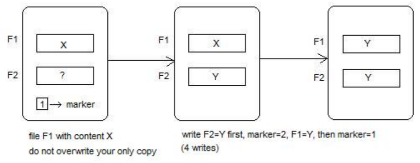
15. File System Robustness

To discuss risks like this, we need to define a few key terms;

- error ~ a mistake by the designer or user
- fault ~ a latent problem in design
- failure ~ the problem caused when the fault trap springs
- durability ~ the system's ability for data to survive limited failure
- atomicity
  a. All-or-Nothing Atomicity
     i. from the point of view of a function's invoker, the sequence either:
        1. Completes
        2. Aborts such that it appears the action was never started (**<u>back out</u>**)
     ii. How can we give the READ function AoN atomicity?
        1. Blocking Read: wait & sets return address before READ
        2. Non-Blocking Read: kernel returns {} if stream is empty
        3. Non-Atomic: READ waits and blocks until char is delivered
  b. Before-or-After Atomicity
     i. from the view of a function's invokers, the result is the same as if the actions occurred completely before or completely after one another
     ii. If two actions have before or after atomicity, they are **<u>serializable</u>**:
        1. There exists some serial order of those concurrent actions that would, if followed, lead to the same ending state
  - Additionally some applications require **<u>sequential/external time consistency</u>**:
  - If it appears to the outside that the events occurred in a certain order, the correct result is as if they were executed in this order

So take a test editor, say EMACS, and assume it is writing to blocks
What happens if the power goes out while we are overwriting our data?
This leads us to our Golden Rule of Atomicity: never delete your only copy

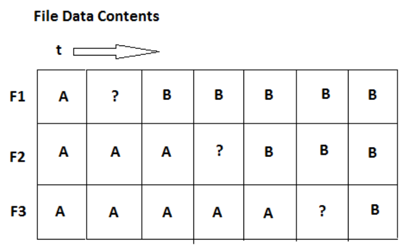

F1
F2
X
?
1 → marker
F1
F2
X
Y
F1
F2
Y
Y
file F1 with content X
do not overwrite your only copy
write F2=Y first, marker=2, F1=Y, then marker=1
(4 writes)

Our first attempt at atomicity: we create a new file and write the new data there; on completion we switch which is the active file.

BUT what if we lost power mid-swap? We wouldn't know which is the active data?

~ we can solve this in a probabilistic sense with checksums post fail

BUT what if it a block write isn't atomic? Write from A to B is 3 stages: A → ? → B
~ we can make an atomic write using 3 blocks!


File Data Contents

t →

F1	A	?	B	B	B	B	B
F2	A	A	A	?	B	B	B
F3	A	A	A	A	A	?	B

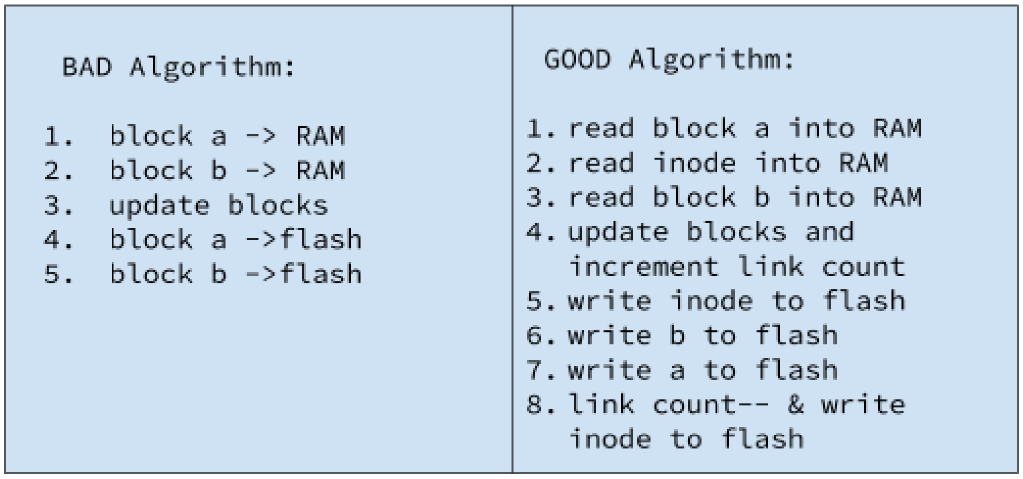
Algorithm:
- write 3 in series
- use best 2/3 to choose
- choose block 0 if they all differ

