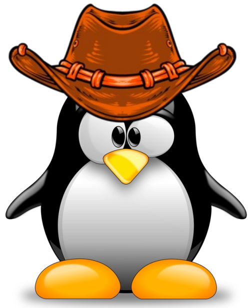
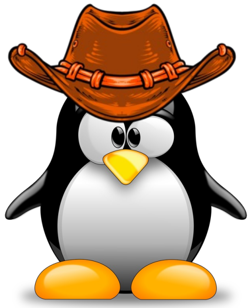


CSE 330: Operating Systems

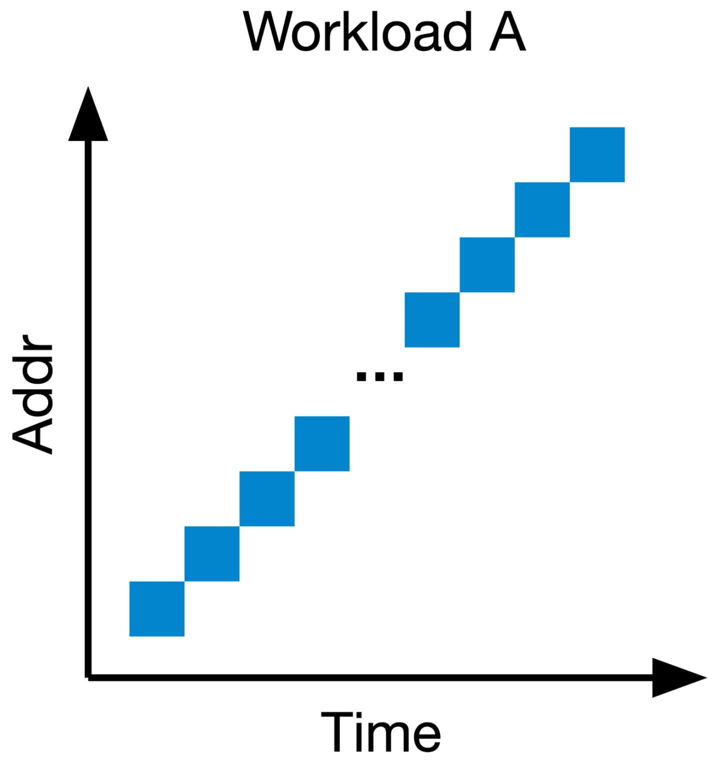
Adil Ahmad

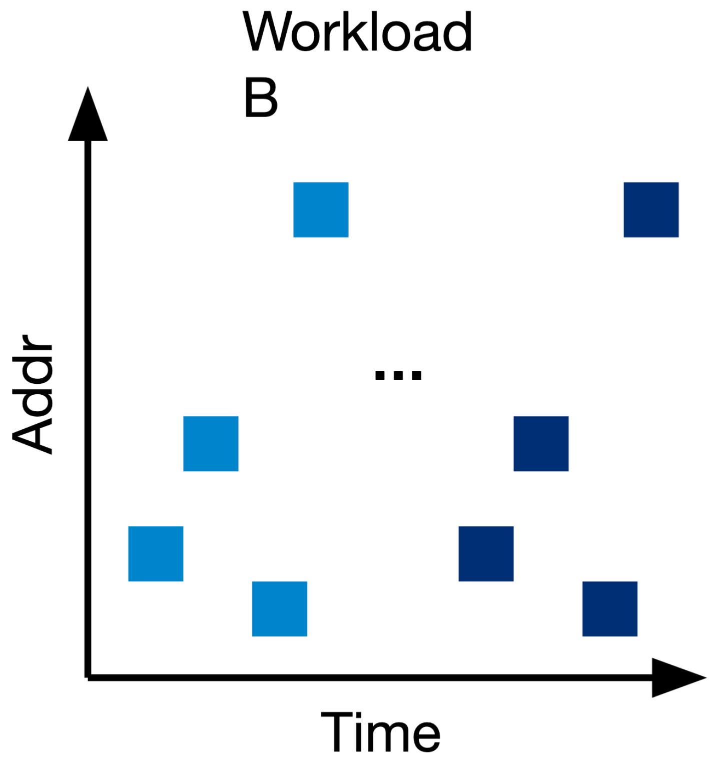
Lecture #20: Filesystem logging (crash recovery and consistency)



