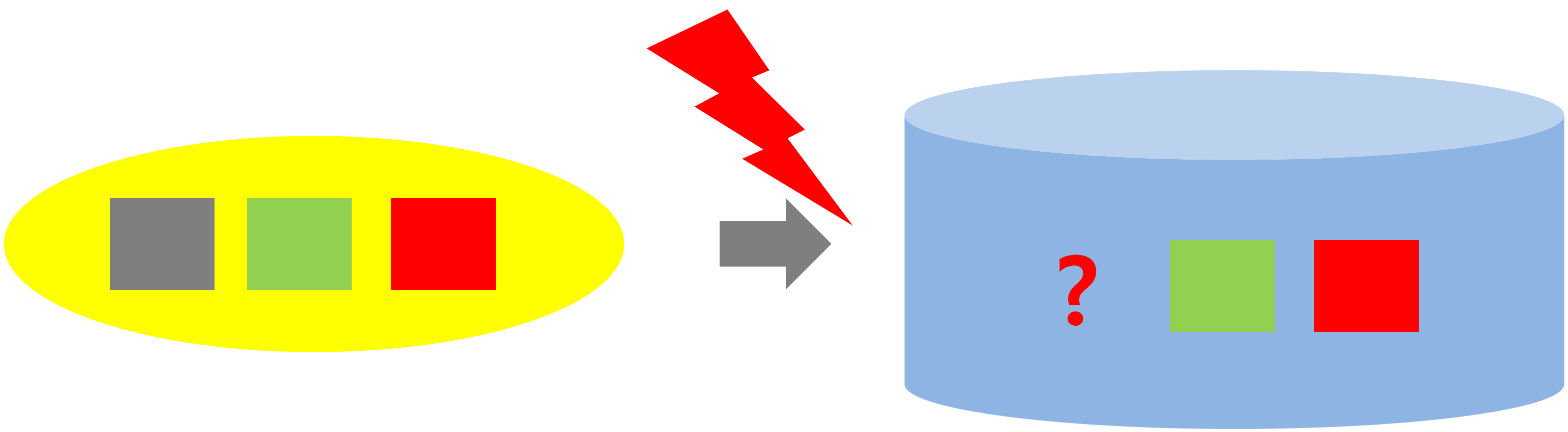
Filesystem: Logging

Youjip Won

KAIST EE

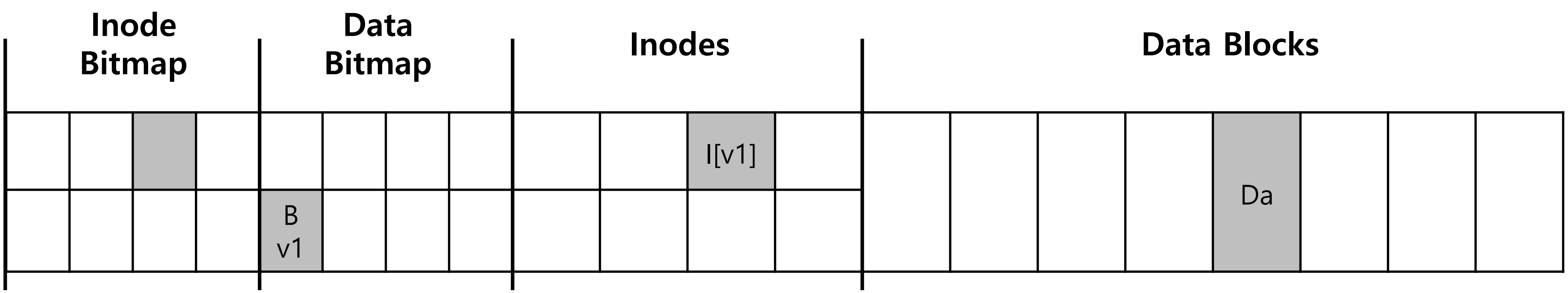
Crash



`create("hello.c")`

An Example of Crash

- Scenario
  - Append of a single data block to an existing file.

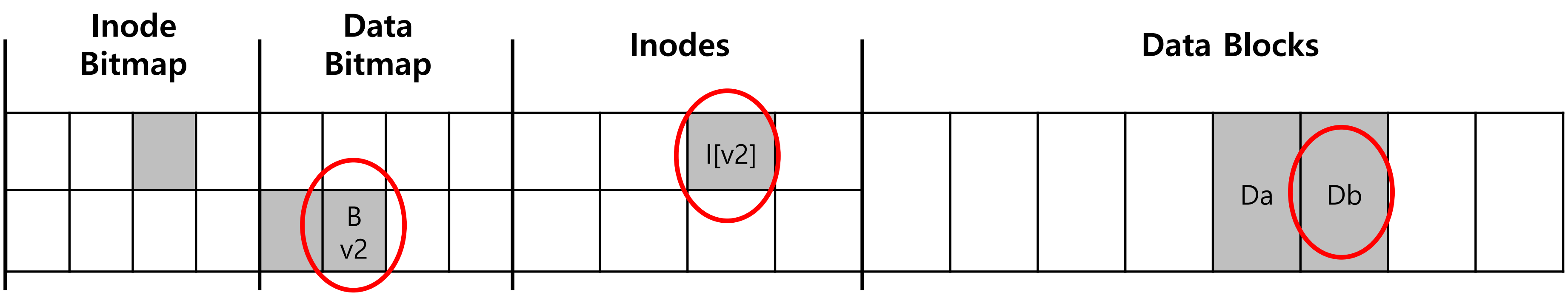


Before Append a single data block

Youjip Won
3

An Example of Crash

- File system perform three writes to the disk.
  - inode I[v2]
  - Data bitmap B[v2]
  - Data block (Db)



After Append a single data block

Crash Scenario

- Only one of the below block is written to disk.
  - Data block (Db): lost update
  - Update inode (I[v2]) block: garbage, **consistency problem**
  - Updated bitmap (B[v2]): space leak

- Two writes succeed and the last one fails.
  - The inode(I[v2]) and bitmap (B[v2]), but not data (Db).: consistent
  - The inode(I[v2]) and data block (Db), but not bitmap(B[v2): inconsistent
  - The bitmap(B[v2]) and data block (Db), but not the inode(I[v2]): inconsistent

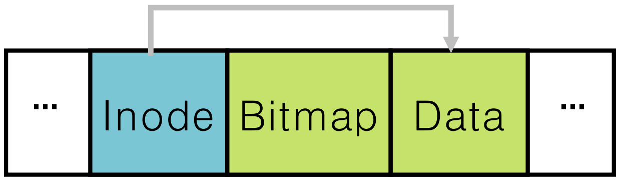
Metadata should be consistent.

Crash-consistency problem (consistent- update problem)

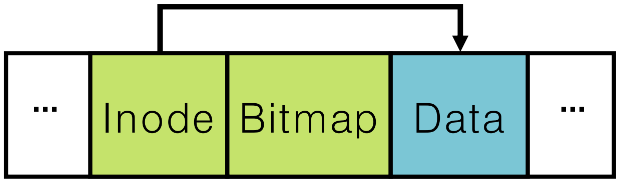
Youjip Won 5

Crash - No Inode

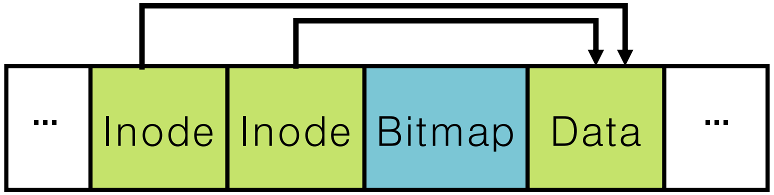
- Inode is lost.



- Data block write is lost.

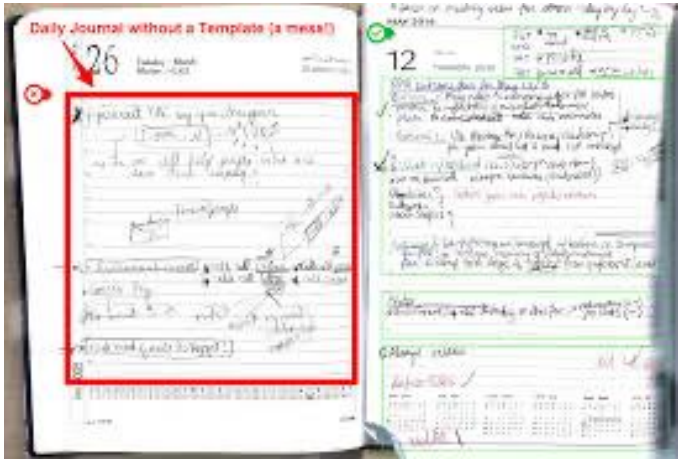


- Bitmap is loat.



Solution: Journaling (Write-Ahead-Logging)

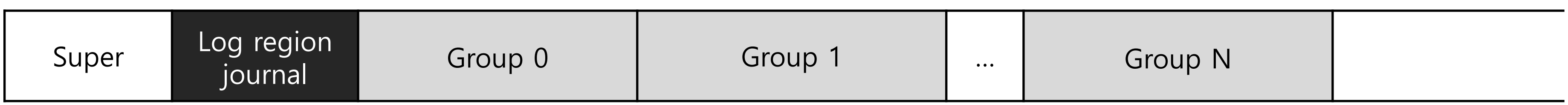
- In filesystem, it is “write -ahead-logging”.
- Bring back the filesystem to safe state after system crash.
- Rule
  - when you update the metadata, record it to the log space (journal).
  - If it is stored to the log space safely, then reflect it to the original location sometime later.



Log Region

- File system reserves some small amount of space within the partition or on another device.