Since we have atomic block write, we can discuss file system robustness on a larger scale:
We use a **Lampion-Sturgis** failure model:
i. storage writes may fail
ii. storage writes may corrupt other blocks
iii. storage blocks may decay spontaneously
iv. a read can detect a bad block (using checksums)
v. errors are rare
vi. reports can be done in time

We also need to establish a few **file system robustness invariants**:
i. every block serves at most one purpose
   1. failure allows a program to overwrite another program's data
ii. all referenced blocks are properly initialized
   1. failure allows an uninitialized block to look like a pointer to a random block
iii. all referenced blocks are marked as "used"
   1. failure allows data to be overwritten
iv. all non-referenced blocks are marked as "free"
   1. failure leaves an unused block marked as used — data leak

The failure of most of these would mean the loss of data—except number 4
→ our failure hinges on number 4

Take a risky operation...say rename("d/a", "e/b")
- fsck() catches errors, but it is insanely slow!
  - It prioritizes the inode data & moves unreferenced inodes to a lost & found
  - File permissions are set to kernel only



```
BAD Algorithm:

1.  block a -> RAM
2.  block b -> RAM
3.  update blocks
4.  block a ->flash
5.  block b ->flash
```

```
GOOD Algorithm:

1. read block a into RAM
2. read inode into RAM
3. read block b into RAM
4. update blocks and
   increment link count
5. write inode to flash
6. write b to flash
7. write a to flash
8. link count-- & write
   inode to flash
```

BAD:

- Failure between 4 & 5 would lose data!

GOOD:

- Instruction (5-6): only old link exists but lc = 2
- Instruction (6-7): the old and new copy exist and lc = 2
- Instruction (7-8): only new link exists but lc = 2

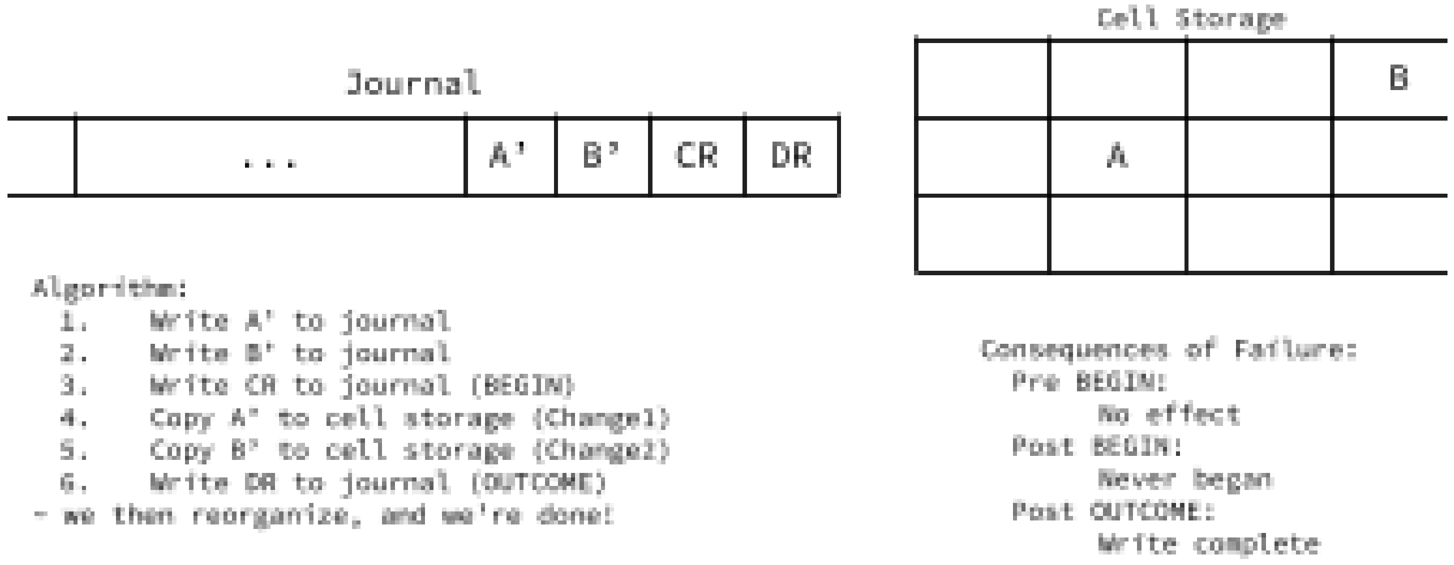
We would prefer to lose space over data!