FS optimization(s) recap

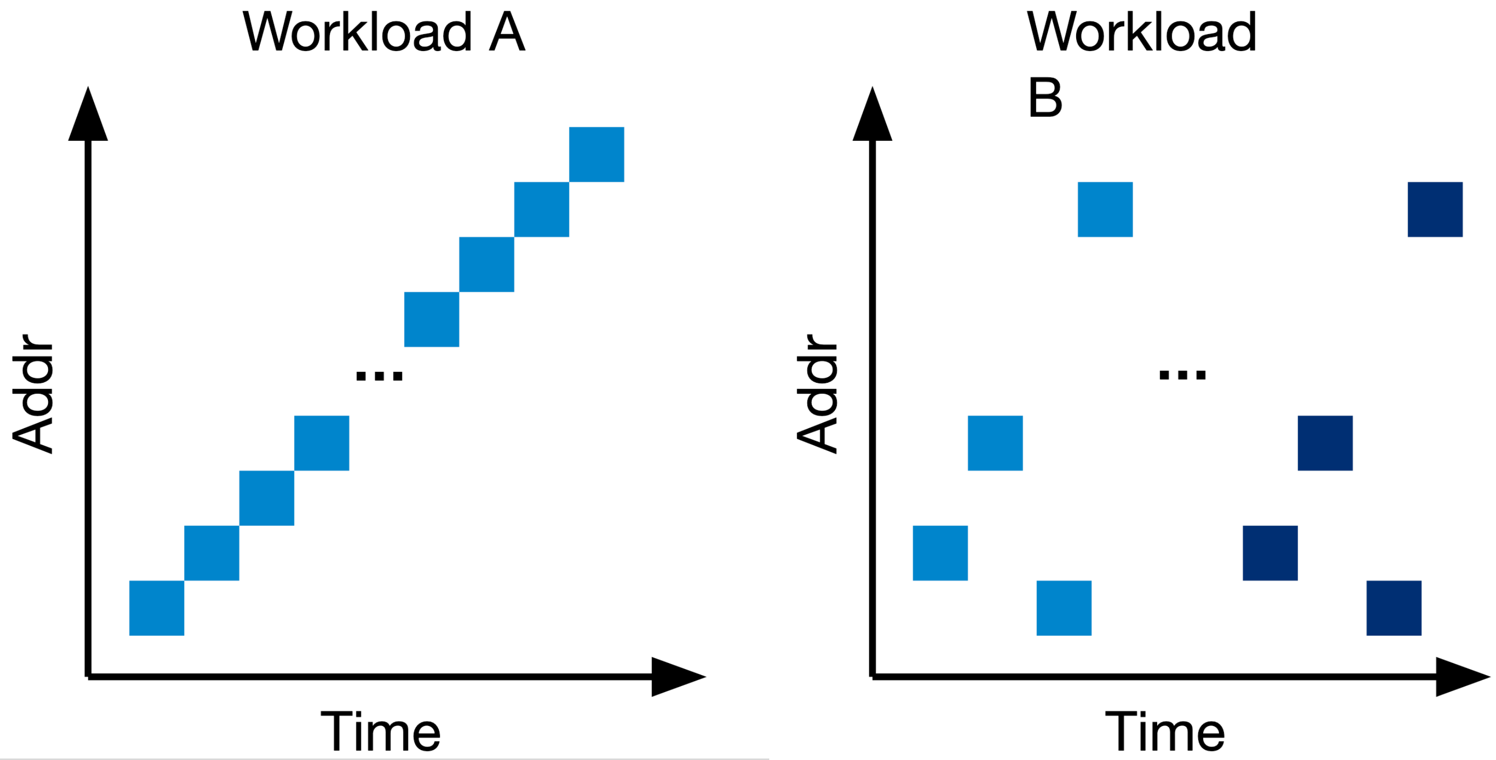
What is the difference between the two workloads?