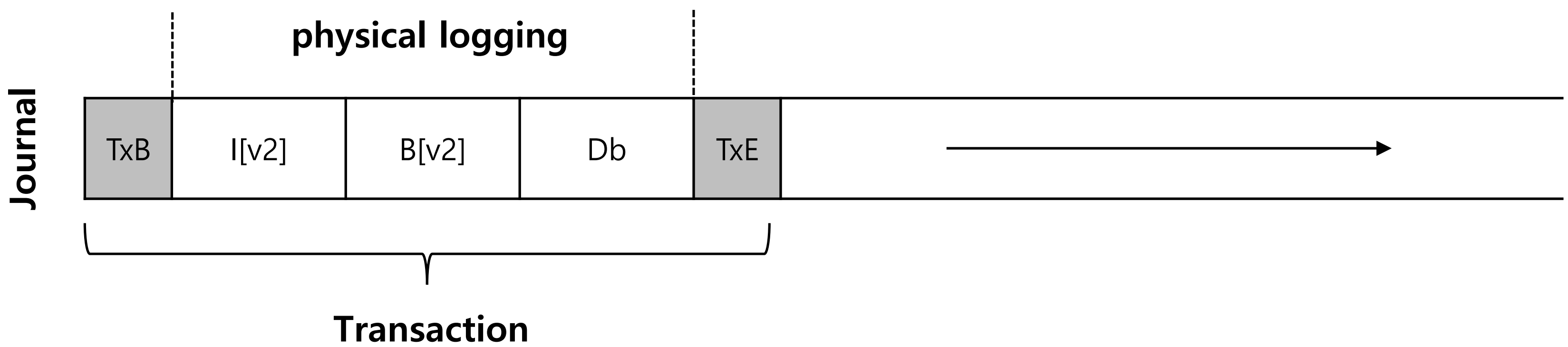
without journaling



with journaling

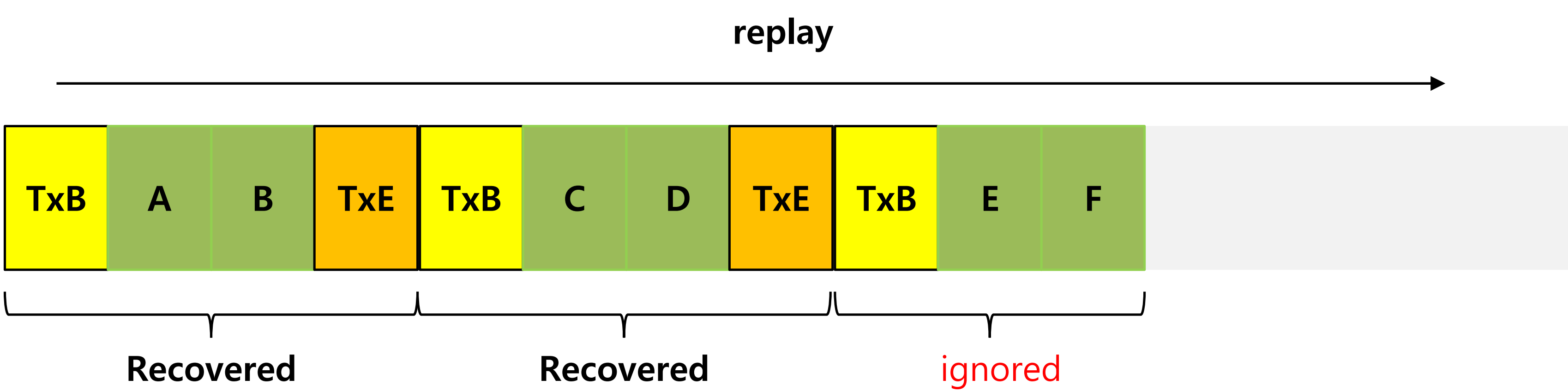
Transaction

- A set of blocks that need to be written as a single unit.
  - Transaction header (TxB): Place a description of all the disk writes it wishes to make in a lo g on the disk.
  - Log blocks
  - Transaction commit mark (TxE): Once the system call has logged all of its writes, it writes a special commit record to the disk indicating that the log contains a complete operation.



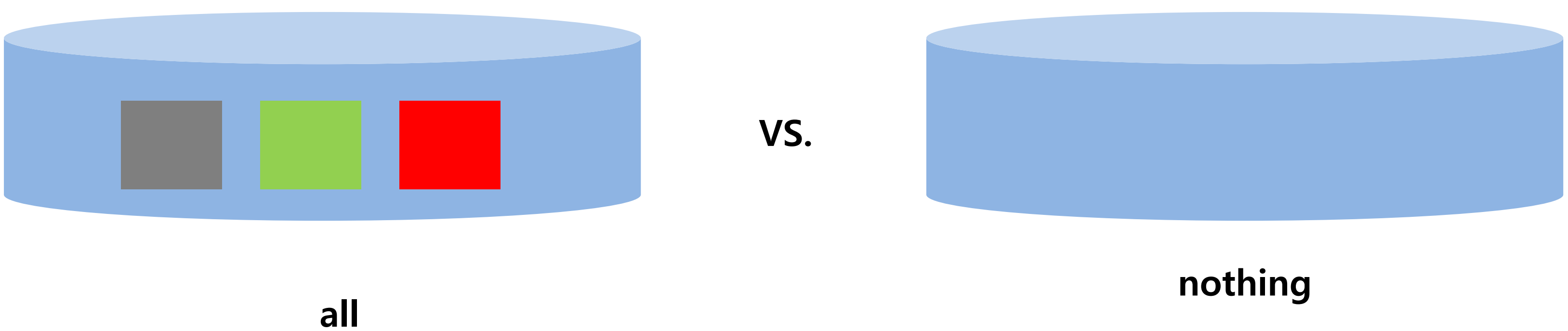
Logging and Recovery in XV6

- Recovery
  - Scan the log region and replay the log.
- Incomplete transaction
  - For the transaction with commit record missing, the recovery code ignores it.
  - The state of the disk will be if the operation had not even started.
- Committed transaction (Complete Transaction)
  - If the crash occurs after the operation commits, the recovery will replay all of th e operation’s writes.

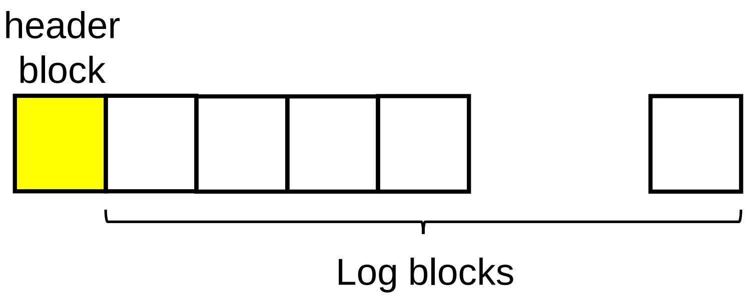


Logging and Recovery in XV6

- The log makes the operation ***atomic*** with respect to crash.
  - After recovery, either all of the operation's write appear on the disk, or none of t hem appear.



Structure of log region in xv6



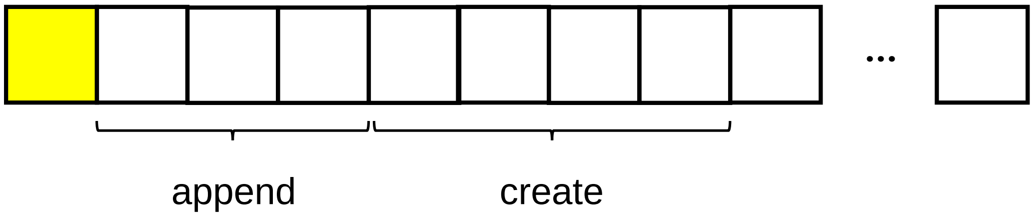
- Header block

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

- Header block: TxB + TxE
  - written when a transaction commits
  - count is set to zero after copying the log blocks to the file system.

Structure of Log Region

- The log region can accommodate one log structure.
- Compound transaction

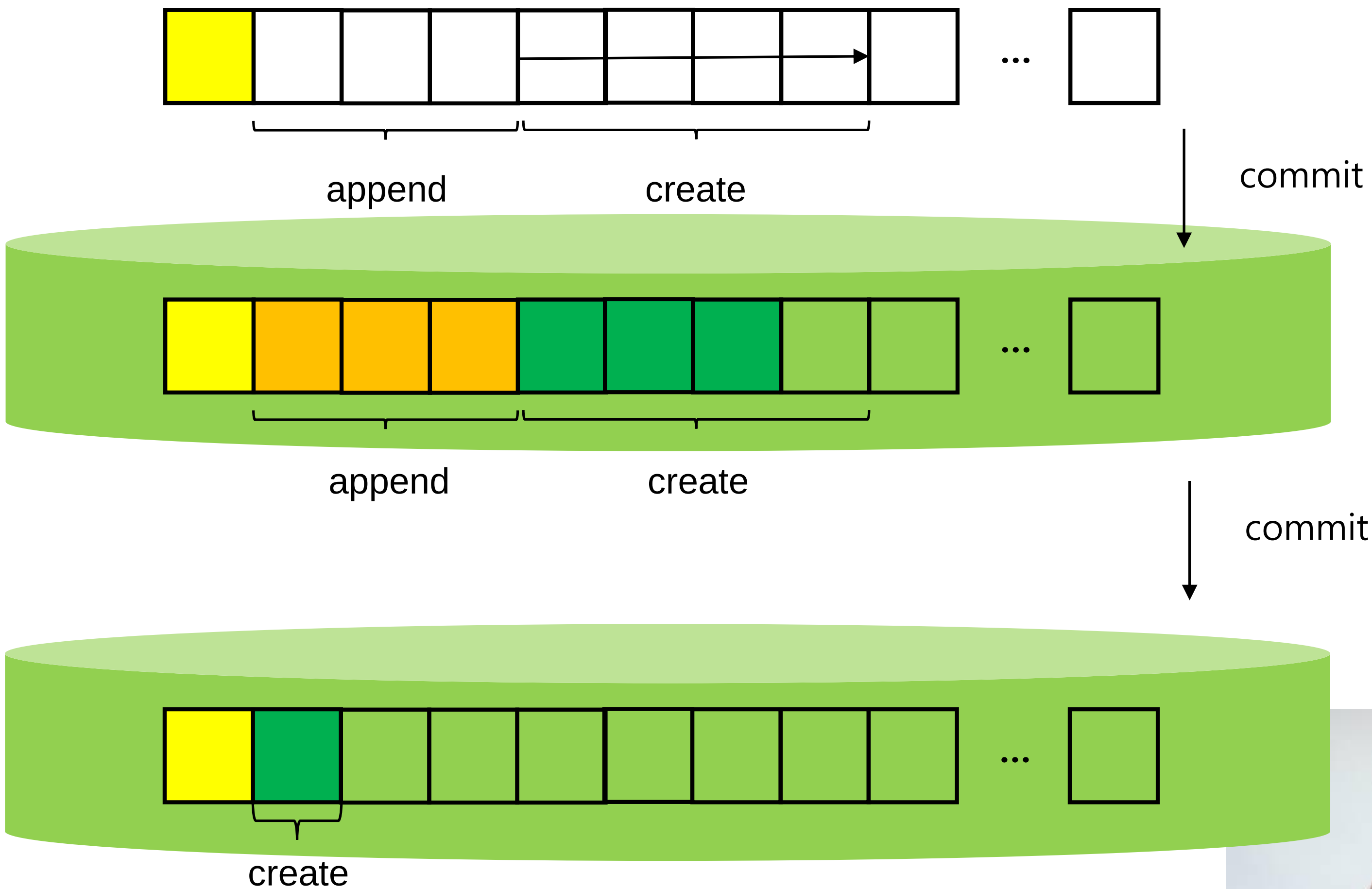


  - multiple system calls into one transaction.

