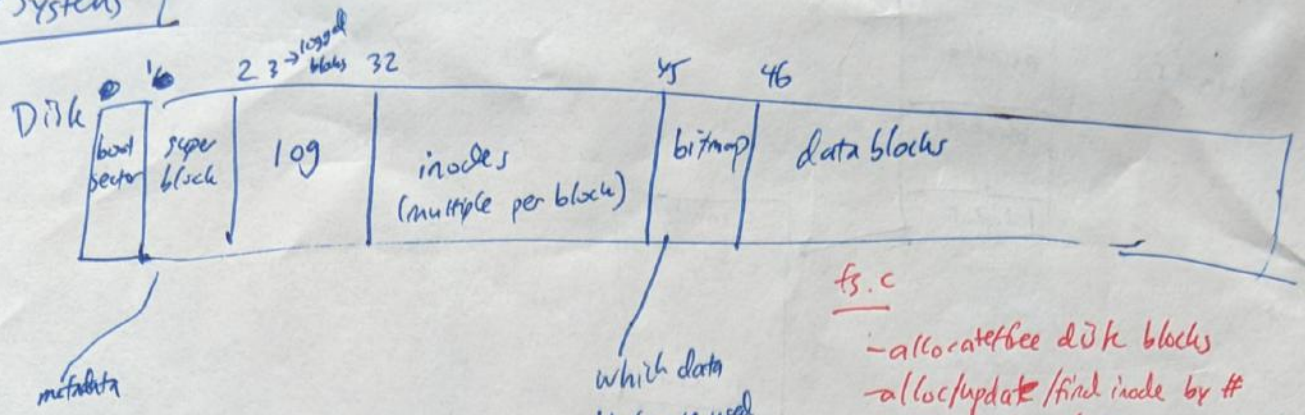
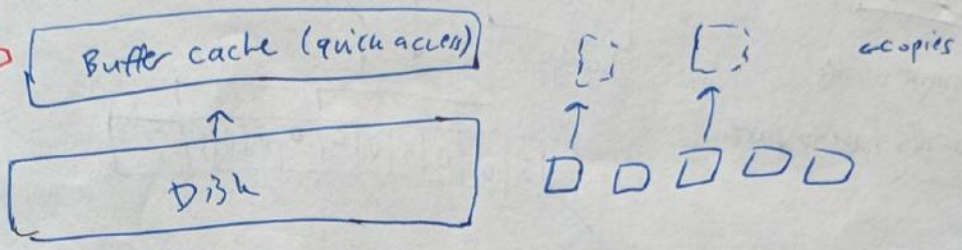
File Systems 1



fs.c
- allocate/free disk blocks
- alloc/update/find inode by #
- managing locks/references to inodes
- map blocks to inodes

bio.c
- binit creates linked list of buffers
- each buf has a blockno, lock
- bget is called to get data
- brelse - releases a locked buffer
- bwrite: write changed data to disk before releasing buffer
- balloc - alloc new disk block



PROTECTIONS: IFF sb→size b. bitmap is OK
* bcache.lock protects cache description
* b→lock protects one buffer
* b→refcnt prevents buf from being recycled

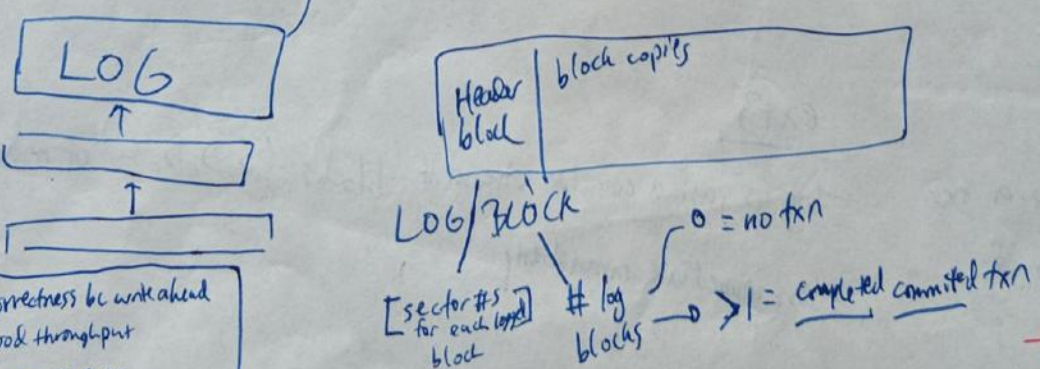
Ex → write data to new file
1) write to block 33 for allocating new inode
2) write to block 46 for writing directory entry
3) write to block 32 - update directory inode
4) write b 45 to alloc new block
5) write to b 524 for the new block
6) write block 33 for adding new block to inode