- **Spatial** locality: addresses are close to each other in “space”
- **Temporal** locality: addresses are close to each other in “time”

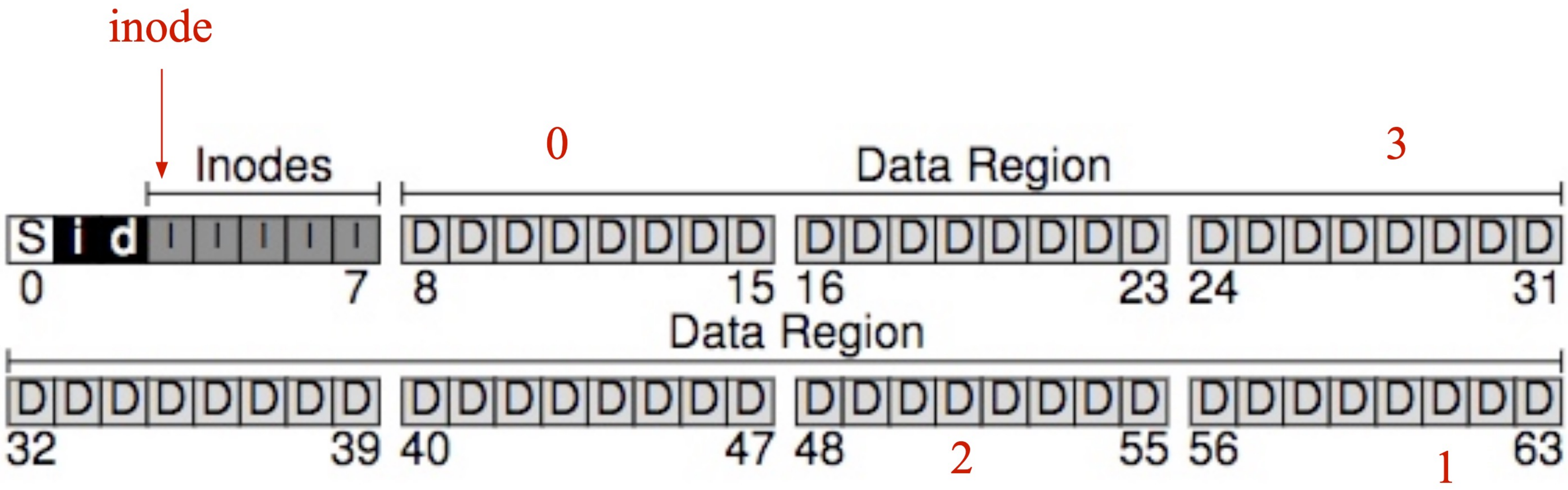
Which of the following is actually better for disks?



➢ **Spatial** locality: addresses are close to each other in “space”

