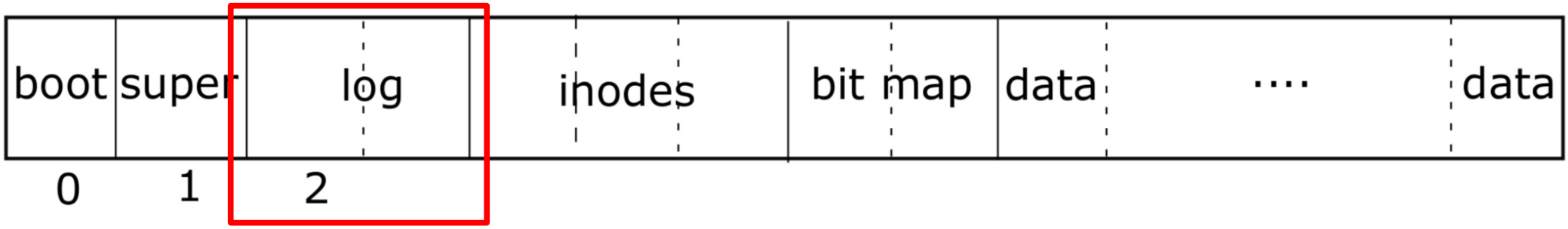
- Block #0: Boot code

File system layout on disk



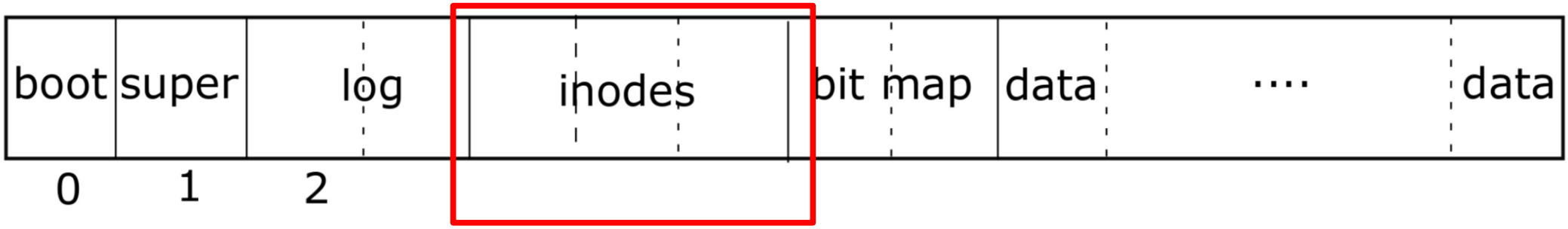
- Block #0: Boot code
- Block #1: (superblock) Metadata about the file system
  - Size (number of blocks)
  - Number of data blocks
  - Number of inodes
  - Number of blocks in log

