large sequential writes: append updates the log 

\[ \text{segments written after checkpoint} \]

\[ \text{log} \]

\[ \downarrow \]

\[ \text{checkpoint region (fixed loc)} \]

\[ \text{loc of inode map pieces at time of checkpoint} \]

\[ \text{tracks last checkpointed segment} \]
How do we update the checkpoint Region?

→ CR spans over multiple blocks, update must be atomic!
  
  → txn-begin [timestamp]
  → CR blocks
  → txn-commit [timestamp]

→ 2CRs, toggle update
  → both valid, pick newer one
  → only 1 valid, pick the valid one

→ is it possible for both to be invalid?

→ CR spans over multiple blocks, update must be atomic!

→ upon a crash, if begin & commit has matching timestamp, the tx is valid! Otherwise, CR is invalid.

→ 2CRs, toggle update
  → both valid, pick newer one
  → only 1 valid, pick the valid one

→ is it possible for both to be invalid?

→ recall that CR is written at an infrequent interval ⇒ may recover consistent but stale fs state!

→ roll forward by apply valid segments past the checkpoint
Garbage Collection

→ Segment
  ↓
  some blocks are live & some are garbage
  each has a segment summary (sometimes multiple...)
  tracking each data block's inode # & offset

→ compact live blocks
within multiple segments,
write into a new segment!

→ threshold for compaction
  if 90% are live, probably shouldn't compact

→ hot vs. cold segment
  some data might be updated more frequently,
delay compaction could see more garbage.
one more can fs = ZFS

Root Inode  Root Inode’s Indirect Blocks  Inode File  File’s Indirect Blocks  File’s Data Blocks

until a new root blk is written to point to new sets of updated blocks, no change is observed by the filesys.

→ actually carries out recursive update (no logical ptr like LFS’s indemap)
Updates appear atomically after the new root inode becomes active.

Isn't this a lot of disk writes for just adding one new data block?

Buffer more updates in memory to amortize the cost of path reunite!

Also supports logging for perf.

→ Logical logging (only log operations, not changed blocks)
User Level Threads

- kernel threads (TCB)
  - pthread API, managed & scheduled by kernel (kernel -> user)
  - creation => system call, every context switch involves a mode switch
- MB of stack
- user threads
  - managed & created via user libraries & runtimes
  - Smaller / adaptive stack size
  - every context switch & creation is just a procedure call

N user threads

Kernel thread

> own stack
> own ripl-func
> own stack (reg & TCB)

N kernel threads

Kernel thread

user lib

Kernel thread

Kernel thread

Kernel thread

Kernel thread