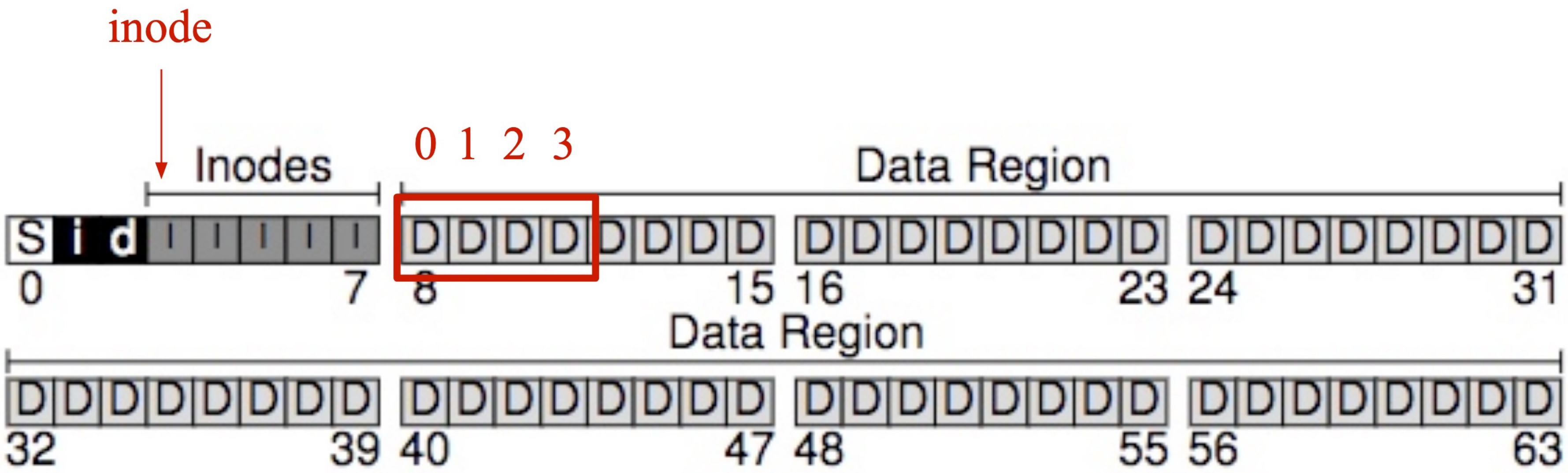
➢ **Temporal** locality: addresses are close to each other in “time”

Imagine that we must choose inode and data blocks



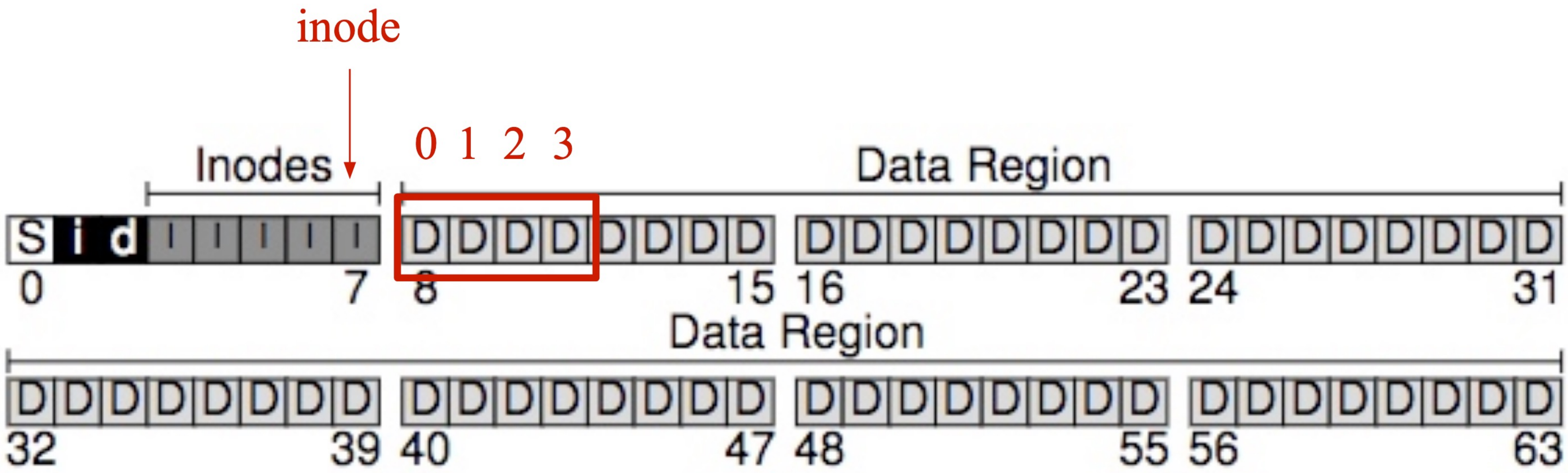
- **Is this a good or a bad policy to choose blocks?**
  - Bad policy; will require slow "random" access