- The total number of blocks written by the system calls in a transaction must fit in that space.
  - Large system call is broken into smaller pieces.
  - A system call can only start when there is a space in the log region.

Structure of Log Region

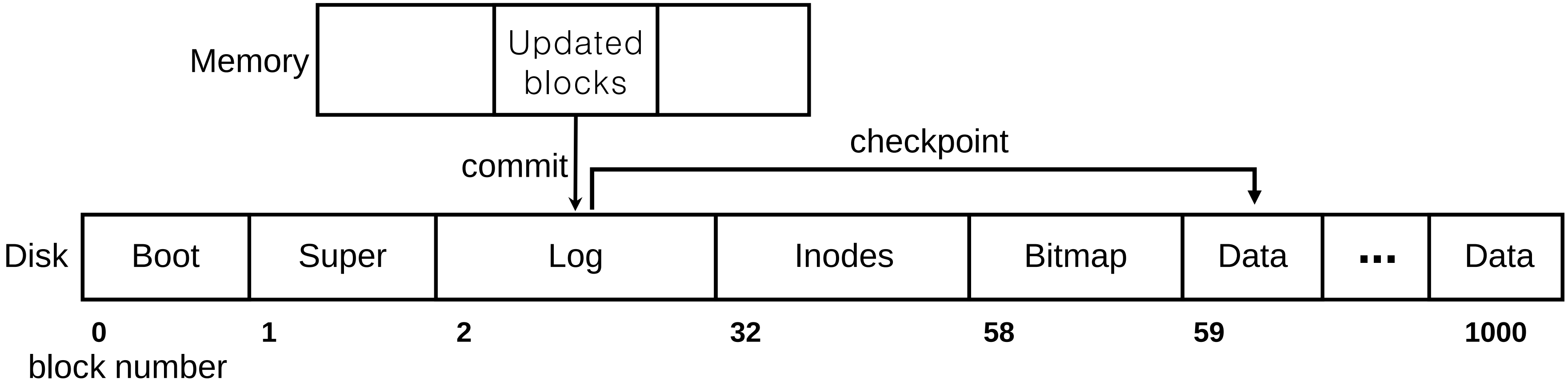
- To commit a transaction
  - Wait for the existing system call to finish



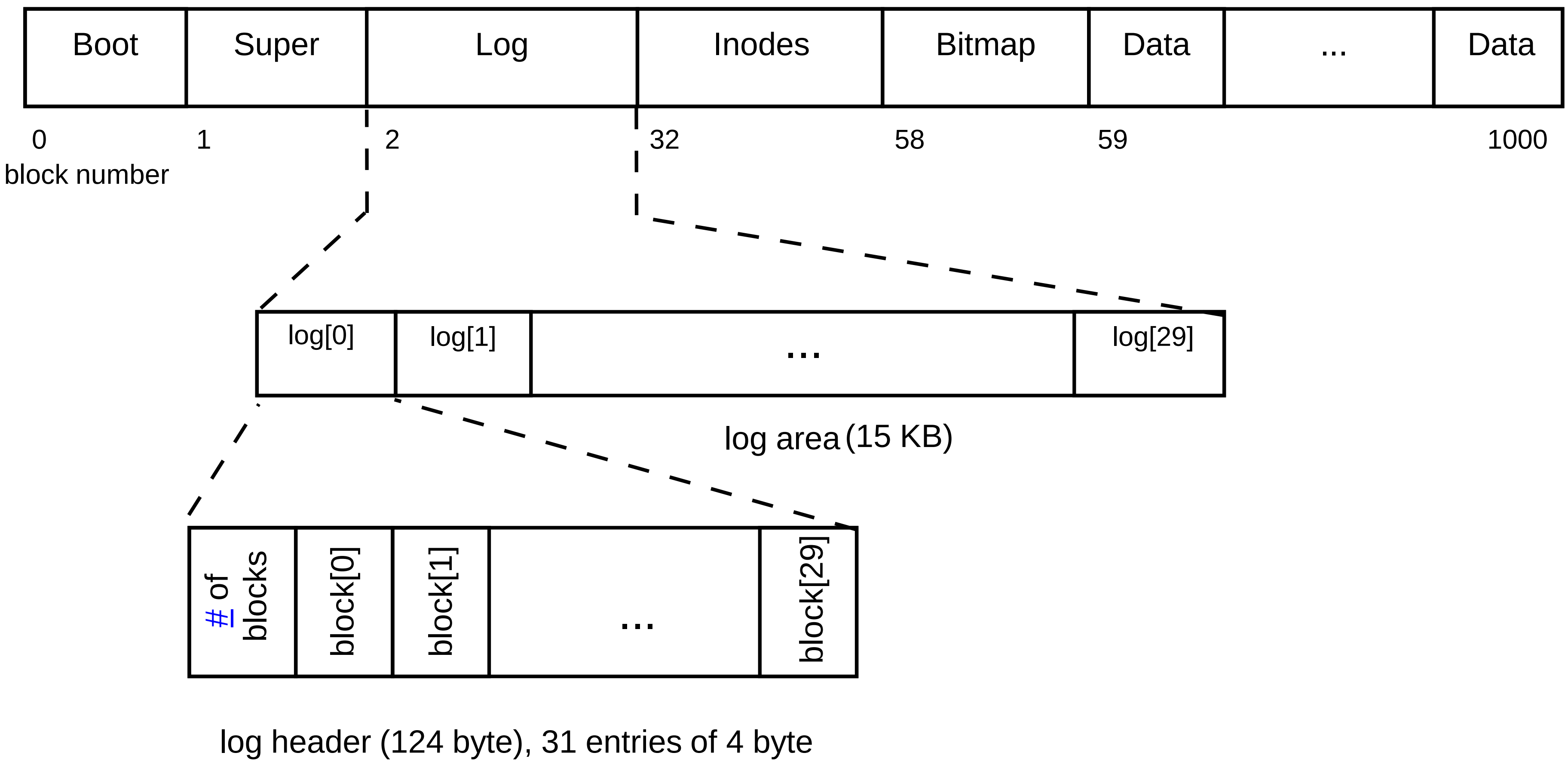
Logging in xv6

- Transaction: A set of blocks that should be written with ACID properties.
- Commit: Writes a transaction to log area.
- Checkpoint: Writes blocks in a committed transaction to their places.

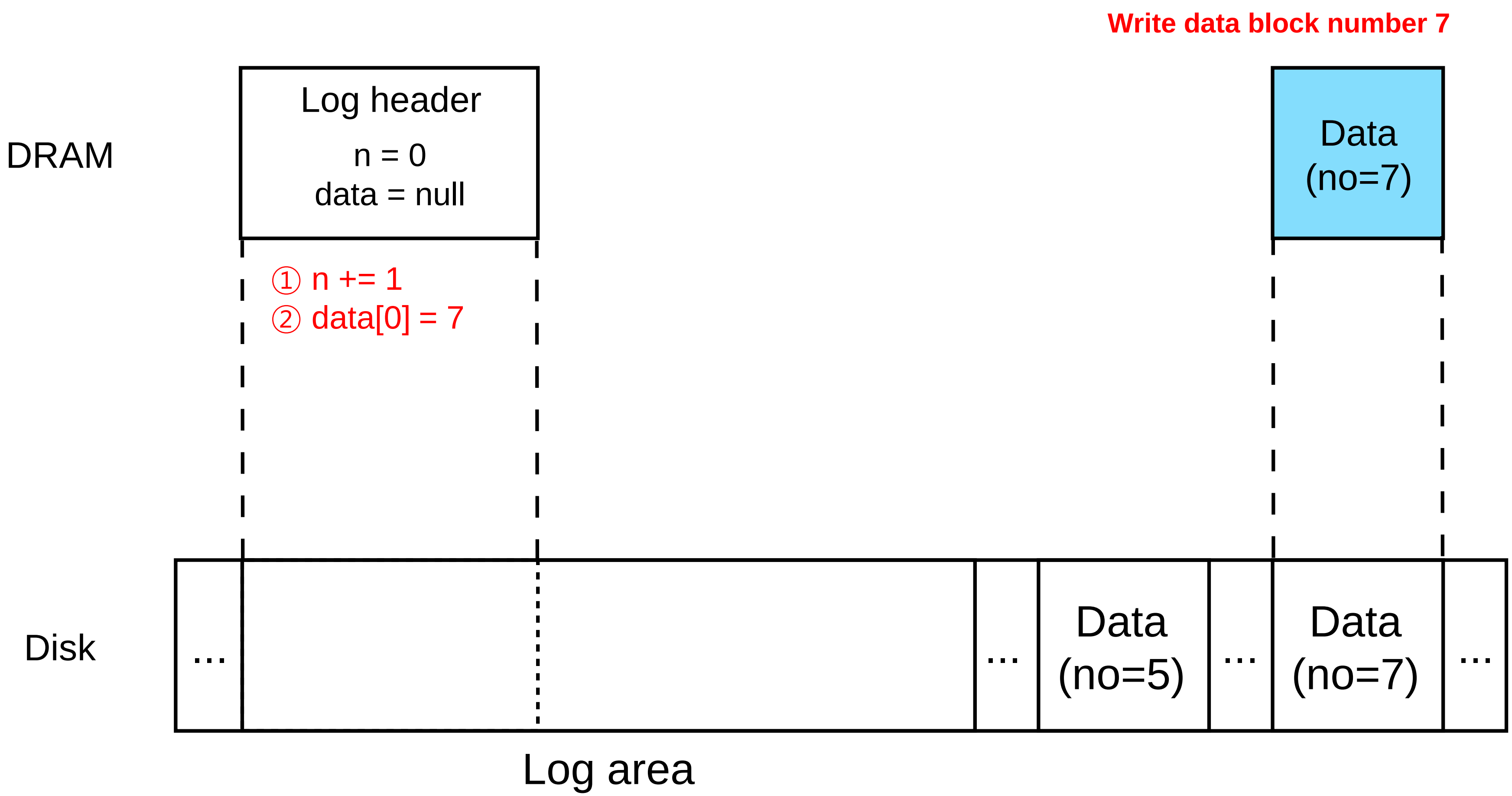
1. Collects the updated contents in memory and freeze them.(Creating a Transaction).
2. Writes them to log area (Commit).
3. Writes them to its places after commit (Checkpoint).