File system layout on disk



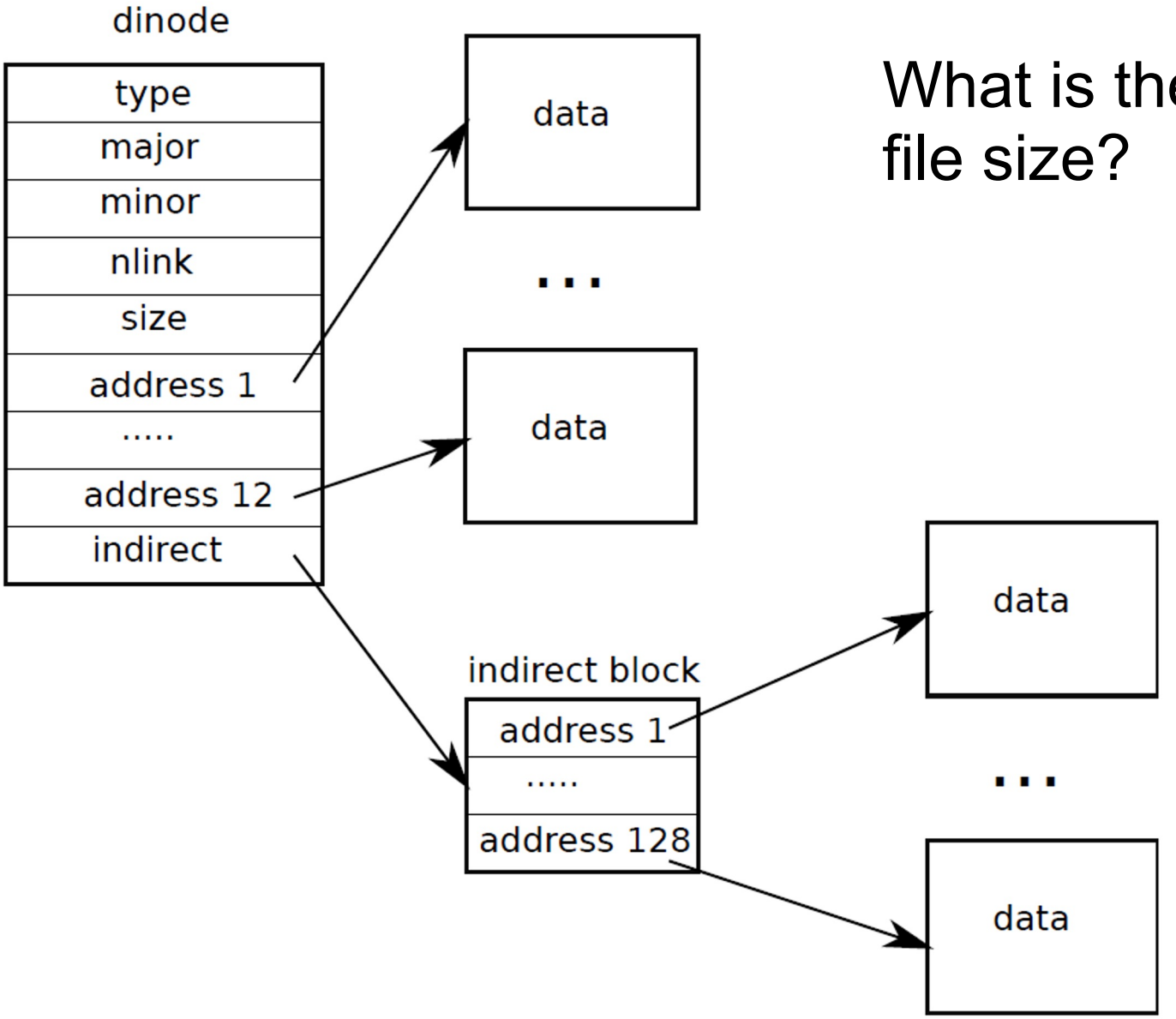
- Block #2: Log area: maintaining consistency in case of a power outage or system crash

File system layout on disk



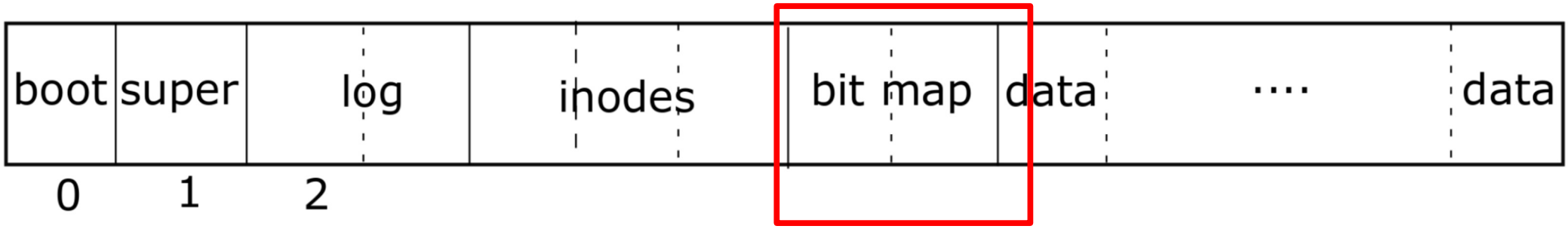
- Inode area
- Unnamed files

Representing files on disk



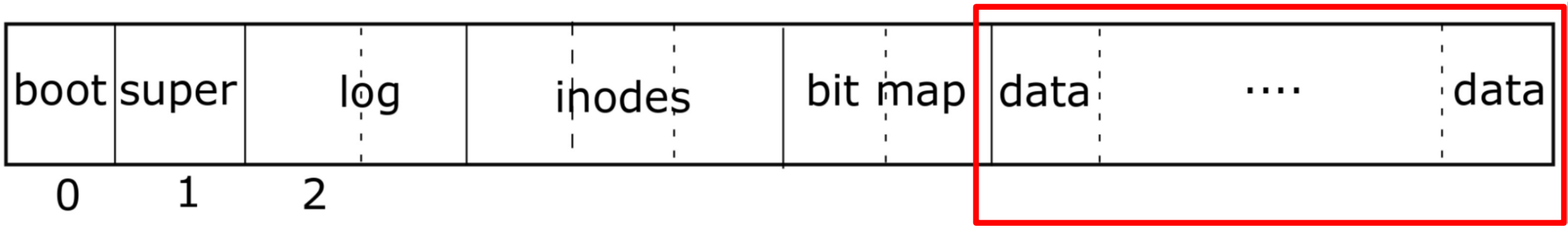
What is the max file size?