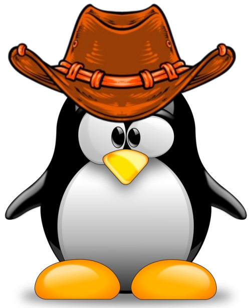
A better policy would be as follows



- **Is there something else that we can optimize?**
  - Location of the inode itself, in this particular case. ☺

An even better policy





FS optimization: buffering in memory

Avoiding excessive I/O operations is important

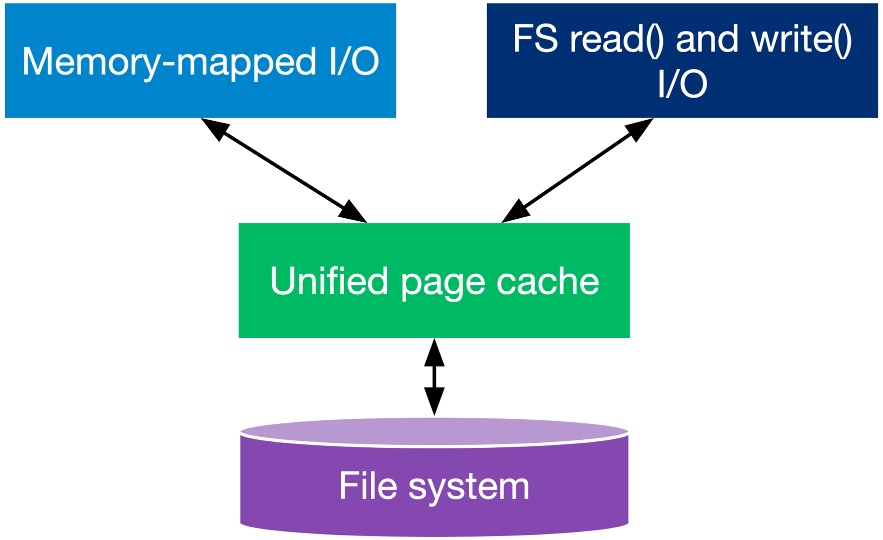
- Even with all possible disk optimizations, I/O remains slow and unpredictable depending on workloads

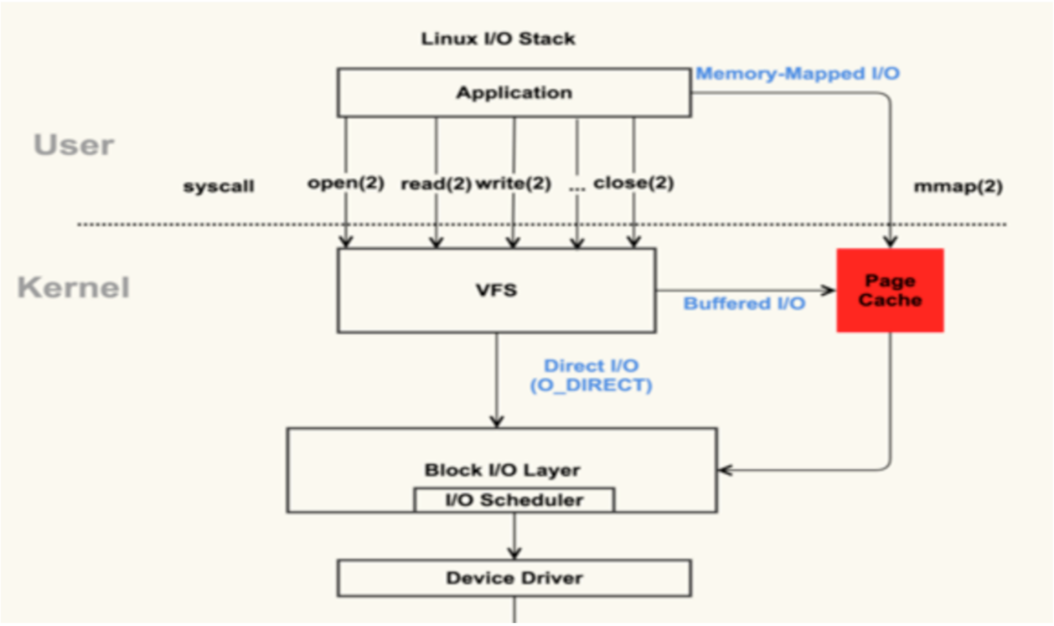
- File system will cache “files” in memory (recall how the xv6 block layer caches “blocks”) to improve performance