alloc
write
alloc
write
write

---

xv6 logging

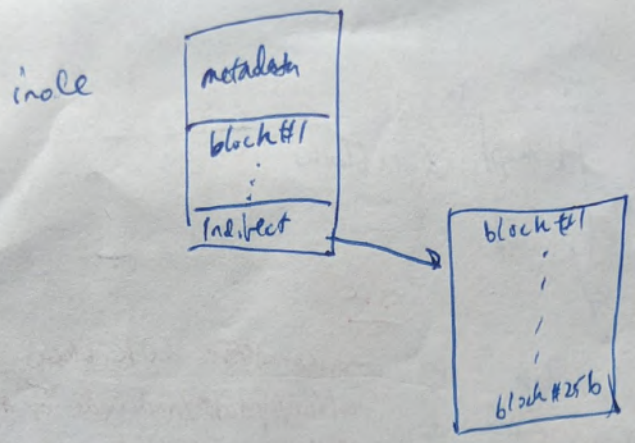
→ description of all disk writes it wishes to make
→ writes commit record for each complete operation



PROS - correctness b/c write ahead
- good throughput

CONS - blocks written twice
- ops can't fit in log?

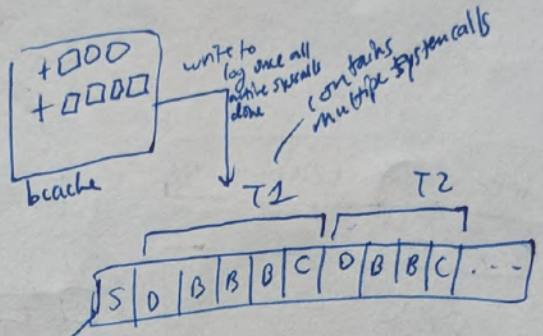
log.c
- log_write writes to a buffer in the buf cache and pins it and appends block # to sector arr
- commit()
  - write modified blocks to log from buf
  - write header (COMMIT PT)
  - install writes to actual loc
  - erase txn
- begin/end op



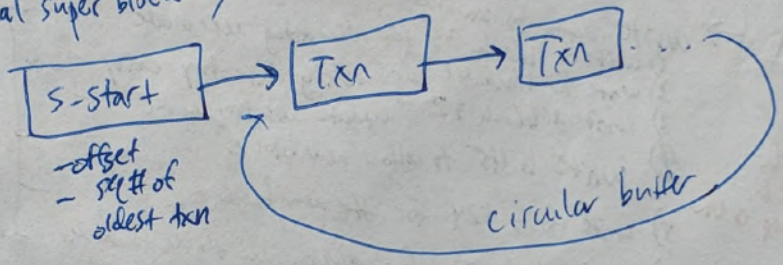
Problem w/ Logging SLOW

→ 2x commit writes

→ new syscalls have to wait



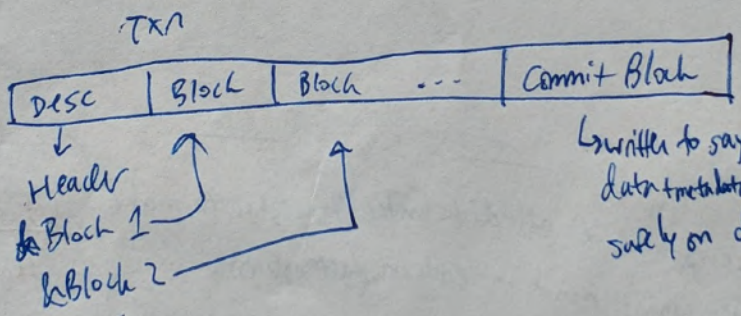
Journal super block



~~offset~~
- seq # of oldest txn

CRASH

1) Start from oldest seq #
2) Find end of log
3) Scan until missing commit or unexpected seq #
4) Return to last valid commit block, replay all blocks through last valid commit