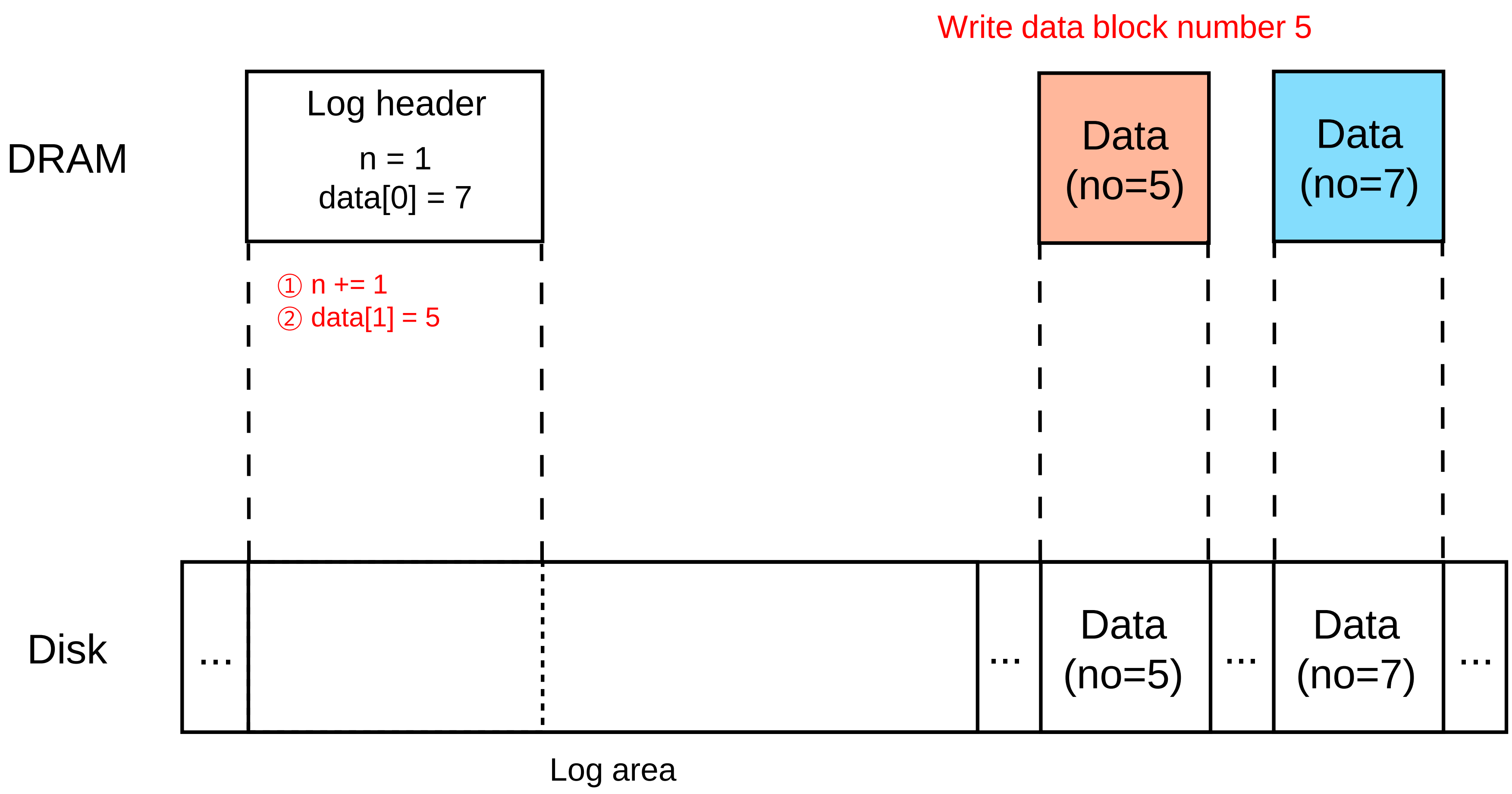
Log area in xv6



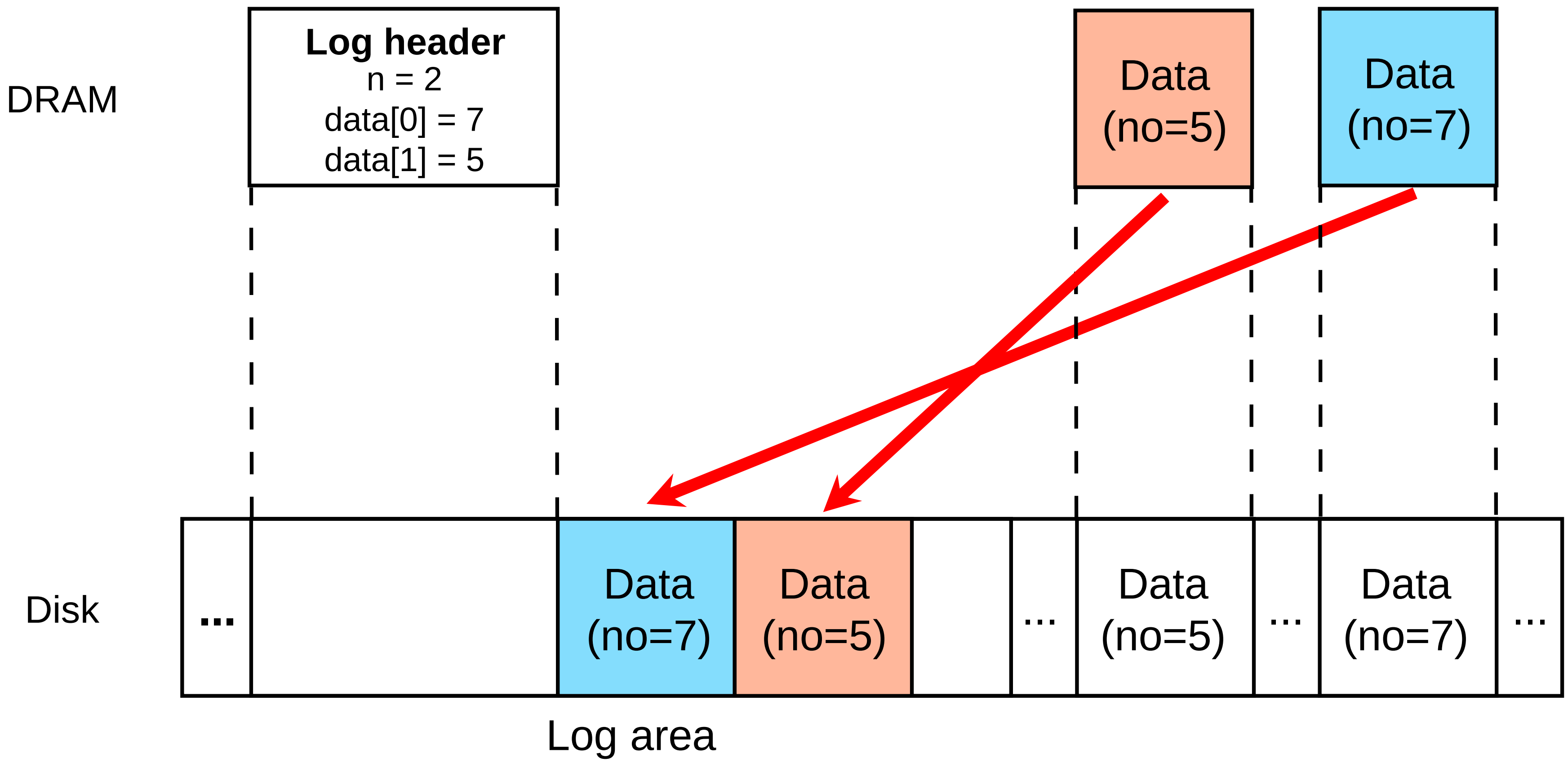
Logging (write the block 7 and block 5)



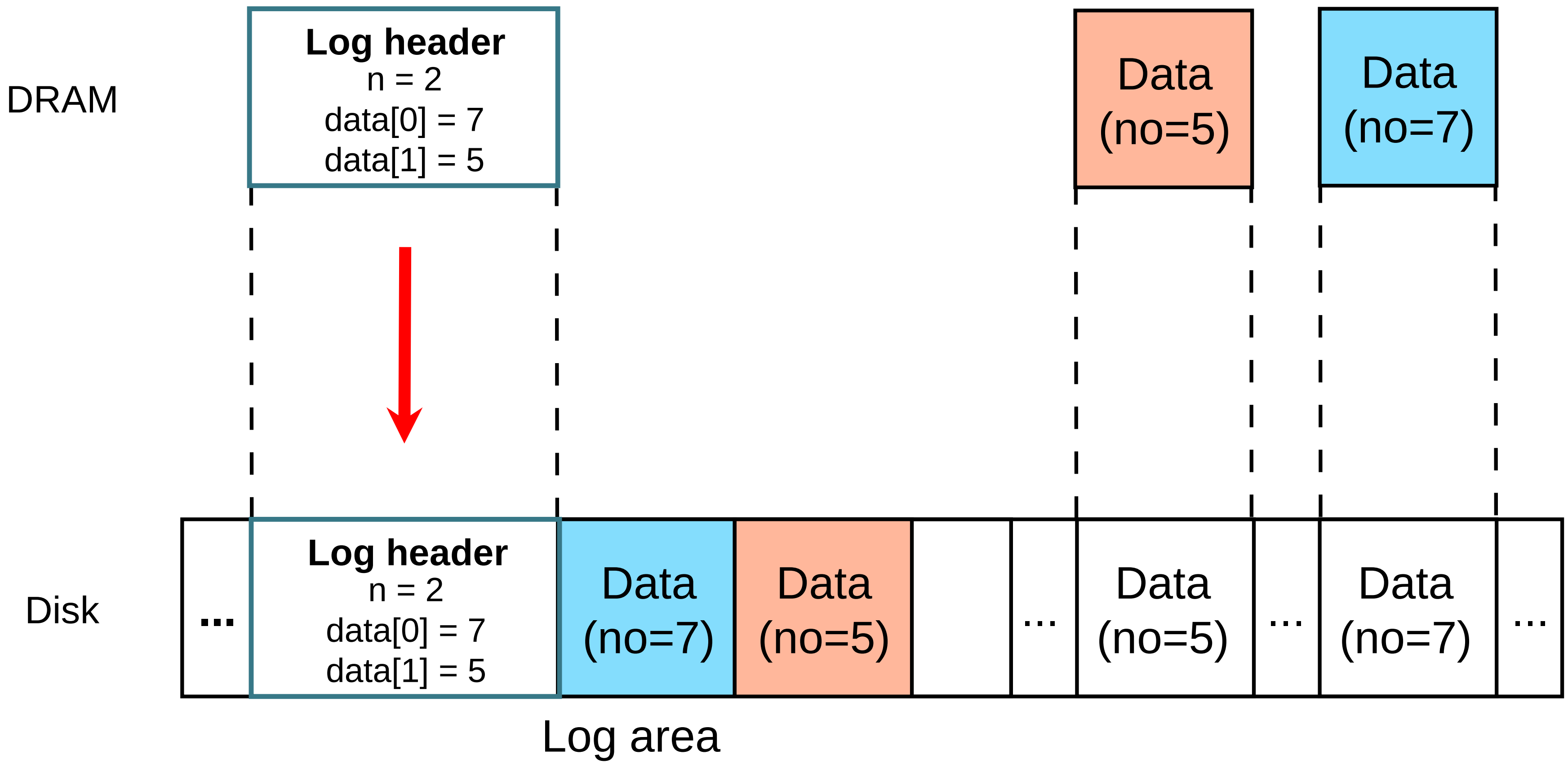
Logging (write the block 7 and block 5)



