ICS143A: Principles of Operating Systems

Lecture 18: File systems

Anton Burtsev November, 2017

The role of file systems

The role of file systems

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

Architecture

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

Crash recovery

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

Multiple users

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

Speed

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

Block layer

File descriptors
Recursive lookup
Directory inodes
Inodes and block allocator
Logging
Buffer cache

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

Transactions

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

Group multiple writes into an atomic transaction

Files

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

Unnamed files

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

Directories

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

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

Pathnames

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

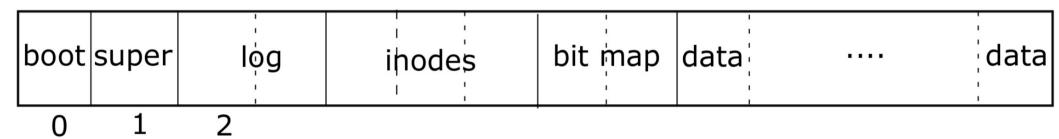
- Hierarchical path names
 - /usr/bin/sh
 - Recursive lookup

System call

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

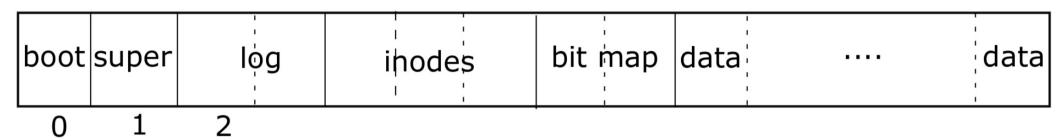
- Abstract UNIX resources as files
 - Files, sockets, devices, pipes, etc.
- Unified programming interface

File system layout on disk



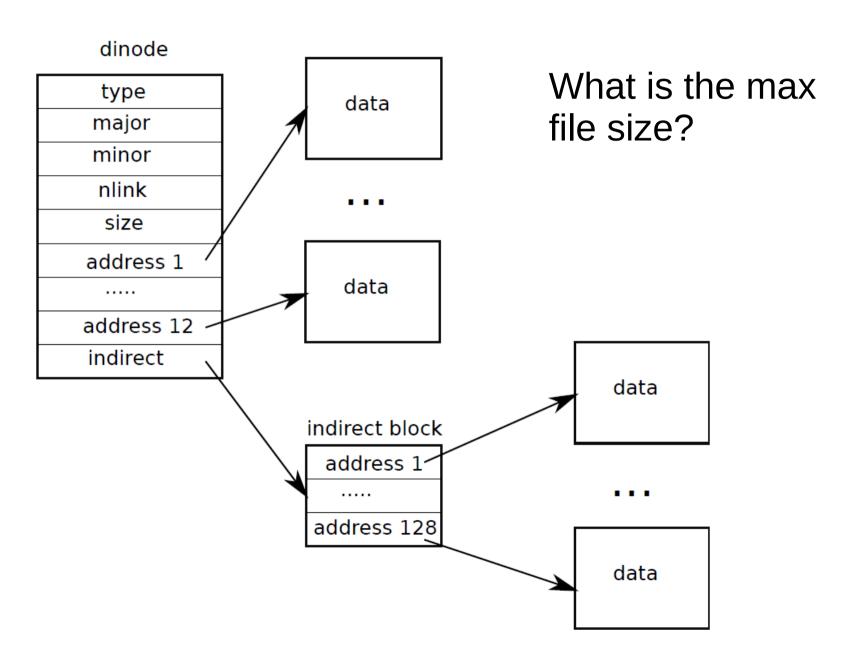
- Block #0: Boot code
- Block #1: (superblock) Metadata about the file system
 - Size (number of blocks)
 - Number of data blocks
 - Number of inodes
 - Number of blocks in log

File system layout on disk

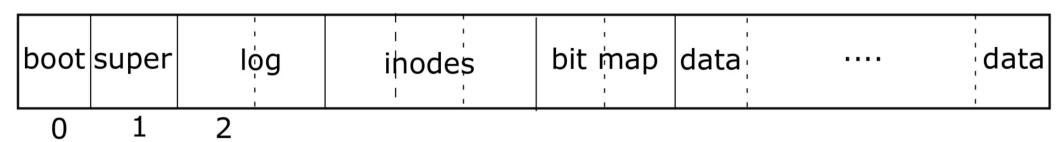


- Block #2: Log area: maintaining consistency in case of a power outage or system crash
- Inode area
 - Unnamed files

Representing files on disk



File system layout on disk



- Block #2: Log area: maintaining consistency in case of a power outage or system crash
- Inode area
 - Unnamed files
- Bit map area: track which blocks are in use
- Data area: actual file data

Buffer cache layer

Buffer cache layer

Two goals:

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

```
3750 struct buf {
                                                          Flags
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 gaeue
     uchar data[BSIZE];
3757
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;
```

```
3750 struct buf {
                                                         Device
3751
      int flags;
3752
     uint dev;

    We might have

3753 uint blockno;
                                                             multiple disks
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;
```

```
3750 struct buf {
                                                         Block number on disk
3751
      int flags;
3752
     uint dev;
     uint blockno:
3753
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;
```

```
3750 struct buf {
                                                          LRU list
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 looked 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 {
      struct spinlock lock;
4330
      struct buf buf[NBUF];
4331
4332
4333
     // Linked list of all buffers, through prev/next.
4334
      // head.next is most recently used.
4335
      struct buf head;
4336 } bcache;
```

```
3750 struct buf {
                                                         Cached data
3751
      int flags;
3752
     uint dev;
                                                          • 512 bytes
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()

```
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

```
4366 bget(uint dev, uint sector)
4367 {
4368
     struct buf *b;
4370
       acquire(&bcache.lock);
4372 loop:
4373
     // Is the sector already cached?
for(b = bcache.head.next; b != &bcache.head; b = b->next){
4375
         if(b->dev == dev && b->sector == sector){
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
      }
4385
. . .
4399 }
```

Getting a block from a buffer cache (part 1)

```
4466 bget(uint dev, uint sector)
4467 {
4468
    struct buf *b;
4470 acquire(&bcache.lock);
4472 loop:
. . .
4485
4486 // Not cached; recycle some non-busy and clean buffer.
4487
      for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
        if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4488
4489
          b->dev = dev:
4490
          b->sector = sector;
4491
         b->flags = B BUSY;
4492
          release(&bcache.lock);
4493
          return b:
                                         Getting a block
4494
4495
                                            from a buffer
4496
      panic("bget: no buffers");
4497 }
                                           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 {
       if((b\rightarrow flags \& B BUSY) == 0)
4428
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()
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++) {</pre>
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. file delete can crash leaving:
 - Directory entry pointing to a free inode
 - Allocated but unlinked inode

Logging

- Writes don't directly go to disk
 - Instead they are logged in a journal
 - Once all writes are logged, the system writes a special commit record
 - Indicating that log contains a complete operation
- At this point file system copies writes to the ondisk 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();
```

```
4532 struct logheader {
4533 int n;
                                 Log (in memory)
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();
```

```
4626 // called at the start of each FS system call.
4627 void
4628 begin op(void)
4629 {
                                            begin op()
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 {

    Case #1

          log.outstanding += 1;
4638
4639
          release(&log.lock);
                                           Log is being
4640
          break;
                                            committed
4641
4642 }

    Sleep

4643 }
```

```
4626 // called at the start of each FS system call.
4627 void
4628 begin op(void)
4629 {
                                           begin op()
      acquire(&log.lock);
4630
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 {

    Case #2

          log.outstanding += 1;
4638
4639
          release(&log.lock);
                                          Log doesn't have
4640
          break;
                                          enough space for the
4641
                                           new transaction
4642 }
4643 }
```

```
4626 // called at the start of each FS system call.
4627 void
4628 begin op(void)
4629 {
                                            begin op()
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 {

    Case #3

          log.outstanding += 1;
4638
4639
          release(&log.lock);

    All ok, reserve space

4640
          break;
                                           in the log for the new
4641
                                           transaction
4642 }
4643 }
```

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

log_write() replacesbwrite(); brelse()

```
4722 log_write(struct buf *b)
                                              log write
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
       }

    Check if already

4736
       log.lh.block[i] = b->blockno;
4737
       if (i == log.lh.n)
                                                 in log
4738
         log.lh.n++;
       b->flags |= B_DIRTY; // prevent eviction
4739
       release(&log.lock);
4740
4741 }
```

```
4722 log_write(struct buf *b)
                                              log write
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);
       for (i = 0; i < log.lh.n; i++) {
4732
4733
         if (log.lh.block[i] == b->blockno) // log absorbtion
4734
           break:
4735
      }

    Add to the log

4736
       log.lh.block[i] = b->blockno;
4737
       if (i == log.lh.n)

    Prevent eviction

4738
         log.lh.n++;
4739
       b->flags |= B DIRTY; // prevent eviction
4740
      release(&log.lock);
4741 }
```

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

```
4653 end op(void)
4654 {
                                                          end op()
      int do commit = 0;
4655
4656
4657
      acquire(&log.lock);
4658
      log.outstanding -= 1;
      if(log.outstanding == 0){
4661
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);
        log.committing = 0;
4675
        wakeup(&log);
4676
        release(&log.lock);
4677
4678
     }
4679 }
```

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

```
4701 commit()
                                  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;
         write_head(); // Erase the transaction
4708
                          from the log
4709 }
4710 }
```

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

    Loop through the entire log

4693
        brelse(to);
4694 }
4695 }
```

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

    Read the log block

4693
        brelse(to);
4694 }

    Log goes to

4695 }
                              log.start+tail+1
```