File system layout on disk



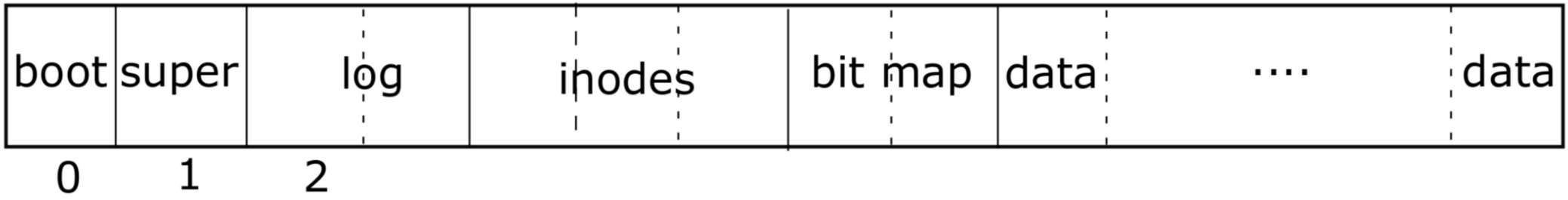
- Bit map area: tracks which blocks are in use

File system layout on disk



- Data area: actual file data

File system layout on disk



- Poll: https://pollev.com/aburtsev
- What's inside the bitmap area?

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
```

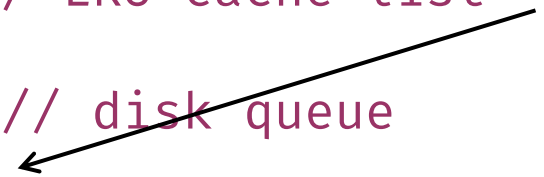
- Array of buffers

```
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

- Cached data
- 512 bytes

```
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

- Flags

```
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

- Device
- We might have multiple disks

```
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

- Block number on disk

```
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

- LRU list
- To evict the oldest blocks

```
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 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()

Common pattern

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

- 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

```
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

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. you can launch the missile twice since you lost the write saying you already did)

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

Typical use of transactions

Log (in memory)

```
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 };
```

```
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
  - Not enough space for a new transaction
  - 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 #3
  - All good
  - Reserve space for a new transaction

Typical use of transactions

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

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

log_write

```
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 }
```

- Check if already in log

log_write

```
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 }
```

- Add to the log
- Prevent eviction

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

Typical use of transactions

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()

```
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 }
```

commit()

```
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 }
```

write_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,
                            log.start+tail+1); // log block
4689     struct buf *from = bread(log.dev,
                            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 }
```

- Loop through the entire log

write_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,
                            log.start+tail+1); // log block
4689     struct buf *from = bread(log.dev,
                            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 }
```

- Get a lock on the block of the log

write_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,
                              log.start+tail+1); // log block
4689     struct buf *from = bread(log.dev,
                              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 }
```

- Read the actual block
- It's in the buffer cache

write_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,
                            log.start+tail+1); // log block
4689     struct buf *from = bread(log.dev,
                            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 }
```

- Copy data between the blocks

write_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,
                            log.start+tail+1); // log block
4689     struct buf *from = bread(log.dev,
                            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 the “to” and release all locks

commit()

```
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 }
```

write_head()

```
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 }
```

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

write_head()

```
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 }
```

- Typecast the buf->data to the logheader

write_head()

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

write_head()

```
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 all block numbers

write_head()

```
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 the logheader back to disk into the log area
- Release the lock

commit()

```
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 }
```

install_trans()

```
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 }
```

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

install_trans()

```
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 }
```

- Read the block from the data area where the data should go

install_trans()

```
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 }
```

- Copy data between the blocks

install_trans()