- To do so, the file system will:
  - Reserve a region for the “file cache”
  - Find out which files are frequently read/written
  - Keep those files within the cache

Modern OSs take a “unified” cache approach

- Keep a single cache for (a) file pages and (b) program memory pages
- Avoids having multiple caches, and separately dealing with them





Buffering creates a tiny headache for the file system..

- Imagine the following set of events from an FS:

    1. Program writes to a new 1G file (e.g., bar)
    2. FS caches all writes to the file in-memory
    3. FS updates the inodes within the disk
    4. FS writes all changes to data blocks within disk (e.g., block-by-block using Linux's block I/O)

- **Can you spot potential problems?**
    - If power goes off after 3, your system has already reserved the data block → FS lost a few free data blocks

Worse problems could also occur on crashes

- Imagine the following set of events from an FS:

  1. Imagine that the FS is deleting a file X:
  2. Frees block for X and changes its data bitmap
  3. Before FS can delete the *inode* for the block, power goes off.
  4. OS restarts; thinks block is free and allocates it to a new file Y.

- **Can you spot potential problems?**
  - Two inodes are pointing to the same data blocks → write to two different files will overwrite each other

- **Can this be a security issue?**
  - Yes, imagine two different users get allocated the blocks

Crash recovery is a significant problem for FSs

- Many filesystem operations require multiple writes to disk
  - creat (file) → 5 disk blocks to be written
    - inode, directory, and actual file data
- A crash (e.g., power failure) after a subset of these writes might put the filesystem in an inconsistent state
- **Note this is only a problem for write operations. Why is that?**
  - ➢ Data/Inode blocks only get changed on writes, not reads.

How would you solve this problem of FS corruption on crashes?

- Imagine the following set of events from an FS:

  1. Program writes to a new 1G file (e.g., bar)
  2. FS caches all writes to the file in-memory
  3. FS updates the inodes within the disk
  4. FS writes all changes to data blocks within disk (e.g., block-by-block using Linux's block I/O)

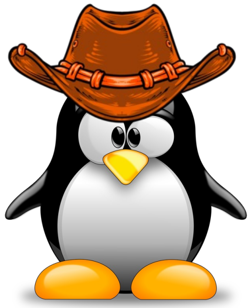
- **Can you spot potential problems?**
  - If power goes off after 3, your system has already reserved the data block → FS lost a few free data blocks

How would you solve this problem of FS corruption on crashes?

- Imagine the following set of events from an FS:

  1. Imagine that the FS is deleting a file X:
  2. Frees block for X and changes its data bitmap
  3. Before FS can delete the *inode* for the block, power goes off.
  4. OS restarts; thinks block is free and allocates it to a new file Y.

- **Can you spot potential problems?**
  - ➢ Two inodes are pointing to the same data blocks → write to two different files will overwrite each other



Ensuring correctness after system crash

*F*ilesystem **logging** addresses crash problems

Let’s look at the high-level idea of logging:

- Never write directly to the on-disk data structures
- Place all writes to a “log” on the disk
- Once all log writes in one operation are complete, issue a “commit”
- FS writes the data to actual disk locations using “log”
- FS removes “log” entry

Does logging prevent corruption on crashes?

