Performance Issues

P1: write(fd, "hello", 5)

read(fd, buf, 5) ← in kernel, looks at cache

you can lose data written just before power failure

cache is volatile, if the computer get turned off while it is in here it will lose what it just did

Issue: how do you know if the last write was actually processed or there maybe it did not lose the last write because it has not been processed yet

- How do you know if it is processed? User has to be notified somehow.

P1: 2 writes at the same time due to the circular elevator algorithm

- if the power is pulled then one write, which was written before the second write may not be written

1 way to fix: complicate the API

int fsync(int fd); ← this make sure that for all cache, this file is written & not just the cache
will return a value to ensure that it was written to the file

void sync(void); ← Schedules writes for all cash

if(fsync(fd)!=0) error(); ← output offset is block aligned?

if(write(fd, buff, 8192)!=8192) error().

if(fsync(fd)!=0) error()

- has to write to 2 blocks - to the file and has to change the inode data (need to change the size and the data)

Solutions: choose not to change the inode data? ~ thus only writing to one block

int fdatasync(int);
???

int close(fd); ~ no insurance that the information was written to cache, to disk, or to flash
a file could close and then another program opens it

Similar issue w/ rename("a", "b")

1. get working directory block into Ram
3) Eventually cache written to disk

2) get directory's data into RAM

problem:
2 renames called ~ may be only 1 rename actually gets done

rename ("x/a", "y/b")

rename("a", "b")

solution: dfd = open ("x", O_RDONLY)

fdatasync(dfd)

issue with 2 hard links to 1 file: the never could get off after the second name is hard linked to the same file but before the link count is updated ~ thus it says there is only 1 link & when one file is removed all the data is removed even though the other link actually links to nothing

Solution: C. read inode

1. read x's data
2. read y's data
3. write inode links count=2
4. write y's inode data
5. write x's inode data
6. write inode again (link count)

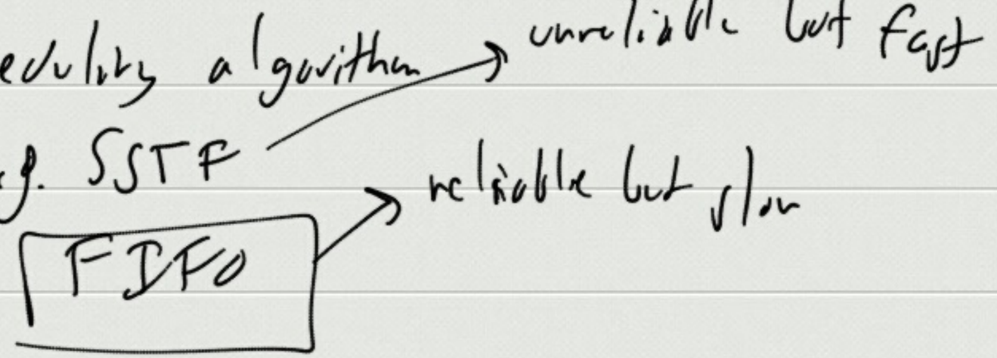
fsck (on an unmounted file system)

(/lost+found

→ after reboot

Interaction between scheduling & robustness

set of pending I/O requests
read block #9312
write block 497162(" ")



Careful ordering of I/O ~ choose to write in the fastest order possible

(3) write 39216
~~(2)~~ write 2196 ← block link count
(2) write 37215
(1) write 2195 ← block link count

Compromise

* we want a compromise between SSTF & FIFO
  1) OK to re-order data blocks (assume no fsync, fdatasync)
  2) remember dependencies in non-data blocks
     low level system must respect dependencies

Robustness terminology & theory

3 main goals of file system
1. durability - survive failures in underlying hardware
   e.g losing power
2. Atomicity - changes either all made or not made at all
3. Performance = throughput & latency

How to implement atomicity atop a device where writes aren't atomic

timeline ⟶

"A" "B"

no write("B") non atomic

void awrite(char x)
  lock all reads
  write x
  unlock ...

Golden Rule of ATOMICITY: Never overwrite the last copy of your data

always have the last 2

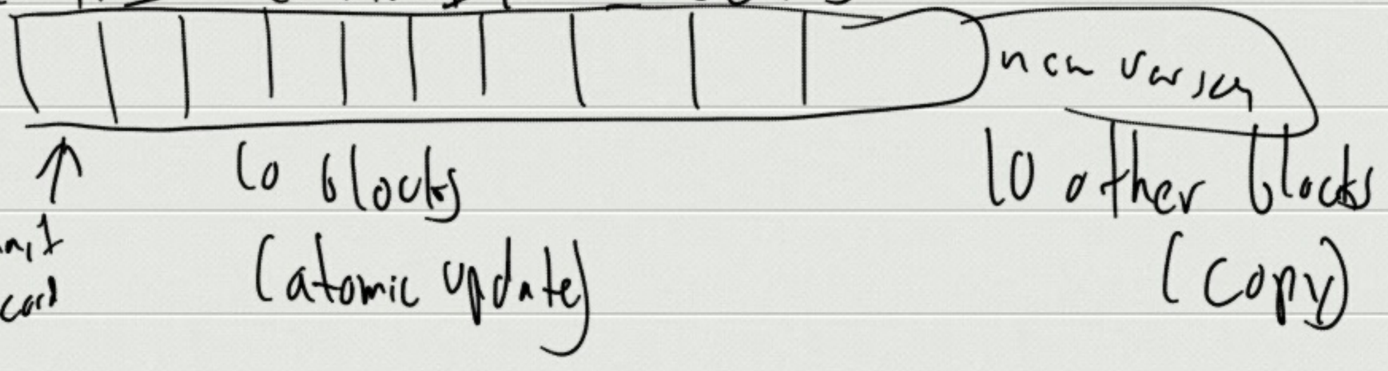
You need 3 copies

I	A	?	B	B	B	B	B
II	A	A	A	?	B	B	B
	A	A	A	A	A	?	B

Lampson-Sturgis assumptions

1. storage writes may fail or may corrupt other blocks
2. a later read can detect the bad blocks
3. Storage can spontaneously decay
4. Errors are rare ~ if everything crashes and all storage spontaneously decays than your fucked
5. Repairs can be done in time

Idea #1 COMMIT RECORD



Idea #2 Journal

- make a log of the planned changes
- then you make a commit record

* if there