↳ written to say data + metadata are safely on disk

Differences

xv6

- every syscall is a txn
- log is an array of blocks / txn

ext3

- log is just a circular stream of blocks (S, D, B, C) of multiple txns
- less immediate availability

VM optimizations — user controlled virtual memory (faster)

TRAP - handle pg fault traps sigaction

PROT1 - decrease accessibility of a page mprotect

PROTN - decrease avail of N pages (mark os vm structs as protected) mprotect

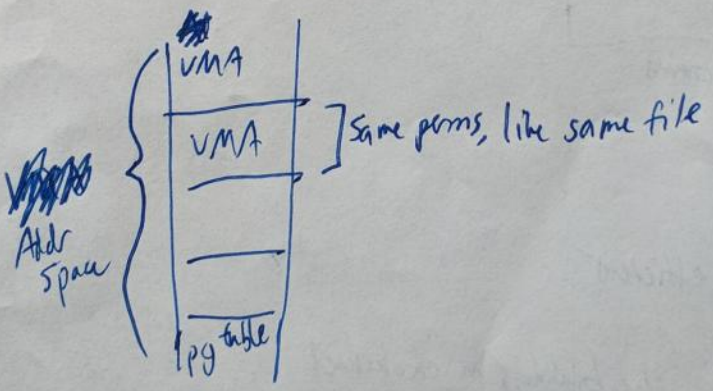
UNPROT - increase accessibility mprotect

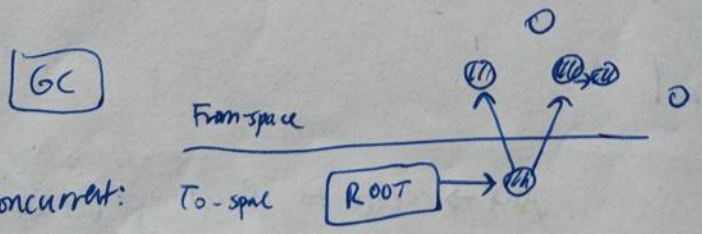
DIRTY - returns list of dirtied pages

MAP2 - map phys page at 2 diff VAs, protect levels

mmap()
- map mem into addr space
- mprotect (PROT_READ, write)
- munmap

Process:





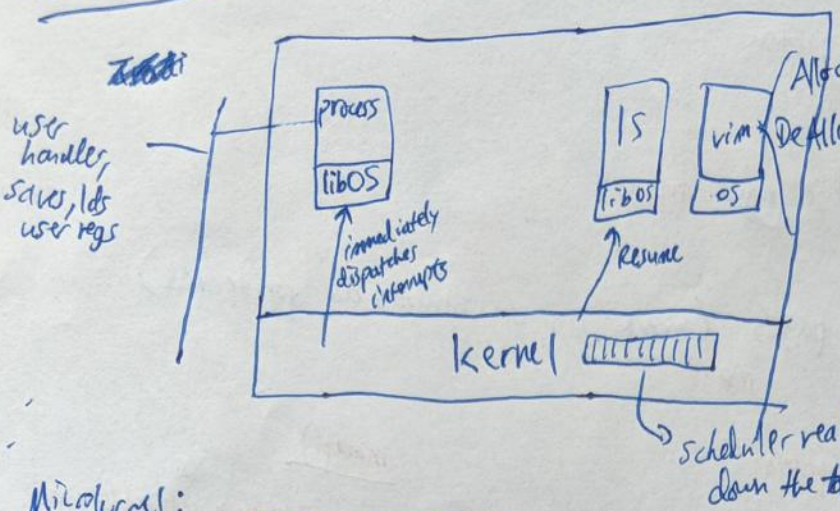
How to avoid race conditions?