Let's look at the high-level idea of logging:

Discard all writes on reboot after crash here

- Never write directly to the on-disk data structures
- Place all writes to a "log" on the disk

All needed data is in disk to complete write

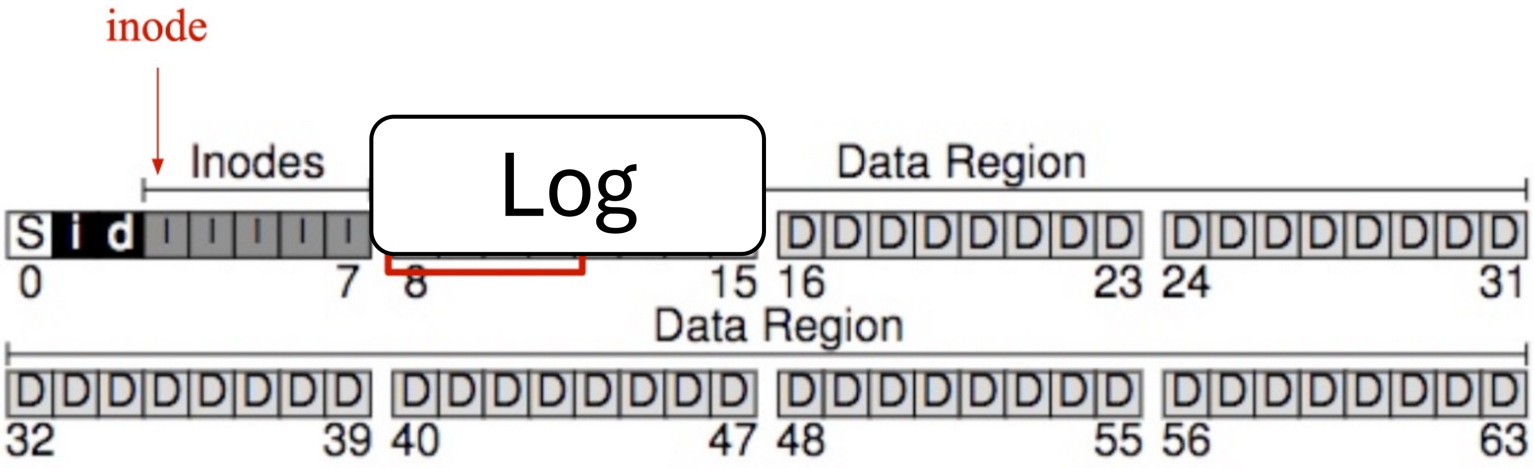
- Once all log writes in one operation are complete, issue a "commit"

Can recover the data using log on crash here

- FS writes the data to actual disk locations using "log"
- FS removes "log" entry

Ensures atomicity: all writes in a transaction are logged or none

Where is the log in this figure BTW?



- ➢ We must reserve it as well in some fixed part of the disk, and note its information (e.g., starting block, size) in the superblock

Code review: xv6's crash recovery through logging

Step #1: Signaling start of log at the write system call

Step #2: Writing data to in-memory buffers

Step #3: Tracking the buffers that must be flushed to disk

Step #4: Commit cache to log disk sectors

Step #5: Moving data from log to final disk locations

Step #1: Signaling the start of a log at `write` syscall

write(..) system call

```
int
filewrite(struct file *f, uint64 addr, int n)
{
```

…

```
    begin_op();
    ilock(f->ip);
    if ((r = writei(f->ip, 1, addr + i, f->off, n1)) > 0)
      f->off += r;
    iunlock(f->ip);
    end_op();
```

Signalling starts here

Perform the writes now

Called at the end

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

Is log being committed?

Is there enough space?

Step #2: Writing data to in-memory buffers

```
int
writei(struct inode *ip, int user_src, uint64 src, uint off, uint n)
```

Called from filewrite()

…

```
for(tot=0; tot<n; tot+=m, off+=m, src+=m){
  uint addr = bmap(ip, off/BSIZE);
  if(addr == 0)
    break;
  bp = bread(ip->dev, addr);
  m = min(n - tot, BSIZE - off%BSIZE);
  if(either_copyin(bp->data + (off % BSIZE), user_src, src, m) == -1) {
    brelse(bp);
    break;
  }
  log_write(bp);
  brelse(bp);
}
```

Try to find a cache to keep block in-memory

Read block from disk

Copy data from userspace

Similar to the copy_from_user(..) you have been using for kernel modules

Step #3: Tracking that a buffer must be written later

```c
int
writei(struct inode *ip, int user_src, uint64 src, uint off, uint n)
```

```c
  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    uint addr = bmap(ip, off/BSIZE);
    if(addr == 0)
      break;
    bp = bread(ip->dev, addr);
    m = min(n - tot, BSIZE - off%BSIZE);
    if(either_copyin(bp->data + (off % BSIZE), user_src, src, m) == -1) {
      brelse(bp);
      break;
    }
    log_write(bp);
    brelse(bp);
  }
```

Must keep metadata for log

Log metadata structures

struct log {
struct logheader;
…
}

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

log_write(..