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

    Read the actual block

4693
         brelse(to);
4694 }

    It's in the buffer cache

4695 }

    Block number is in

                              log.lh.block[tail]
```

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

    Copy block data into the log

4693
        brelse(to);
4694 }
4695 }
```

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

    Write the log block (to)

        brelse(to);
4693
4694 }

    Release both blocks

4695 }
```

```
4701 commit()
                                  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;
         write_head(); // Erase the transaction
4708
                          from the log
4709 }
4710 }
```

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

    Read the log header block

      bwrite(buf);
4613
4614 brelse(buf);
                          • It's in log.start
4615 }
```

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

    Interpret buf->data as log

      bwrite(buf);
4613
                         header
4614 brelse(buf);
4615 }

    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
                                    write head()
4604 write_head(void)
4605 {
4606 struct buf *buf = bread(log.dev, log.start);
      struct logheader *hb = (struct logheader *)
4607
                                         (buf->data);
      int i;
4608
4609
      hb->n = log.lh.n;
4610
      for (i = 0; i < log.lh.n; i++) {
4611
        hb->block[i] = log.lh.block[i];
4612

    Write log size (log.lh.n)

4613
      bwrite(buf);
                         into block of the logheader
      brelse(buf);
4614
4615 }
```

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

    Write the entire log

      bwrite(buf);
4613
                         (numbers of blocks in the
4614 brelse(buf);
                         log) into log header
4615 }
```

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

    Write block to disk

      bwrite(buf);
4613

    Release

      brelse(buf);
4614
4615 }
```

```
4701 commit()
                                  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
         install_trans(); // Now install writes
4706
                             to home locations
4707
         log.lh.n = 0;
         write_head(); // Erase the transaction
4708
                          from the log
4709 }
4710 }
```

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

    Read the block from the log

4582
        brelse(dbuf);
                           area (log.start+tail+1)
4583
      }
4584 }
```

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

    Read the block where data

4582
        brelse(dbuf);
                           should go on disk
4583
      }
4584 }
                            • 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)
                                       install trans()
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
        memmove(dbuf->data, lbuf->data, BSIZE); // copy block
4579
                                             // to dst
4580
         bwrite(dbuf); // write dst to disk
4581
        brelse(lbuf);

    Copy data

4582
        brelse(dbuf);
4583 }
4584 }
```

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

    Write the block to disk

4582
         brelse(dbuf);
4583
      }

    Release both blocks

4584 }
```