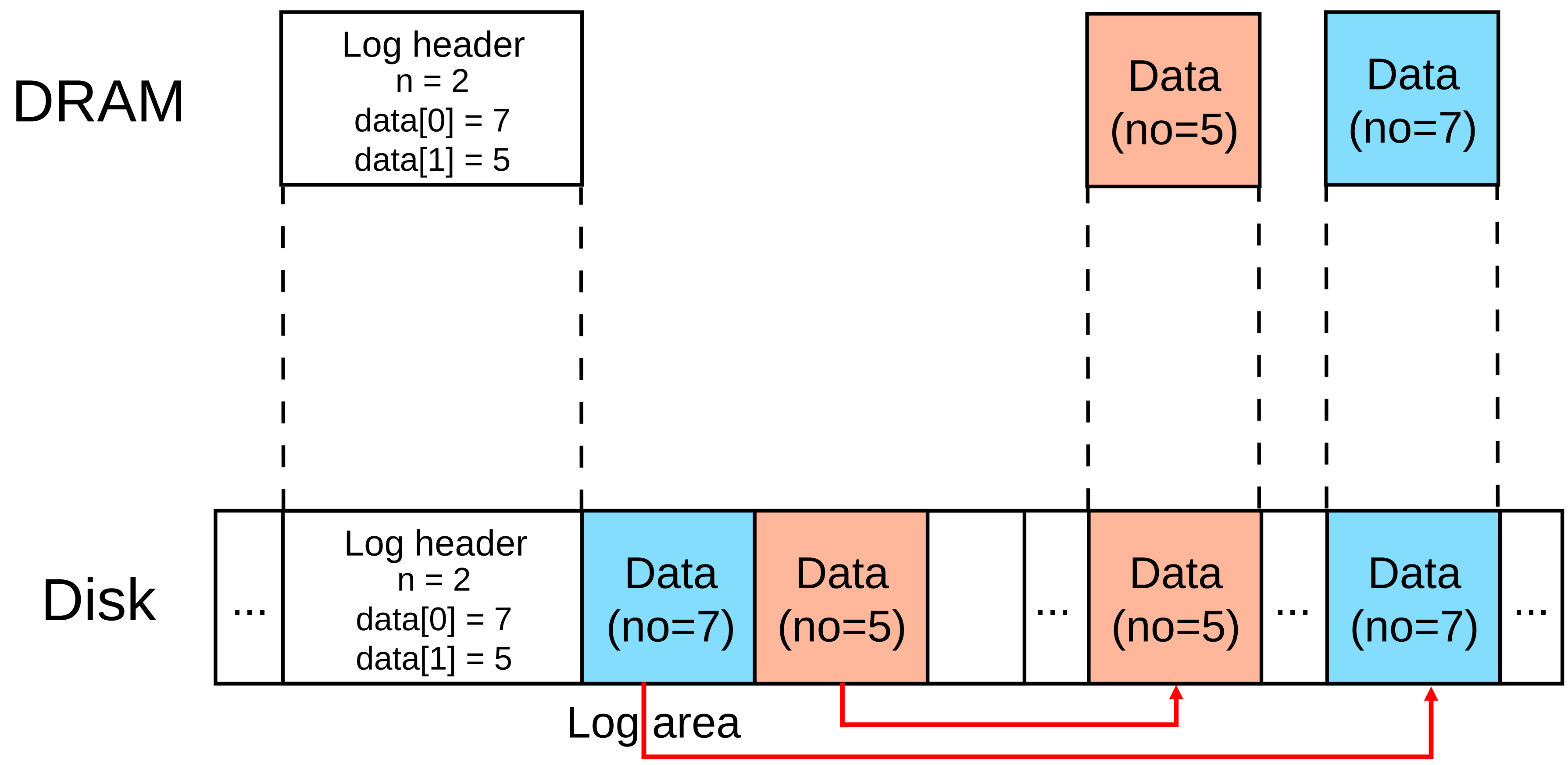
Logging (write the block 7 and block 5)



Logging (write the block 7 and block 5)

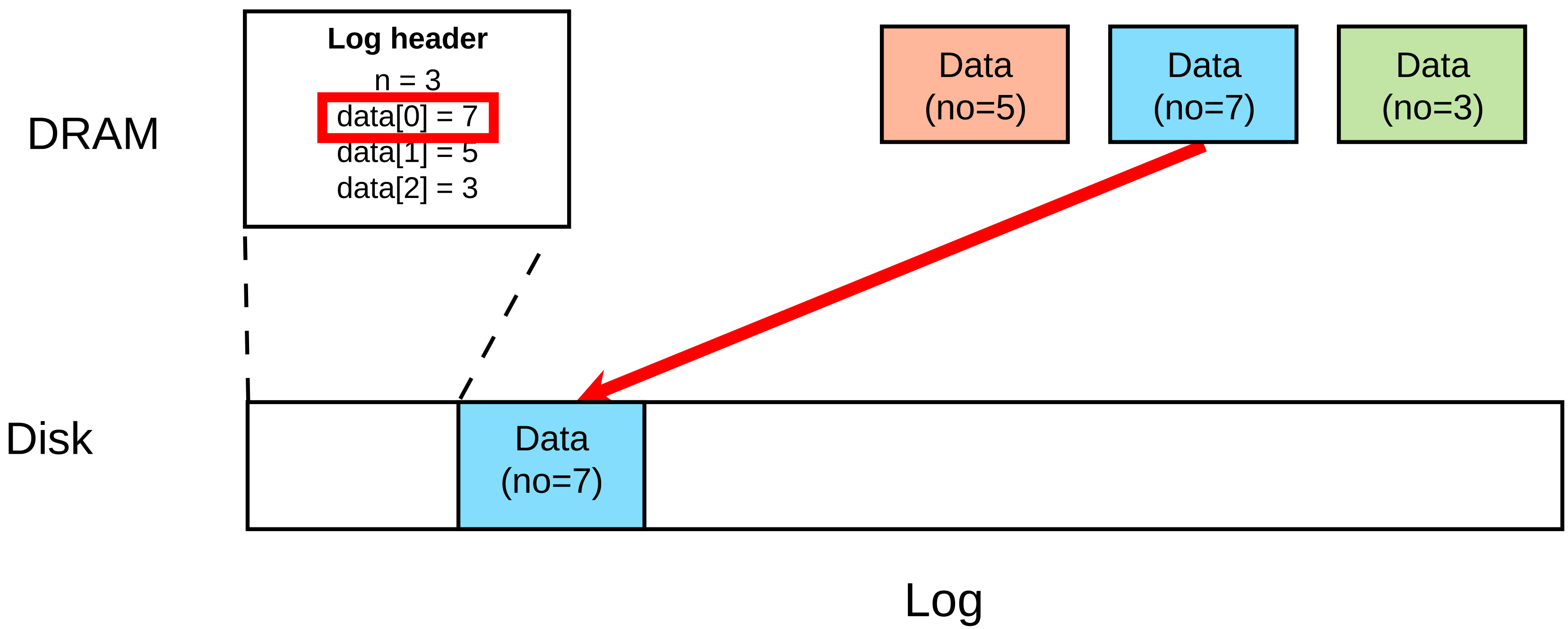


Checkpoint (write the block 7 and block 5)