```
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 }
```

- Write the block back to disk
- Release locks

commit()

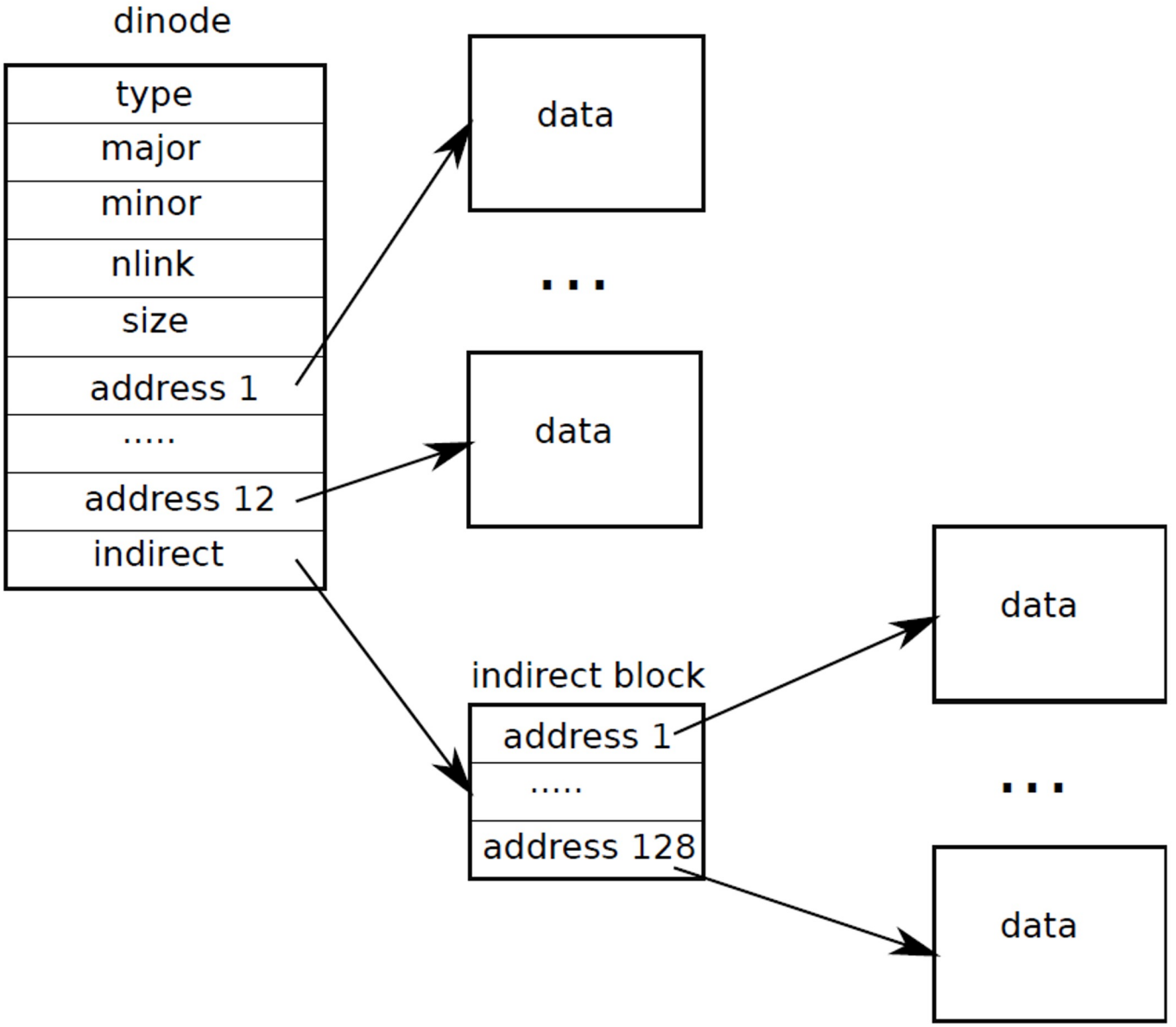
```
4701 commit()
4702 {
4703   if (log.lh.n > 0) {
4704     write_log(); // Write modified blocks
                        from cache to log
4705     write_head(); // Write header to disk --
                        the real commit
4706     install_trans(); // Now install writes
                     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