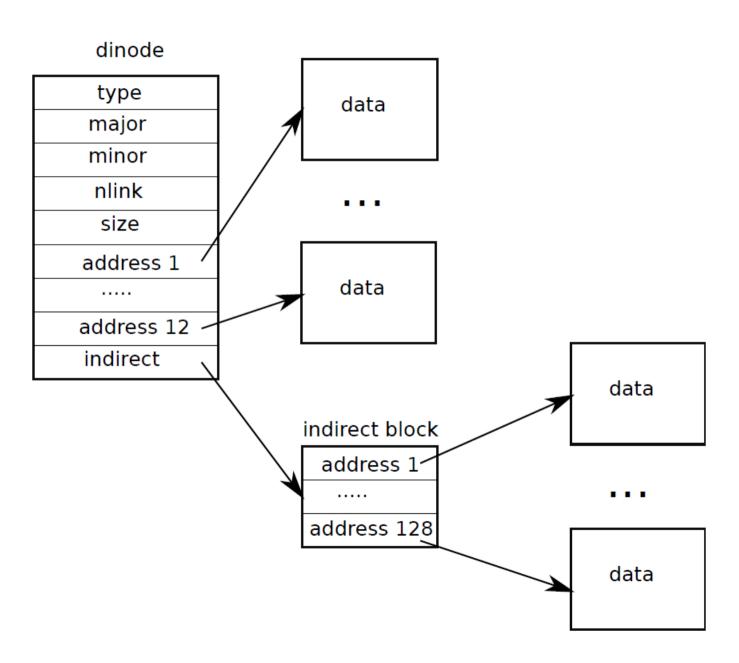
```
4701 commit()
                                  commit()
4702 {
4703 if (log.lh.n > 0) {
4704
         write_log(); // Write modified blocks
                          from cache to log
4705
         write head(); // Write header to disk --
                          the real commit
4706
         install_trans();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

Inode layer

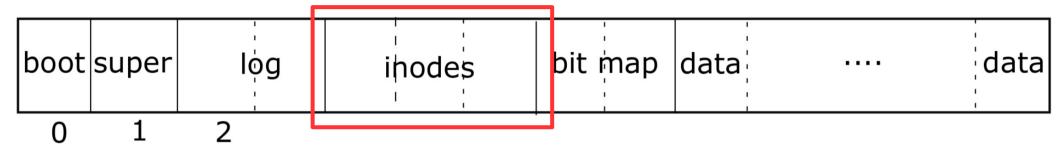
Inode

- 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

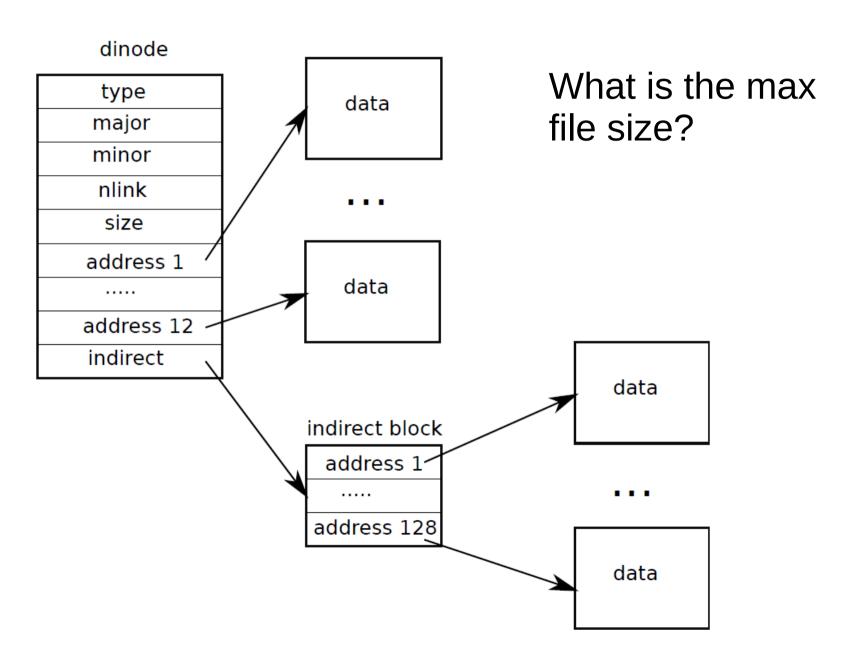


- Inodes are stored as an array on disk
 - sb.startinode
- Each inode has a number (indicating its position on disk)
- The kernel keeps a cache of inodes in memory
 - Synchronization

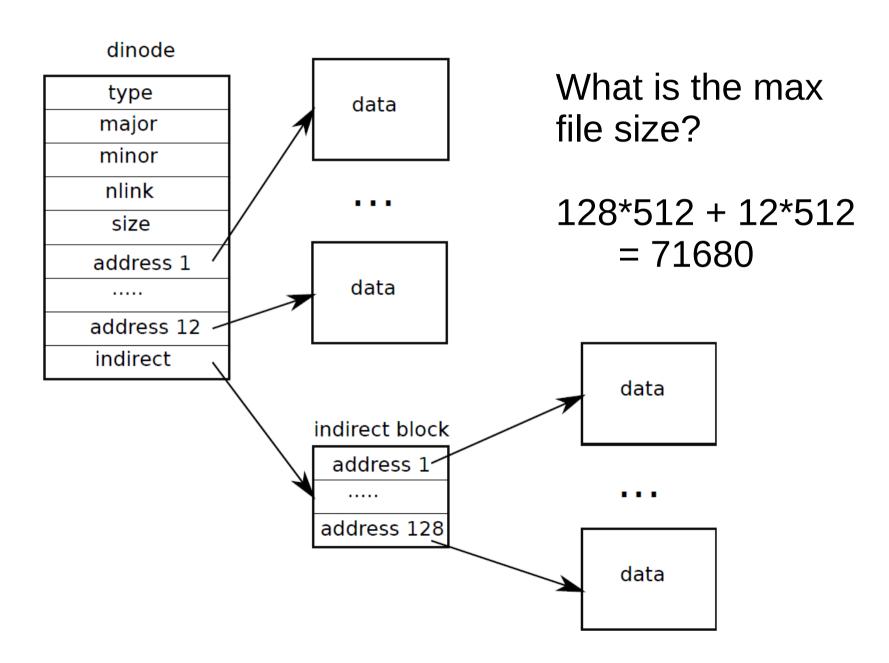
Inode on disk

```
3927 // On-disk inode structure
3928 struct dinode {
      short type; // File type
3929
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