* Split To-space into Scanned R|W
  Unscanned fault on access

read protect: pgfault, GC immediately scans, all its neighbors, UNPROT

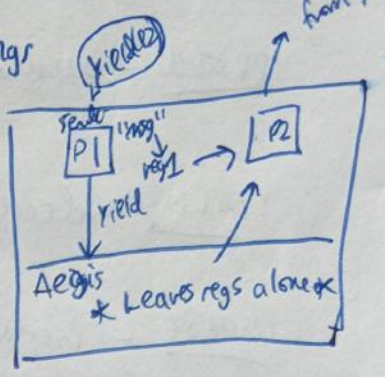
Generational: separate by how recently they were alloc'd
- youngest Gen is more often looked at
- also more ptrs around in space

EXOKERNEL - How abstracted does a CPU need to be?

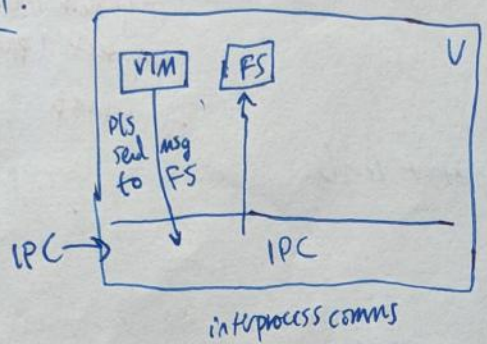


* libOS goes to do whatever with the interrupt signals
AllocPage
DeAllocPage

* secure bindings



Microkernel:



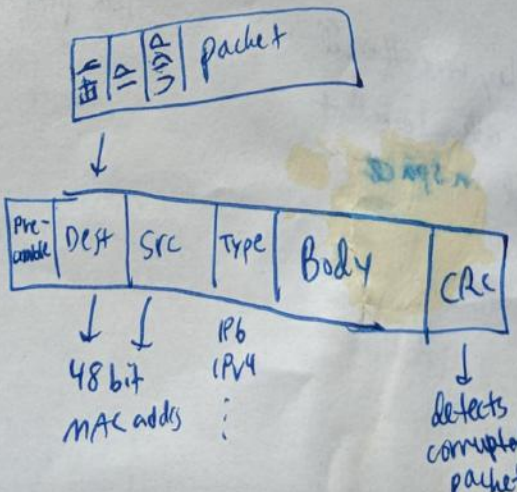
Revocation

- visible revocation - libOS guides process

Secure bindings

- Hardware: low level, efficient
- Software caching - cache bindings in exokernel
- downloading code into kernel

Networking

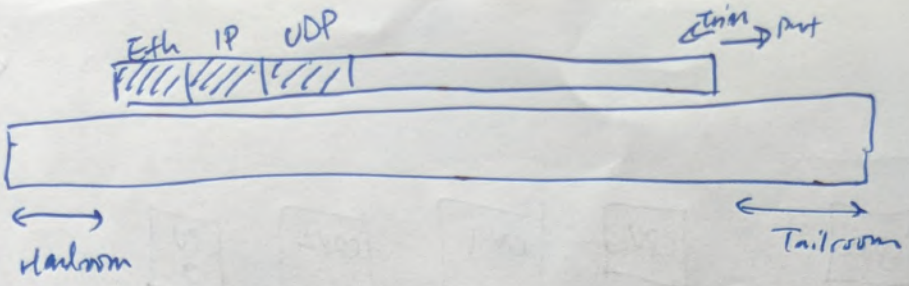


ARP → send to IP addr ... but which MAC

if in same network:
use MAC addr
else:
use MAC of gateway
- Gateway receives, broadcasts request for src MAC/IP, dest IP
- Target responds w/ MAC

datagram (UDP)	stream (TCP)
- note passing	phone call
- order, data guarantee not preserved	- reliable
- faster (less overhead)	- order
	- slower

Buf:

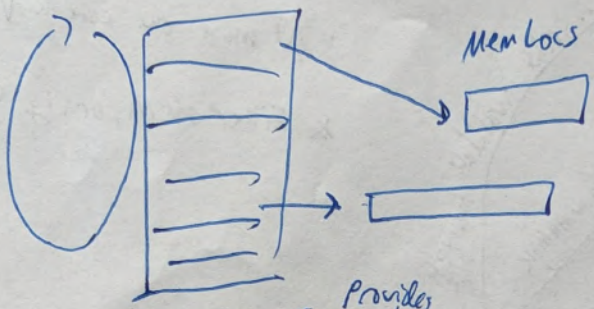


Sockets

- Datagram sockets (unreliable)
- Stream sockets (bi-directional pipes)

ex receive ring empty? DROP

ex transmit ring empty? wasted bandwidth



NIC
descriptors
flags

Provides locs

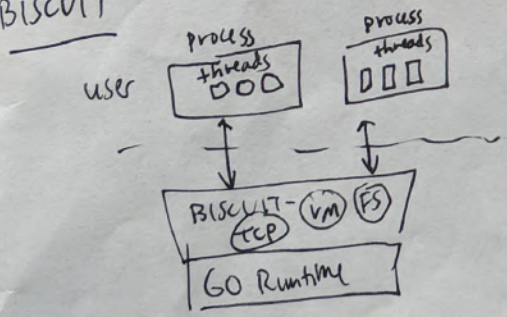
OS → must constantly poll or interrupt for new packets