Inode layer

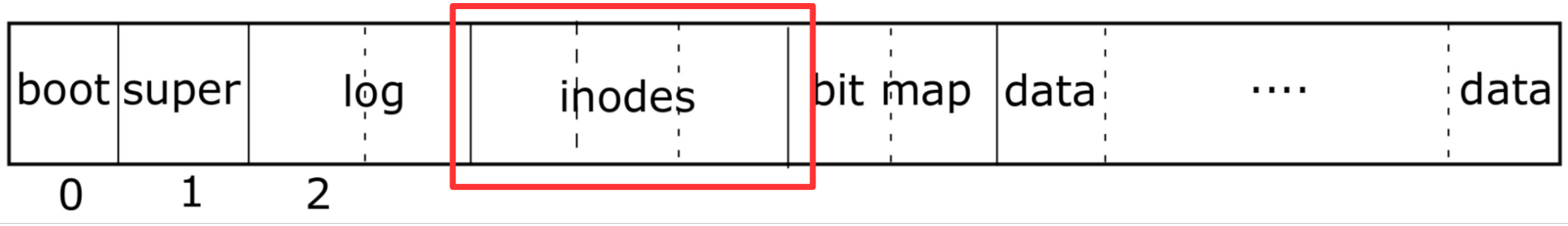
Inode

- 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

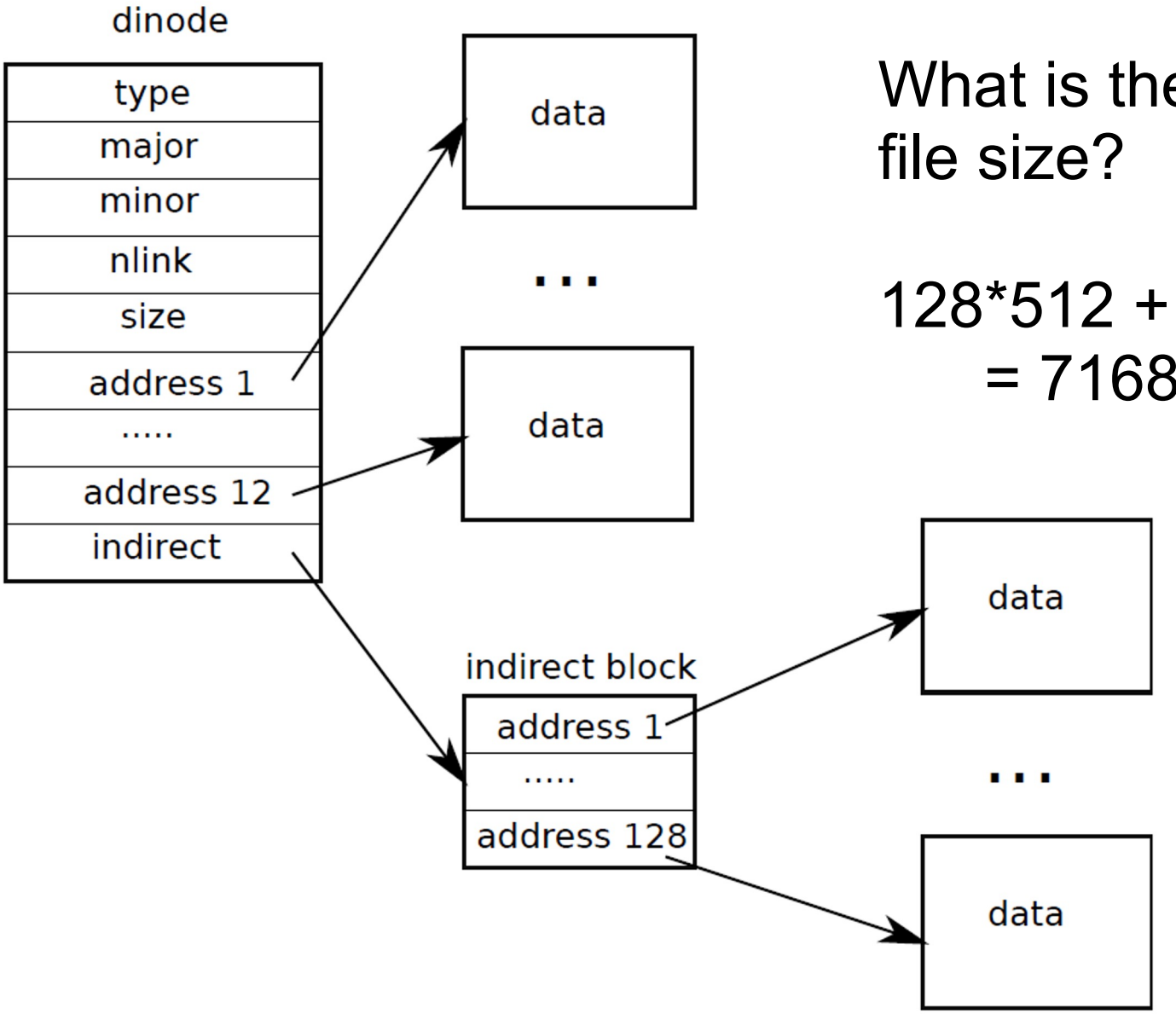