We can now look at abstracting a single block write into multiple blocks:

COMMIT RECORDS

- document writes such that writes are atomic
- put commit records and all writes into a well recognized location

JOURNALING



```
Algorithm:
  1.    Write A' to journal
  2.    Write B' to journal
  3.    Write CR to journal (BEGIN)
  4.    Copy A' to cell storage (Change1)
  5.    Copy B' to cell storage (Change2)
  6.    Write DR to journal (OUTCOME)
- we then reorganize, and we're done!
```

```
Consequences of Failure:
  Pre BEGIN:
       No effect
  Post BEGIN:
       Never began
  Post OUTCOME:
       Write complete
```

- keep cells in RAM & only copy to disk on write
  — Wastes storage and will eventually run off disks
  + Solves many inconsistency problems
  + If mostly writing, avoids seeks

Our recovery strategy must also be idempotent, since we can fail during reboot
~ execution one time is the same as executing any number of times

We have to write each entry twice...what if we do a large write? Failure seems likely
~ this is true, but most apps think between writes, so long writes are rare

What do we do if a directory is replaced with a file & there is a crash?
- Linux uses the idea of "revoke records" to record a negative change in extv4
- Linux also writes

Is this viable for flash?
No — the performance benefit of this is that we ignore seek costs for write, but we know where we are going to write here

We have two major journaling options: (the example uses write-ahead)

Write Ahead (ext4 option)	**Write Behind**
1. log changes in data 2. write changes in data 3. write the commit record 4. write the done record ON FAILURE: • replay commit records DOWNSIDE: • the start of our replay on failure can be hard to find BENEFITS: • more likely to save last action • doesn’t need to keep multiple versions	1. log old data values 2. write commit record 3. write changes in data 4. write the done record ON FAILURE: • undo the changes DOWNSIDE: • we must copy all of our data on each write BENEFITS: • old data is typically cached, so write ahead benefit is small • more conservative – more recovery is possible • faster recovery – we do not have to seek for the end

We need to be able to handle the failure of low level operations

- we describe two interaction processes:
  - cascading aborts
    - if a low level operation fails, the high level one fails
    - very automate-able
    - Example: Write-Ahead Protocol Cascading Abort:
      1. While logging planned writes, error occurs
      2. Abort record
      3. Send cascading aborts to higher level functions.
  - compensating actions
    - if a low level operation fails, the higher one makes up for it
    - very flexible
    - Example: Write-Ahead Protocol Compensating Action:
      1. Log planned writes
      2. Commit record
      3. While writing to disk, error occurs. After reboot compensating actions continue writing to cell memory using data written in the log.