- predictable tasks
- unnecessary waiting

- unpredictable tasks
- wasteful when frequent

provides notif of finishing

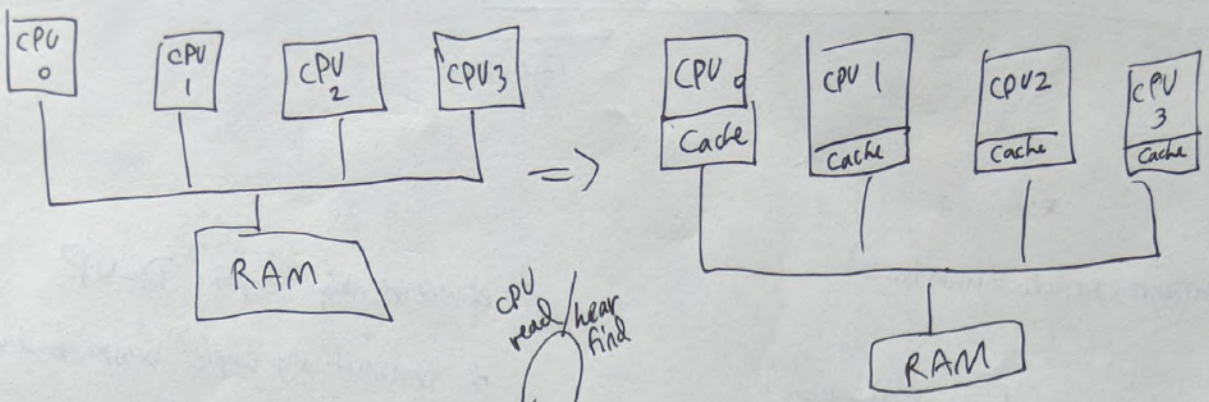
BISCUIT

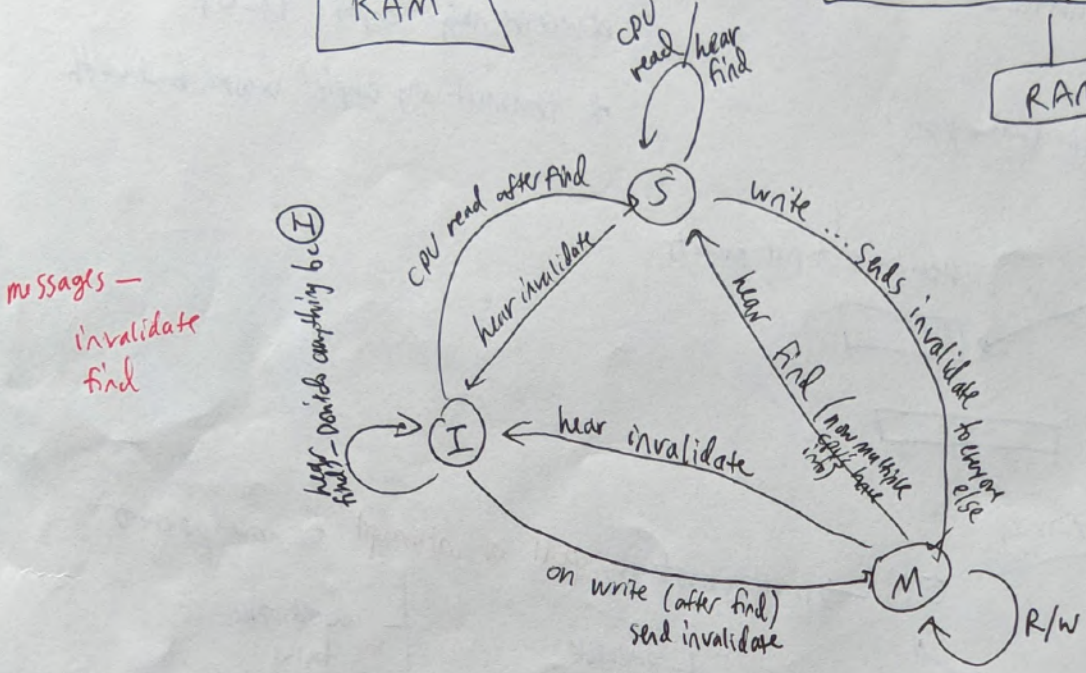


HLL	C
-automatic mem mgmt	-no implicit code
-GC	-flexibility
-less use-after-free bugs	-control of mem/alloc/free
-poor performance	-hard to write
	-RCU error prone

GC makes this way easier

SCALABLE LOCKS





messages —
invalidate
find

* At most one core in M
* Either one M, or 1+ S

MCS Locks

CPU requestor: q node {
locked: T/F (have lock or nah)
*next – next person to have the queue.
}

1) Lock → empty node NULL

2) Lock → My qnode (running!)
- locked = F
- next = NULL

3) Lock → My qnode
- locked = F
- next → Other qnode (waiting, spinning on locked)
- locked = T
- next → NULL

RCU Usage

Reader – getval

```
rcu_read_lock()
p = rcu_dereference(item)   // read
:
: ops
:
rcu_read_unlock()
```

prevent interrupts (disable)

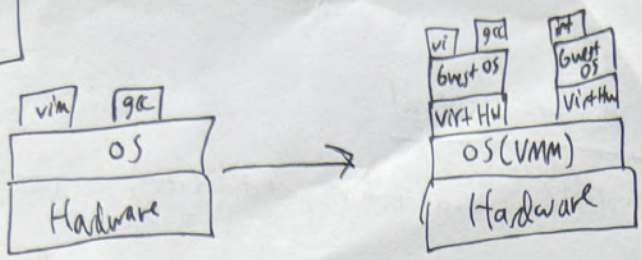
* no concurrent writes!

* Read doesn't lock.

Writer – setval

```
spinlock
oldp = rcu_dereference_protected(item)   – mem barrier
:
: ops, alloc newp, copy, set vals
:
rcu_assign_pointer(item, newp)   – mem barrier
spinunlock
rcu_sync()
kfree(oldp)
```

VMs



Before: Host VA → (MMU) → Host PA

Now: guest: Guest VA → (Guest PT) → Guest PA → (VMM Map) → Host PA

* maps guest PAs → Host PAs
→ this + guest PT helps VMM post to shadow pg table

*on reads, VMM will translate guest PAs to host PAs.

Actual: Guest VA → (Shadow pg table) → Host PA

*on writes, guest OS is fine, but real MMU traps bc no mapping. VMM handles this and puts a mapping in shadow page table, or the actual table.

* 1 shadow page table / VM

memory tracing → * Guest PTs are read-only. Writes fault. Traps to VMM