File system layout on disk



- 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

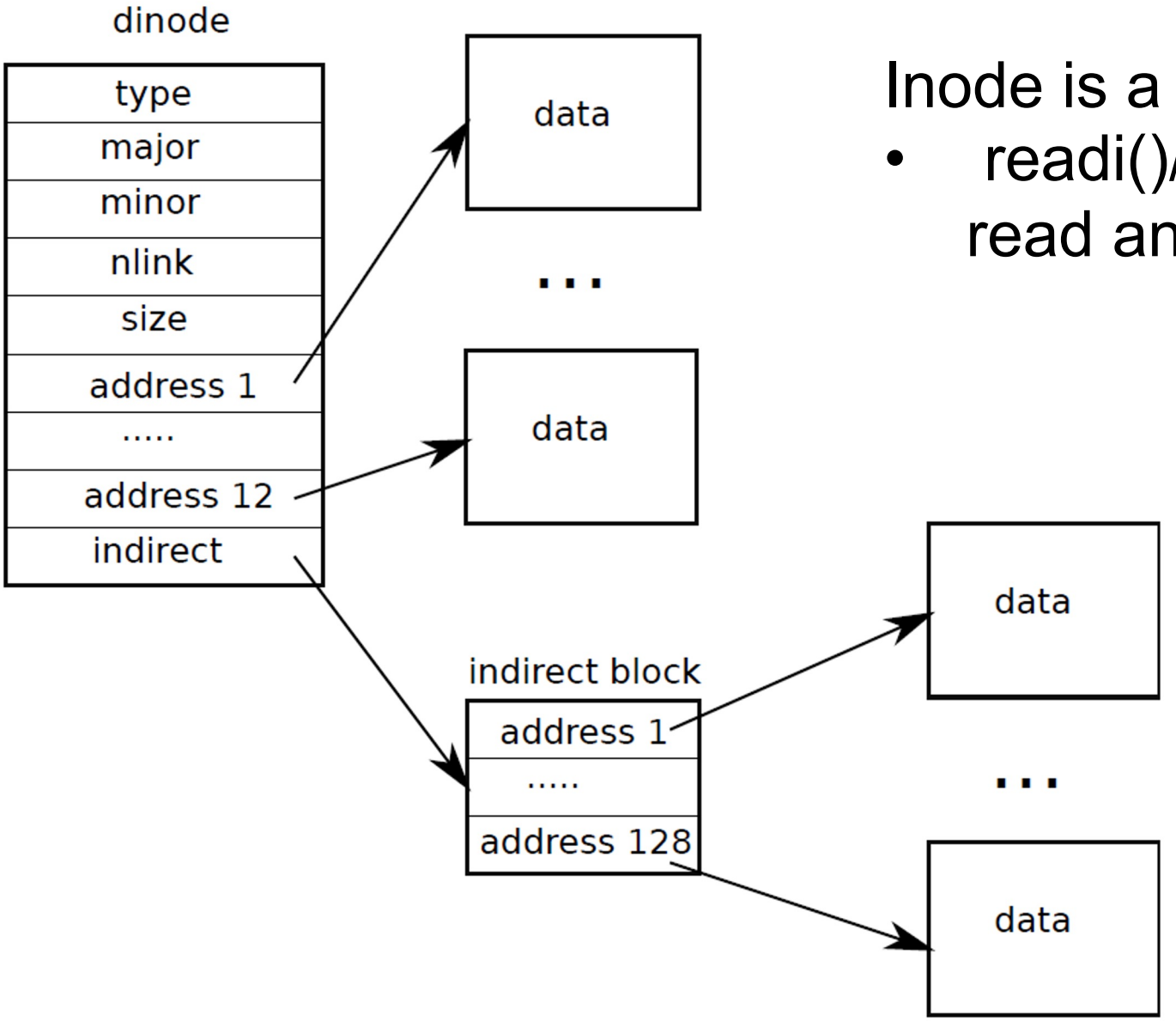
Representing files on disk



What is the max file size?