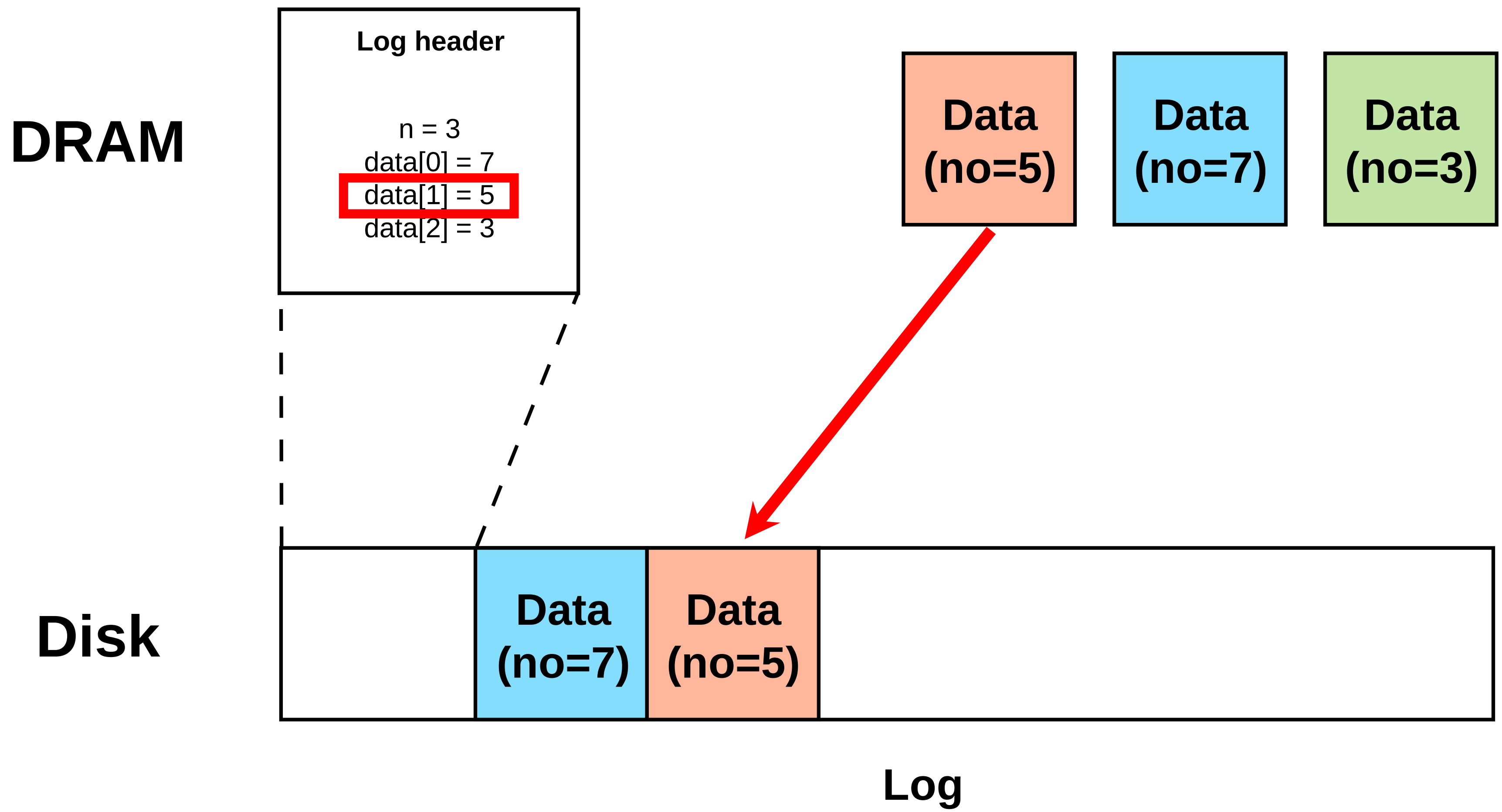


Process of commit in xv6

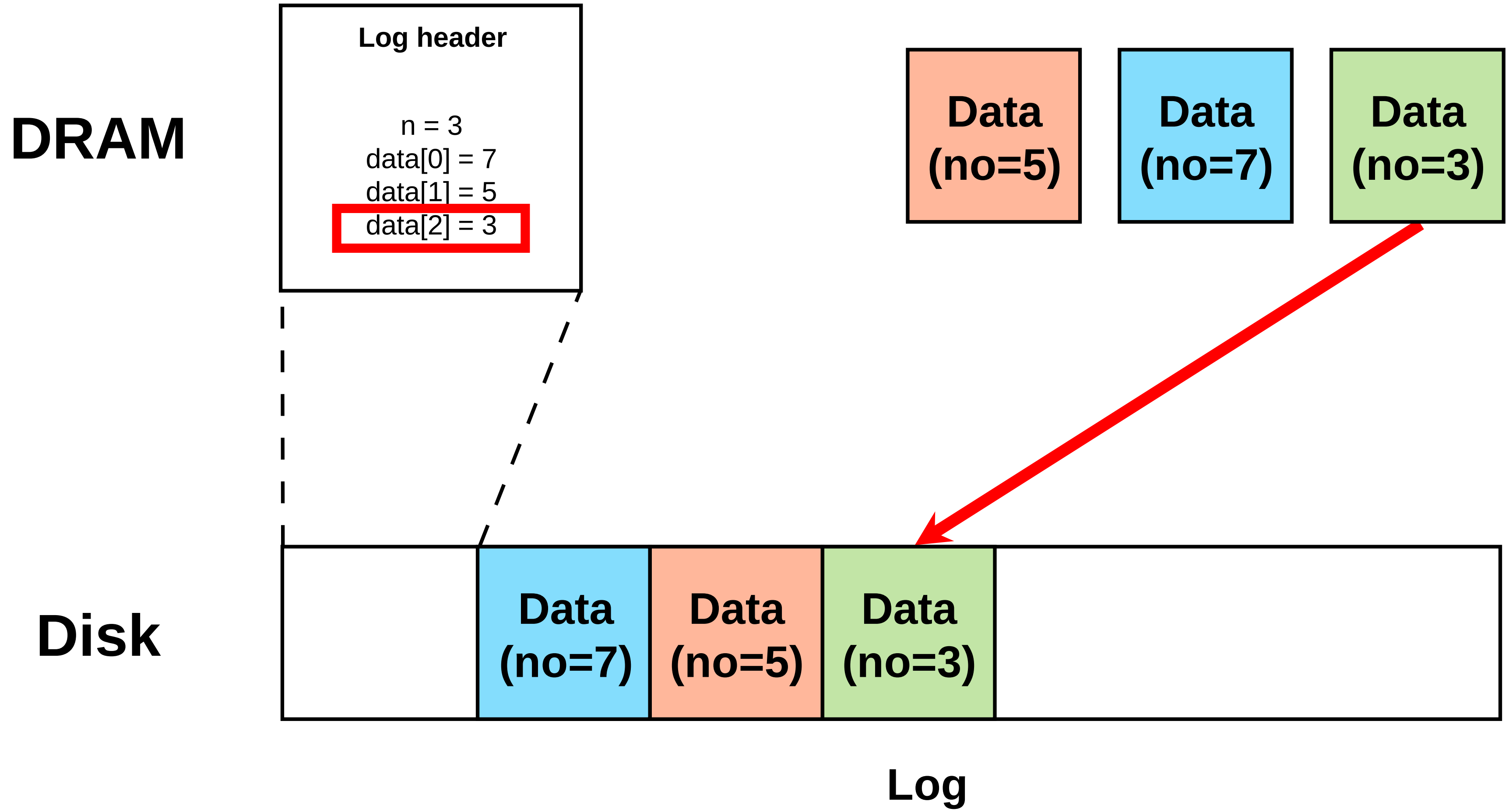
- Commit starts when there is no committing transaction.
- Write the data blocks specified in the log header to the log area persistently.
- Write the log header to the disk persistently.