Reuse if already allocated

```c
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno)   // log absorption
      break;
  }
  log.lh.block[i] = b->blockno;
```

Place each block into an empty log block

Step #4: Commit cache to log sectors

writei(..) → *filewrite(..)* → *end_op()*

```
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(0); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}
```

Write from cache to log

Not actually 'committed' until we save log header

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

Read log header from disk

Change, then write-back

Is the log actually 'committed' at this point?

Step #5: Write cache to disk and delete log entry

writei(..) → *filewrite(..)* → *end_op()*

```c
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(0); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}
```

Remove the log to avoid re-copy on crash

```c
// Copy committed blocks from log to their home location
static void
install_trans(int recovering)
{
  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
    if(recovering == 0)
      bunpin(dbuf);
    brelse(lbuf);
    brelse(dbuf);
  }
```

Read log src and disk dst blocks

Write-back to dst

Is the store to disk "actually committed" at this point?

Aspects of FS logging can become challenging

- All data at system call is first written to in-memory block cache
- Then transferred to the log disk space inside your disk
- Also, FS system calls can happen concurrently

Can you spot the major challenges to ensure smoothness?

Challenges of logging?

- **System call data must fit inside the log disk space**
  - Recall that each write(..) puts all blocks into log regions
- xv6’s solutions:
  1. Set an upper bound of all log entries
     - Set log size >= upper bound
  2. Breakdown large writes into smaller transactions
     - Problem: large writes are thus not atomic
     - However, overall system remains in a consistent state

Challenges of logging?

- **Concurrent system call handling**

  1. Must allow different FS system calls from different proc at the same time
  2. Block may be written multiple times while still in-memory
     - Creating a new log entry for each time is wasteful

- xv6's solutions:

  1. Check if a system call's data can fit in log before starting it
     - Else, sleep and wait for log to free up
  2. If the block is already assigned a log entry, return without creating new entry
     - Also called "write absorption" → absorb multiple writes into a single

Pros of xv6's logging?

- FS internal invariants are maintained
  - Metadata remains consistent (e.g., no two inodes point to same blocks)
- Except for the last few operations, data is preserved on the disk
  - Every system call is written at the end before return
  - No surprises of losing data written *way back*
- Write order is preserved
  - $ echo A > X ; echo B > Y
  - X will be written before Y

Cons of xv6's crash recovery method?

- Several inefficiencies

Design choice: ensure either old or new copy is correct. No corruption

- Every block is written twice (once for log + once for final)

- Eagerly writes to disk after every write system call
- Writes each block one-by-one to disk

Implementation choice: *How would you overcome?*

Keep data in-memory for longer periods and batch disk writes

Ordered mode in ext3/4 FS avoids twice data writes

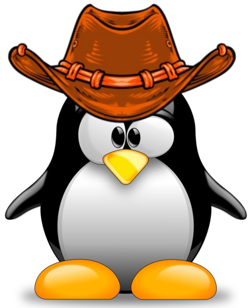
- Log the FS metadata but write data directly to the final disk location
  - Crash **consistency**, but **not recovery**

- High-level overview of ordered mode:
  1. Metadata is committed to log
  2. Data is written to final disk location
  3. Metadata is moved from log to final disk location

- Limitation?

- Suggested reading(s):

Journaling the Linux ext2fs: https://pdos.csail.mit.edu/6.1810/2022/readings/journal-design.pdf
Analysis and Evolution of Journaling File Systems (Prabhakaran et. al)



Questions? Otherwise, see you next class!