128*512 + 12*512
= 71680

Reading and writing inodes



Inode is a file

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

Example: sys_read()

```
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 }
```

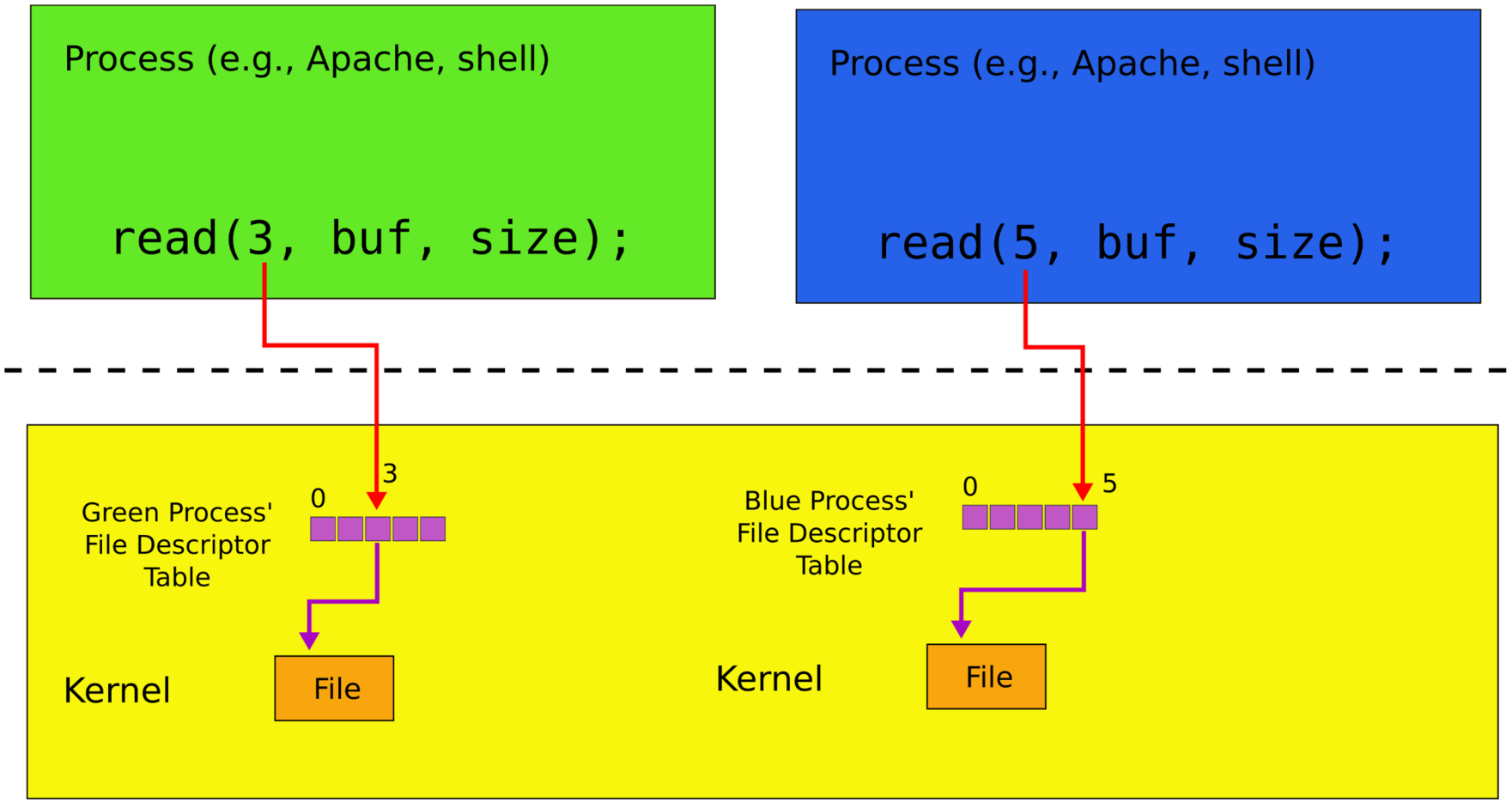
- Question:
- Where does f come from?

```
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[]

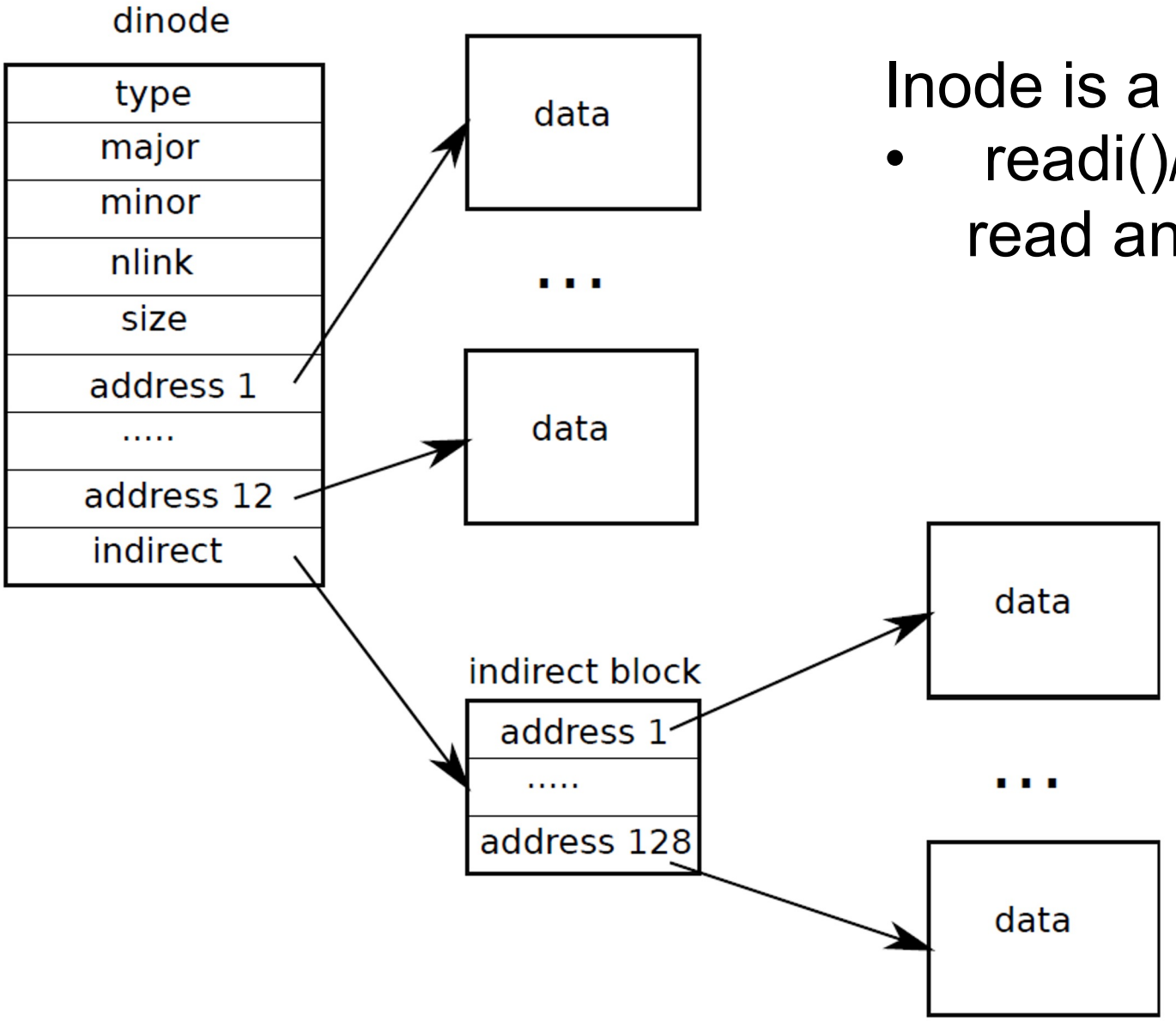
File descriptors: two processes