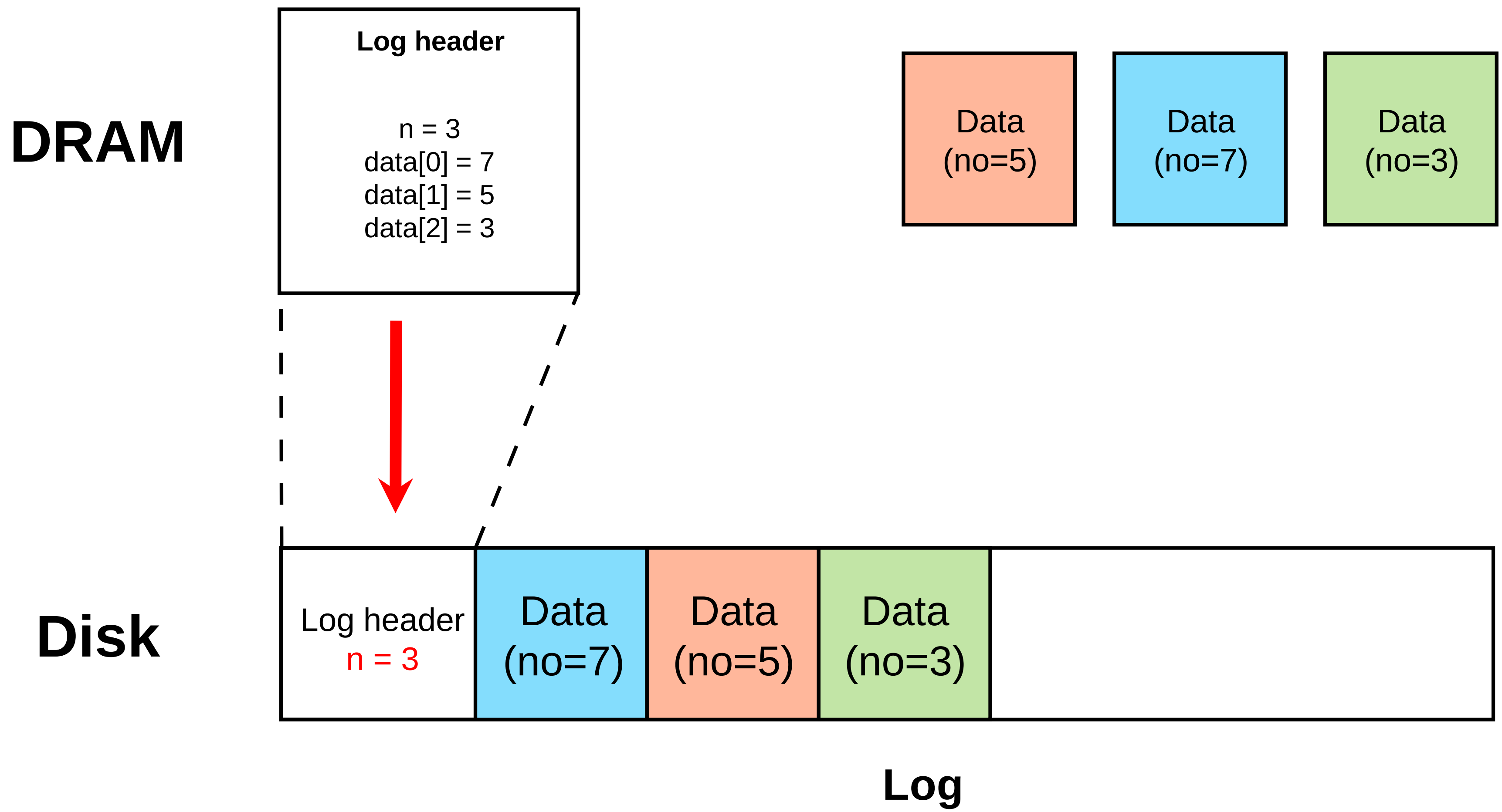
Process of commit in xv6



Process of commit in xv6

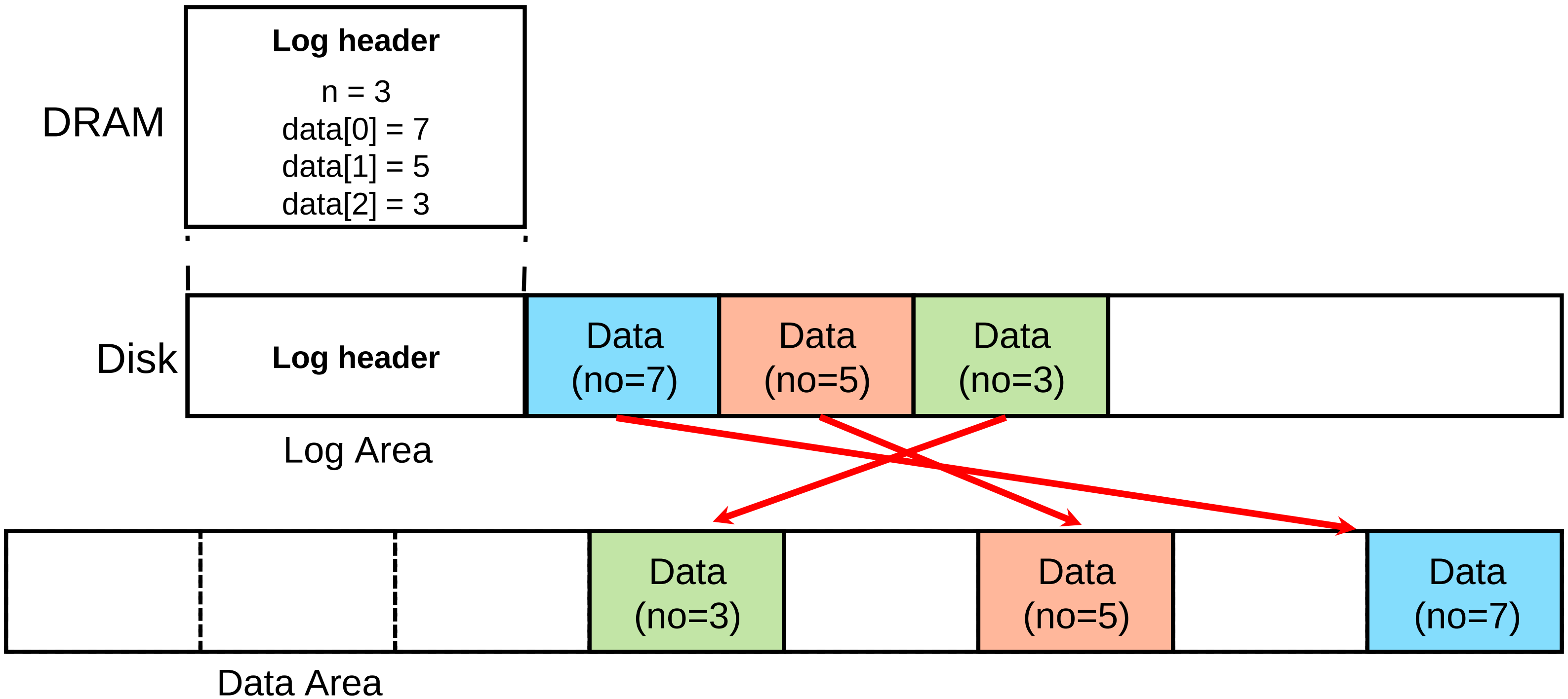


Process of commit in xv6

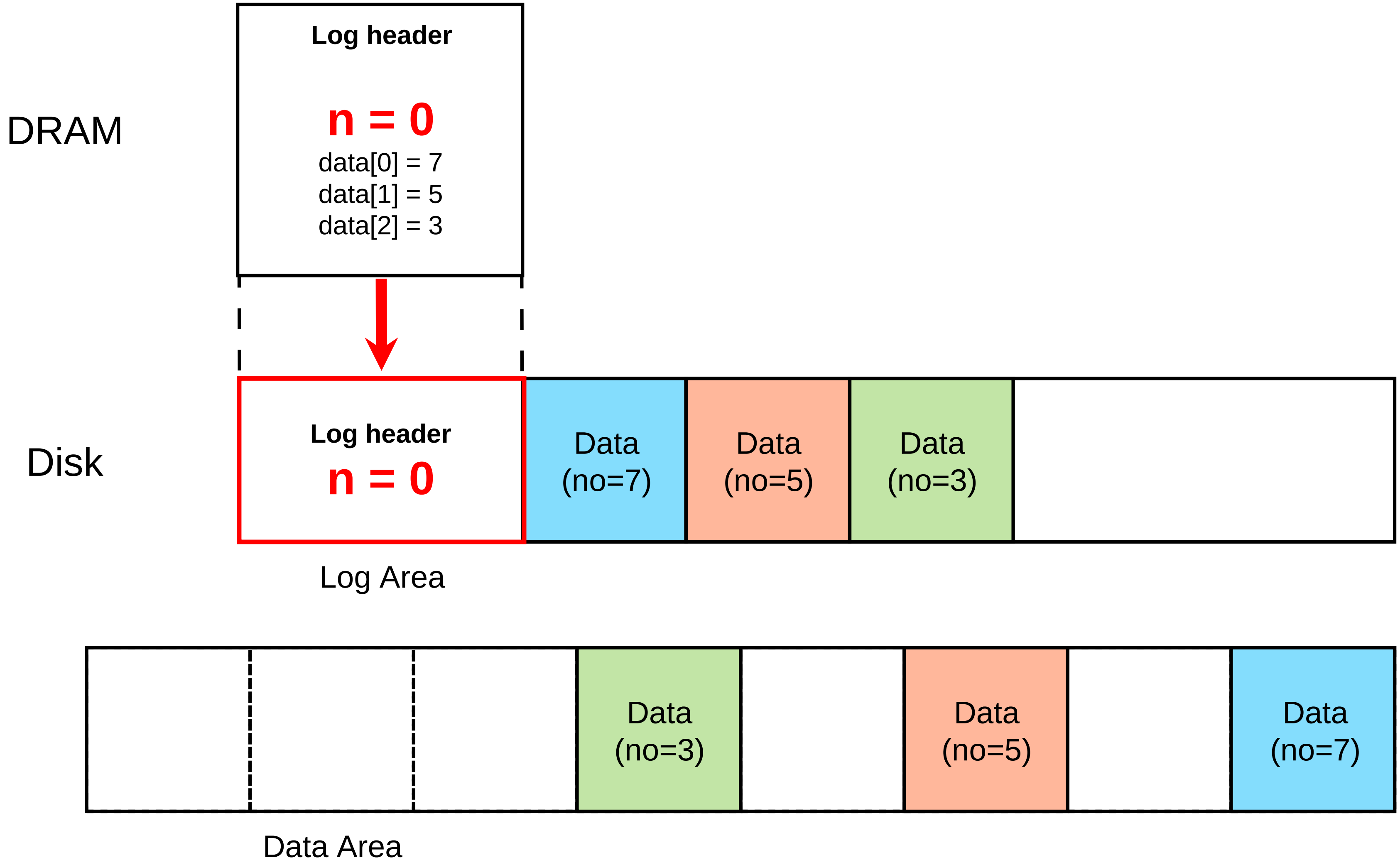


Process of checkpoint in xv6

- Checkpoint writes the committed data blocks to their original place.
- After the checkpoint, set the number of blocks in the log header to zero. Then, write the updated log header to the disk.



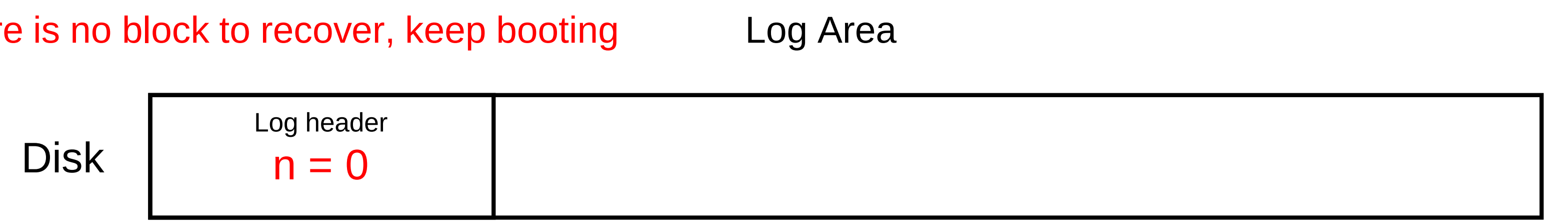
Process of checkpoint in xv6



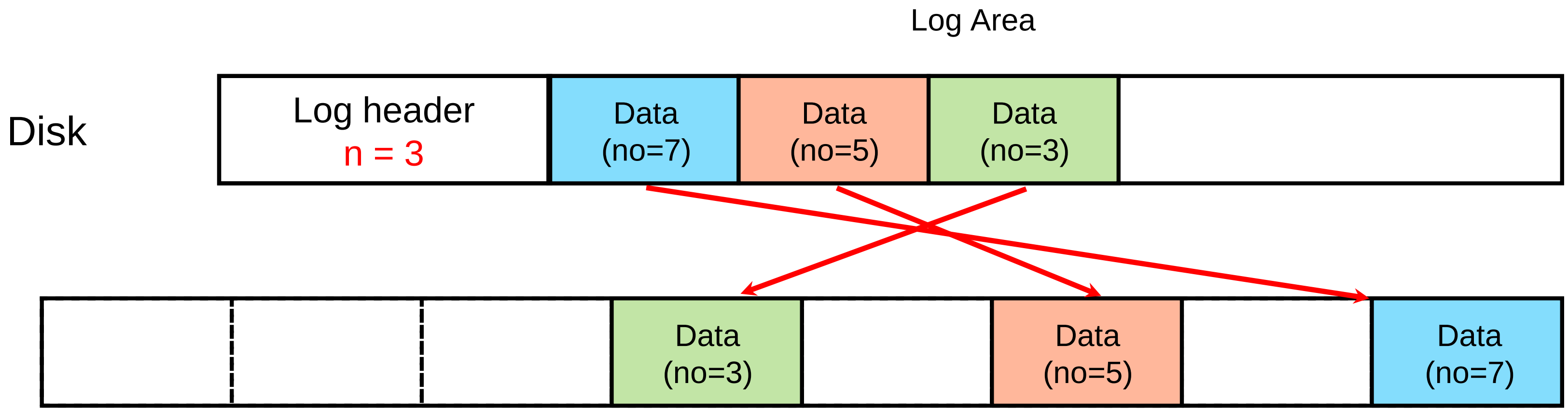
Recovery

- Recovery routine checks the "number of blocks" in the log header.
  - If the number of block in the log header is 0, it skip recovery phase.