Inode in memory

```
4011 // in-memory copy of an inode
4012 struct inode {
4013
      uint dev; // Device number
4014
      uint inum; // Inode number
       int ref; // Reference count
4015
4016
       int flags; // I_BUSY, I_VALID
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)

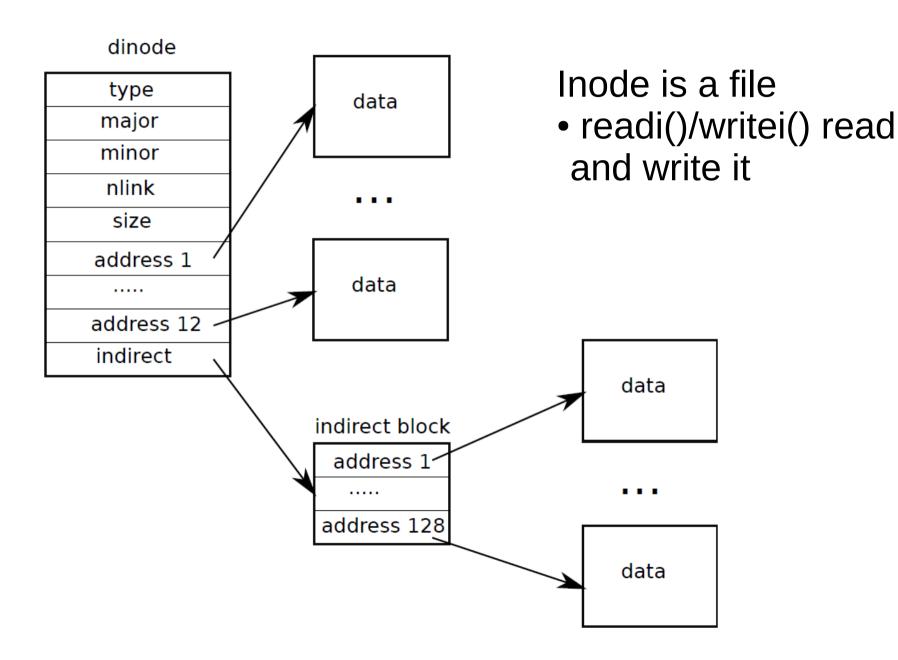
4897 // ... examine and modify ip->xxx ...

4898 // iunlock(ip)

4899 // iput(ip)
```

```
5004 iget(uint dev, uint inum) {
                                                     iget()
. . .
5008
       acquire(&icache.lock);
5010
       // Is the inode already cached?
5011
       emptv = 0;
5012
       for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){</pre>
5013
         if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5014
           ip->ref++;
           release(&icache.lock);
5015
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



```
5864 int
                                                Example:
5865 sys_read(void)
5866 {
                                              sys read()
5867 struct file *f;
5868 int n;
5869
    char *p;
5870
      if(argfd(0, 0, &f) < 0 \mid | argint(2, &n) < 0 \mid | argptr(1, &p, n) < 0)
5871
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 {
                                                  argfd()
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;
      if(pfd)
5828

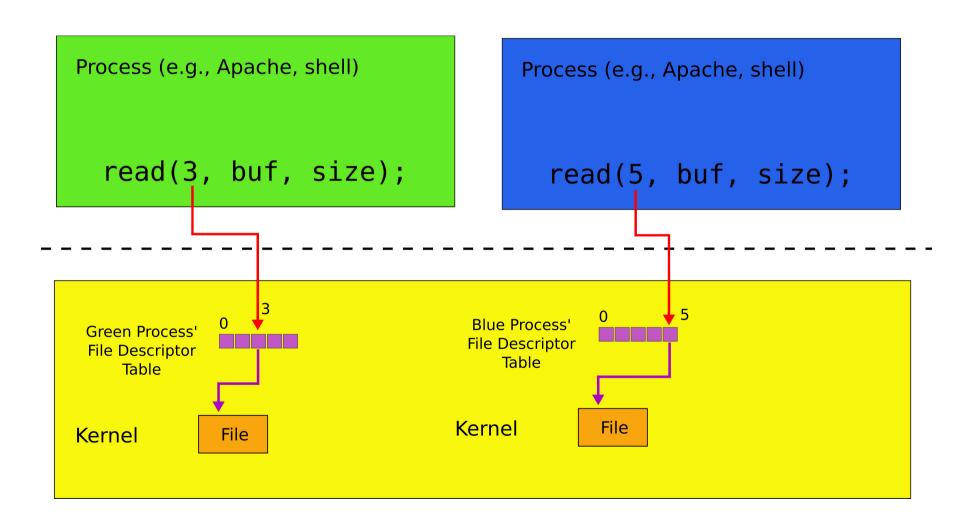
    Remember file descriptors?

        *pfd = fd;
5829
5830
      if(pf)

    Each process has a table

        *pf = f;
5831
5832
      return 0;
                             proc->ofile[]
5833 }
```

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
      struct proc *parent; // Parent process
2359
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
      struct file *ofile[NOFILE]; // Open files
2364
2365 struct inode *cwd; // Current directory
2366 char name[16];
                    // Process name (debugging)
2367 };
```