```
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”

Reading and writing inodes

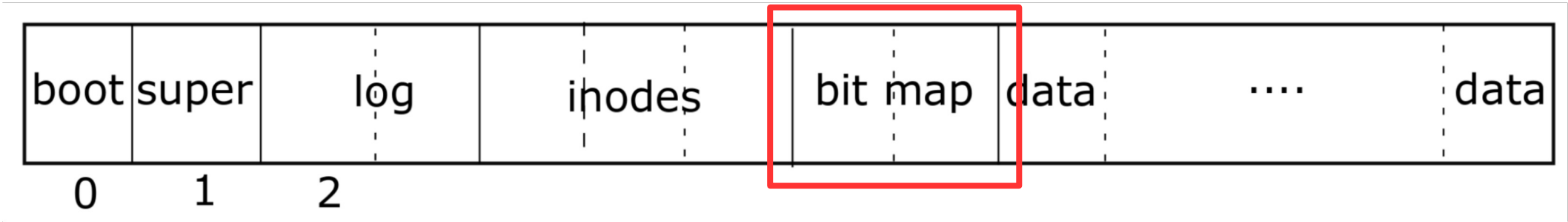


Inode is a file

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

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

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 };
```

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

```
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 }
```

Eaxmple:
sys_open

File descriptor layer

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 filewrite(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