If there is no block to recover, keep booting



  - Otherwise, it performs recovery; It write the blocks in the log area to the original locations.



Typical system call pattern

```
System call ()
```

1. wait for the existing commit to finish the available space in the log region.
2. update the buffer cache.
3. Register the buffer cache entries at the log header and pin the buffer cache blocks.
4. write them to the log region and checkpoint.

```
System call ()
 1. begin_op();
 2. …
 3. bp=bread(…) ;
 4. bp->data[…] = … ;
 5. log_write(bp) ;
 6. …
 7. end_op() ;
```

Code: begin_op()

- Before logging, it check status of log area.
- Wait till
  - The current commit finishes,
  - there is enough space available, or
  - there is no ongoing system calls (log.outstanding)

Code: begin_op()

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

If other threads is committing, wait for them.

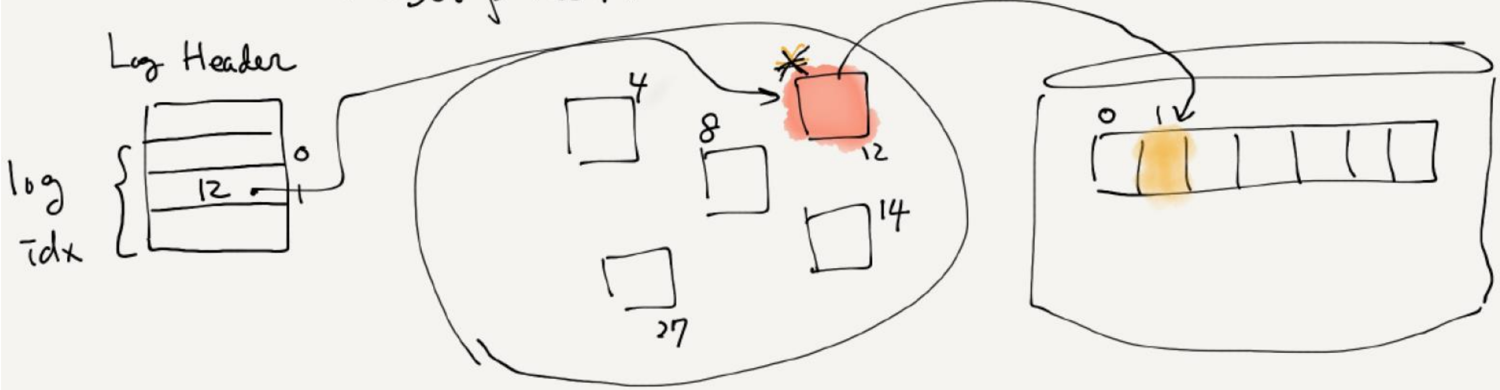
If log area have no enough area to log, wait for checkpoint by other thread

If it don’t need to wait, increase outstanding and start to log

Code: log_write()

log_write();

- records the blocks' sector # in memory.
- reserve a slot in the log.
- mark the buffer as DIRTY.
- prohibit the buffer from going to disk.
- absorption



Code: log_write()

```
void log_write(struct buf *b){
  int i;
  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (log.outstanding < 1)
    panic("log_write outside of trans");
  acquire(&log.lock);
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno)    // log absorbtion
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // prevent eviction
  release(&log.lock);
}
```

Add a new block to the log header

Code: end_op()

end_op();

- decrements the counts of outstanding system calls.
- If the count is 0, call commit(); .
- steps
  1. buffer blocks to the log slots: write_log();
  2. update header block: write_head();
  3. checkpoint: install_trans();
  4. reset the counter at log header: end_op();

Code: end_op()

- Complete logging: Commit and Checkpoint

```
void end_op(void){
  int do_commit = 0;
  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.committing)
    panic("log.committing");
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
  } else {
    // begin_op() may be waiting for log space, and decrementing
    // log.outstanding has decreased the amount of reserved space.
    wakeup(&log);
  }
  release(&log.lock);
  ...
```

Code: end_op()

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

Code: commit()

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

① Write log blocks to log area in storage

② Write log head to log area in storage (commit)

③ Write log blocks to original location in storage(checkpoint)

④ Initialize n of journal head to 0(transaction invalidation)

⑤ Write n initialized in ④ to storage

Code: write_log()

- write the updated blocks in the buffer cache to the on-disk log area.

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

① Acquiring buffer cache from the log area (to)

② Acquiring modified buffer cache (from)

③ Copy the contents of modified buffer cache(from) to buffer cache for log area(to)

④ Write buffer cache for log area to storage

⑤, ⑥ release buffer cache

Code: write_head()

- Write the log header to on-disk log area.

```
static void write_head(void){
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}
```

1. Acquire buffer cache for the first block of log area.
2. Copy the contents of log head to buffer cache.
3. Write buffer cache.

Code: install_trans()

- Checkpoint: write modified data blocks in buffer cache to on-disk area.

```
static void install_trans(void){
  int tail;

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

Recovery

After initializing log area, start recovery

```
void forkret(void){
  ...
  if (first) {
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
  }
}
```

```
void initlog(int dev) {
  if (sizeof(struct logheader) >= BSIZE)
    panic("initlog: too big logheader");

  struct superblock sb;
  initlock(&log.lock, "log");
  readsb(dev, &sb);
  log.start = sb.logstart;
  log.size = sb.nlog;
  log.dev = dev;
  recover_from_log();
}
```

Recovery

Perform log replay (checkpoint).

```
static void recover_from_log(void) {
  read_head();
  install_trans();  // if committed, copy from log to disk
  log.lh.n = 0;
  write_head();  // clear the log
}
```

Writing to a file

- Write data to file

```
file write ()

while () {
    begin_op();
    ilock(f->ip);
    r=writei(f->ip, ...);
    iunlock(f->ip);
    end_op();
}
```

Code: filewrite()

```
int filewrite(struct file *f, char *addr, int n){
  ...
  int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
  int i = 0;
  while(i < n){
    int n1 = n - i;
    if (n1 > max)
      n1 = max;
    begin_op();
    ilock(f->ip);
    if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
      f->off += r;
    iunlock(f->ip);
    end_op();
    ...
    i += r;
  }
  return i == n ? n : -1;
}
```

Whole data are devided to max size of log and written.

Excluding inode, indirect block, allocation blocks, and divided by 2

Code: writei()

- Write data to block, referring inode

```
int writei(struct inode *ip, char *src, uint off, uint n){
 ...
  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    brelse(bp);
  }
 ...
}
```

① Find buffer cache of a block to modify

② Get size to modify in the buffer

③ In memory update by memmove(dst, src, sz)

④ Add block number of modified block to log header

⑤ Release buffer cache

What if you switch (4) and (5)?

Important of logging

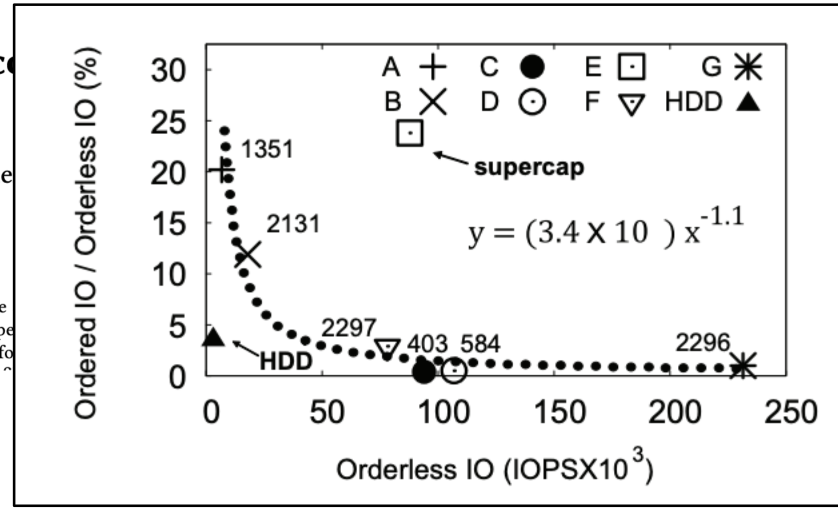
Scaling a file system to many c
using an operation log

Srivatsa S. Bhat,[†] Rasha Eqbal,[‡] Austin T. Cleme
M. Frans Kaashoek, Nickolai Zeldovich
MIT CSAIL

ABSTRACT

It is challenging to simultaneously achieve multicore scalability and high disk throughput in a file system. For exam-

allow file-system-intensive
10, 13, 23, 26, 31]. This pape
system design that allows fo



SpanFS: A Scalable File System on Fast Storage Devices

Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma and Jinpeng Huai
SKLSDE Lab, Beihang University, China
mashuai}@act.buaa.edu.cn, zblgeqian@gmail.com, huaijp@buaa.edu.cn

NAND flash-based

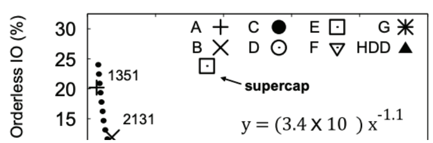
ized file system service with a collection of independent micro file system services, called *domains*, to achieve scalability on many-core. Each domain performs its file

Barrier-Enabled IO Stack for Flash Storage

Youjip Won[1] Jaemin Jung[2*] Gyeongyeol Choi[1]
Joontaek Oh[1] Seongbae Son[1] Jooyoung Hwang[3] Sangyeun Cho[3]
[1]*Hanyang University* [2]*Texas A&M University* [3]*Samsung Electronics*

Abstract

This work is dedicated to eliminating the overhead required for guaranteeing the *storage order* in the modern IO stack. The existing block device adopts a prohibitively expensive approach in ensuring the storage or-



summary

- Logging
- API's
  - `begin_op(), log_write(), end_op()`

```
System call ()
 1. begin_op();
 2. …
 3. bp=bread(…) ;
 4. bp->data[…] = … ;
 5. log_write(bp) ;
 6. …
 7. end_op() ;
```