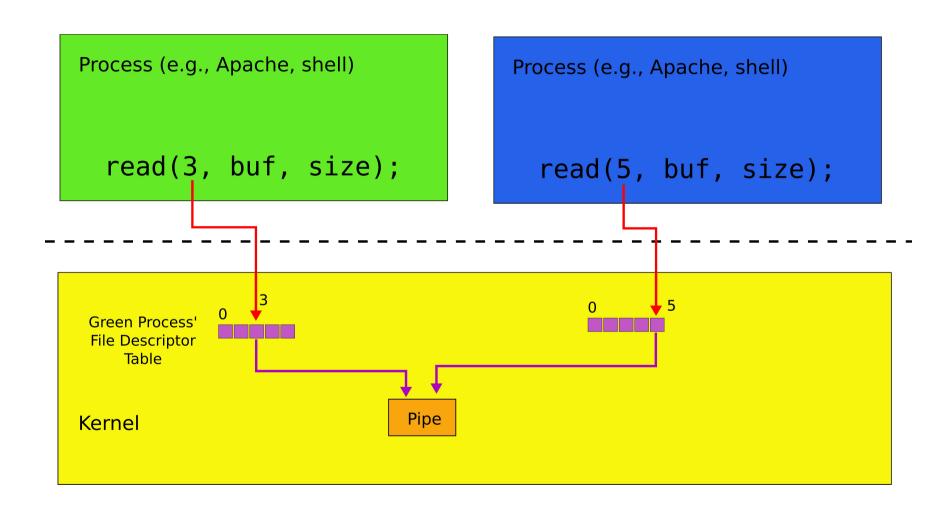
- struct proc has an array of struct file pointers
 - Each element is a "file descriptor"

```
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 {
                                           readi()
5717 int r;
5718
if(f->readable == 0)
        return -1;
5720
      if(f->type == FD PIPE)
5721
        return piperead(f->pipe, addr, n);
5722
5723
      if(f->type == FD_INODE){
5724
        ilock(f->ip);
        if((r = readi(f->ip, addr, f->off, n)) > 0)
5725
5726
          f \rightarrow off += r;
        iunlock(f->ip);
5727
5728
        return r;
5729 }
5730 panic("fileread");
5731 }
```

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

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

```
5714 int.
5715 fileread(struct file *f, char *addr, int n)
5716 {
                                             readi()
5717
     int r;
5718
if(f->readable == 0)
5720
         return -1;
       if(f->type == FD PIPE)
5721
         return piperead(f->pipe, addr, n);
5722
5723
       if(f->type == FD_INODE){
5724
         ilock(f->ip);
         if((r = readi(f->ip, addr, f->off, n)) > 0)
5725
           f \rightarrow off += r;
5726
         iunlock(f->ip);
5727

    Note

5728
         return r;
5729 }

    Read starts with the

5730 panic("fileread");
                                 current offset (f->off)
5731 }
```

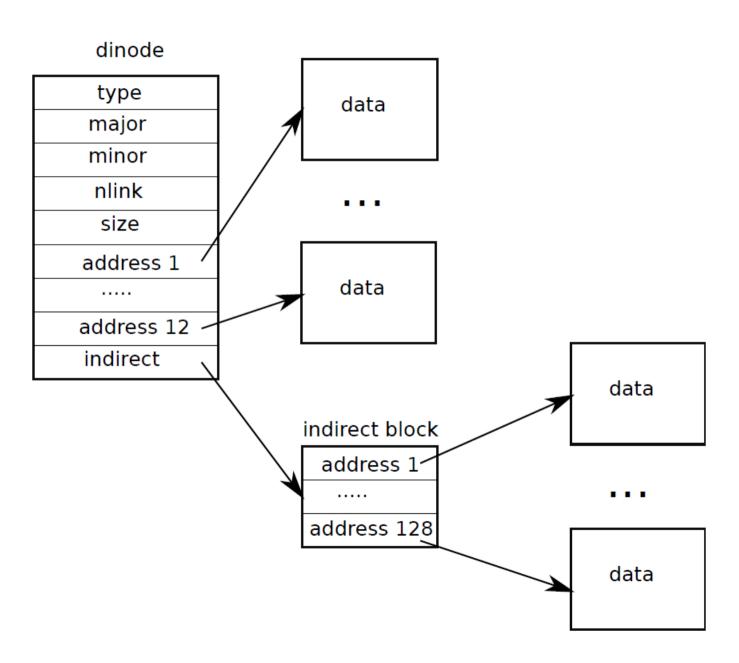
```
5252 readi(struct inode *ip, char *dst, uint off, uint n)
5253 {
5254 uint tot, m;
                                             readi()
5255 struct buf *bp;
5256
. . .
       if(off > ip->size || off + n < off)</pre>
5263
         return -1;
5264
5265
       if(off + n > ip->size)
5266
         n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

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

    What is this bmap()

5273 }
                               function?
5274 return n;
5275 }
```

Representing files on disk



```
5159 static uint
5160 bmap(struct inode *ip, uint bn)
                                                        bmap()
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){</pre>
5173
        // Load indirect block, allocating if necessary.
        if((addr = ip->addrs[NDIRECT]) == 0)
5174
5175
          ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
5177
        if((addr = a[bn]) == 0){
5178
          a[bn] = addr = balloc(ip->dev);
5179
5180
          log write(bp);

    Each inode has some

5181
5182
        brelse(bp);
                                     number (NDIRECT) of
5183
        return addr;
                                     direct pointers
5184 }
```

5187 }

```
5159 static uint
5160 bmap(struct inode *ip, uint bn)
                                                          bmap()
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){</pre>
5173
         // Load indirect block, allocating if necessary.
         if((addr = ip->addrs[NDIRECT]) == 0)
5174
5175
           ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176
         bp = bread(ip->dev, addr);
         a = (uint*)bp->data;
5177
         if((addr = a[bn]) == 0){
5178
           a[bn] = addr = balloc(ip->dev);
5179
5180
           log write(bp);

    No it's beyond NDIRECT

5181
5182
         brelse(bp);
5183
         return addr;
5184
5187 }
```

```
5159 static uint
5160 bmap(struct inode *ip, uint bn)
                                                           bmap()
5161 {
. . .
5165
       if(bn < NDIRECT){</pre>
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){</pre>
5173
         // Load indirect block, allocating if necessary.
         if((addr = ip->addrs[NDIRECT]) == 0)
5174
5175
           ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176
         bp = bread(ip->dev, addr);
         a = (uint*)bp->data;
5177
         if((addr = a[bn]) == 0){
5178
           a[bn] = addr = balloc(ip->dev);
5179
           log write(bp);
5180

    Read an indirect block

5181
5182
         brelse(bp);
5183
         return addr;
5184
```

5187 }

```
5159 static uint
5160 bmap(struct inode *ip, uint bn)
                                                         bmap()
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){</pre>
5173
        // Load indirect block, allocating if necessary.
         if((addr = ip->addrs[NDIRECT]) == 0)
5174
5175
           ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
5177
         if((addr = a[bn]) == 0){
5178
          a[bn] = addr = balloc(ip->dev);
5179
          log write(bp);
5180

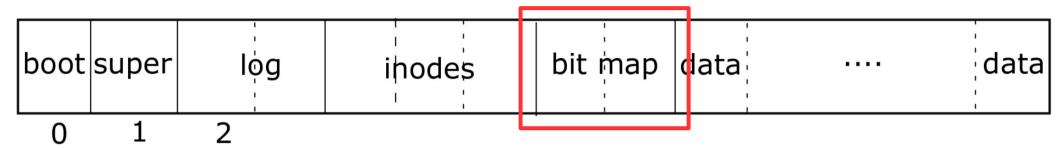
    Check if a pointer in the

5181
5182
        brelse(bp);
                                      indirect block is already
5183
        return addr;
                                      allocated
5184
```

5187 }



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):
           if((bp->data[bi/8] \& m) == 0){ // Is block free?}
4814
4815
             bp->data[bi/8] |= m; // Mark block in use.
4816
             log write(bp);
             brelse(bp);
4817
4818
             bzero(dev, b + bi);
4819
             return b + bi;
4820
4821
         brelse(bp);
4822
4823
4824
       panic("balloc: out of blocks");
4825 }
```

```
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):
           if((bp->data[bi/8] \& m) == 0){ // Is block free?}
4814
4815
             bp->data[bi/8] |= m; // Mark block in use.
4816
             log write(bp);
             brelse(bp);
4817
4818
             bzero(dev, b + bi);
4819
             return b + bi;
4820
4821
         brelse(bp);
4822
4823
4824
       panic("balloc: out of blocks");
4825 }
```

```
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);
             brelse(bp);
4817
4818
             bzero(dev, b + bi);
4819
             return b + bi;
4820
4821
         brelse(bp);
4822
4823
4824
       panic("balloc: out of blocks");
4825 }
```

```
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):
           if((bp->data[bi/8] \& m) == 0){ // Is block free?}
4814
4815
             bp->data[bi/8] |= m; // Mark block in use.
4816
             log write(bp);
             brelse(bp);
4817
4818
             bzero(dev, b + bi);
4819
             return b + bi;
4820
4821
         brelse(bp);
4822
4823
4824
       panic("balloc: out of blocks");
4825 }
```

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))
           panic("dirlink read");
5371
5372
         if(de.inum == 0)
5373
           continue:
5374
         if(namecmp(name, de.name) == 0){
5375
           // entry matches path element
5376
           if(poff)
             *poff = off;
5377
5378
             inum = de.inum;
5379
             return iget(dp->dev, inum);
5380
                                               dirlookup()
5381
      }
5382
5383
       return 0;
5384 }
```

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

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

```
5505 namex(char *path, int nameiparent, char *name)
5506 {
. . .
       if(*path == '/')
5509
5510
         ip = iget(ROOTDEV, ROOTINO);
5511
       else
         ip = idup(proc->cwd);
5512
5513
       while((path = skipelem(path, name)) != 0){
5514
5515
         ilock(ip);
         if(ip->type != T DIR){
5516
           iunlockput(ip);
5517
5518
           return 0;
5519
. . .
         if((next = dirlookup(ip, name, 0)) == 0){
5525
5526
           iunlockput(ip);
5527
           return 0;
5528
5529
         iunlockput(ip);
5530
         ip = next;
5531
5532
       if(nameiparent){
         iput(ip);
5533
                                                             namex()
5534
         return 0;
5535
5536
       return ip;
5537 }
```

```
5505 namex(char *path, int nameiparent, char *name)
5506 {
. . .
      if(*path == ',')
5509
         ip = iget(ROOTDEV, ROOTINO);
5510
5511
      else
5512
         ip = idup(proc->cwd);
      // skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5513
5514
       while((path = skipelem(path, name)) != 0){
5515
         ilock(ip);
         if(ip->type != T DIR){
5516
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);
                                                             namex()
5534
         return 0;
5535
5536
       return ip;
5537 }
```

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

Eaxmple: 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 };
```

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

Eaxmple: sys_open

```
5612 struct {
                                             Files and
5613 struct spinlock lock;
5614 struct file file[NFILE];
                                             filealloc()
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++){</pre>
5631
         if(f\rightarrow ref == 0){
5632
          f \rightarrow ref = 1:
5633
          release(&ftable.lock);
5634
           return f;
5635
5636 }
5637
       release(&ftable.lock);
5638
       return 0;
5639 }
```

```
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
      for(fd = 0; fd < NOFILE; fd++){</pre>
5842
         if(proc->ofile[fd] == 0){
5843
          proc->ofile[fd] = f;
5844
5845
          return fd;
5846
5847 }
                               File descriptors
5848
      return -1;
                                 and fdalloc()
5849 }
```

Thank you!

```
4952 struct inode*
                                                   ialloc()
4953 ialloc(uint dev, short type)
4954 {
4955
       int inum;
4956 struct buf *bp;
4957 struct dinode *dip;
4958
4959
       for(inum = 1; inum < sb.ninodes; inum++) {</pre>
4960
         bp = bread(dev, IBLOCK(inum, sb));
4961
         dip = (struct dinode*)bp->data + inum%IPB;
4962
         if(dip->type == 0){ // a free inode
4963
           memset(dip, 0, sizeof(*dip));
4964
           dip->type = type;
4965
           log_write(bp); // mark it allocated on the disk
           brelse(bp);
4966
4967
           return iget(dev, inum);
         }
4968
4969
         brelse(bp);
4970
      }
4971
       panic("ialloc: no inodes");
4972 }
```

```
5160 bmap(struct inode *ip, uint bn)
. . .
                                                      bmap()
      if(bn < NDIRECT){</pre>
5165
         if((addr = ip->addrs[bn]) == 0)
5166
5167
           ip->addrs[bn] = addr = balloc(ip->dev);
5168
         return addr;
5169
5170
     bn -= NDIRECT:
5171
5172
       if(bn < NINDIRECT){</pre>
5173
         // Load indirect block, allocating if necessary.
         if((addr = ip->addrs[NDIRECT]) == 0)
5174
         ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5175
     bp = bread(ip->dev, addr);
5176
5177 a = (uint*)bp->data;
         if((addr = a[bn]) == 0){
5178
5179
           a[bn] = addr = balloc(ip->dev);
5180
           log write(bp);
5181
        }
5182
         brelse(bp);
5183
         return addr;
5184
5185
      panic("bmap: out of range");
5186
5187 }
```

Example: write system call

```
5476 int
                                  Write() syscall
5477 sys_write(void)
5478 {
5479 struct file *f;
5480 int n;
5481 char *p;
5482
       if(argfd(0, 0, &f) < 0
5483
        || \operatorname{argint}(2, \&n) < 0 || \operatorname{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 {
if(f->type == FD_INODE)
. . .
5368
        int i = 0;
5369 while(i < n){
5373
          begin_trans();
5374
          ilock(f->ip);
5375
          if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5376
5377
            f \rightarrow off += r;
          iunlock(f->ip);
5378
          commit_trans();
5379
                                     Write several
5386 }
                                   blocks at a time
5390 }
```

```
6056 static struct inode*
6057 create(char *path, short type, short major, short minor)
6058 {
. . .
       if((ip = dirlookup(dp, name, &off)) != 0){
6067
6068
         iunlockput(dp);
6069
         ilock(ip);
6070
         if(type == T FILE && ip->type == T FILE)
6071
          return ip;
6072
         iunlockput(ip);
6073
         return 0;
6074
6075
      if((ip = ialloc(dp->dev, type)) == 0)
6076
6077
         panic("create: ialloc");
6078
. . .
6085
       if(type == T_DIR){ // Create . and .. entries.
6086
         dp->nlink++; // for ".."
6087
         iupdate(dp);
6088
         // No ip->nlink++ for ".": avoid cyclic ref count.
         if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6089
6090
           panic("create dots");
6091
6092
                                                            dirlookup()
6093
       if(dirlink(dp, name, ip->inum) < 0)</pre>
6094
         panic("create: dirlink");
. . .
6098
       return ip;
```

6099 }