There are many different types of corruption which can occur in a file system:

1. gamma rays can flip a bit
    a. this happens roughly once a week
    b. ECC memory with a parity bit can fix single flips and catch doubles
2. Drive Failure
    a. this includes physical damage to the drive
    b. companies give an annualized fail rate to give % fail chance for a year of operation
    c. we can use SMART to check condition metrics
        i. if we find a bad sector, we have replacement ones
3. User error
    a. ex) rm * .o (remove all files) instead of rm *.o (remove all object files)
4. OER Errors
    a. include configuration errors, which are the majority of errors
    b. application and OS errors also quality

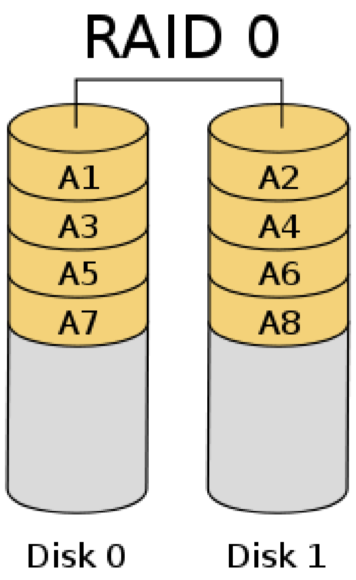
We can catch <u>and</u> fix power failure with our old method
We can catch drive failure with a log structured file system,
    BUT to fix it we need to make a copy

RAID

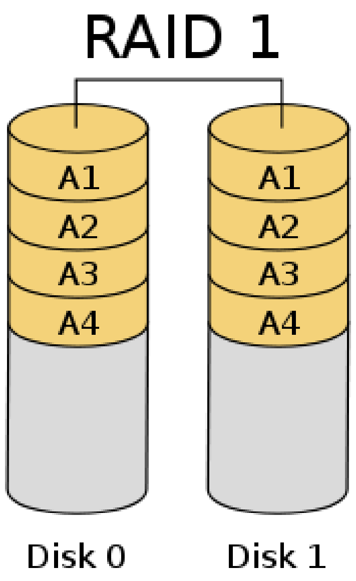
= Redundant Array of Independent Disks

RAID was originally developed to save money by aggregating many small drives into a large one
It is now one of our most useful memory tools!

- we concatenate physical disks to form a large virtual one
- we stripe the data to increase concurrency for read and write



- we mirror half of our disks onto the other half
- we double write speed and half storage
- we can now survive with half our data lost

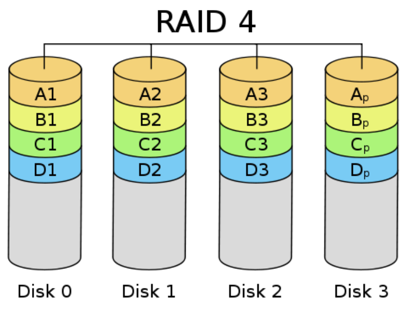


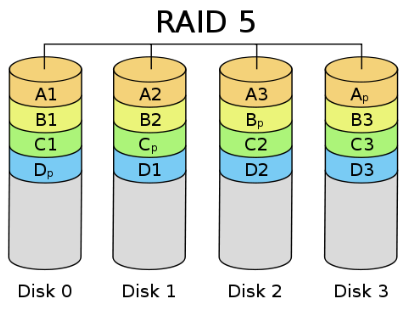
- We reserve one of our disks for parity bits
  - Parity bits are calculated with exclusive-or (^)
- When a drive fails, we go into degraded mode;
  - We especially need to guarantee robustness here
  - We can use a hot spare to replace a corrupt drive
- We can now add drives easily
- We can now recover from a drive loss

- We stripe within a RAID 4 system
- This is faster but more difficult to add drives
- These have almost entirely replaced RAID 4, says the book & eggert, but eggert now seems to disagree

All RAID uses a full-stop model; on detection of an error the operation stops

- it can use checksums that include the data location


RAID 4
A1
B1
C1
D1
A2
B2
C2
D2
A3
B3
C3
D3
Ap
Bp
Cp
Dp
Disk 0
Disk 1
Disk 2
Disk 3


RAID 5
A1
B1
C1
Dp
A2
B2
Cp
D1
A3
Bp
C2
D2
Ap
B3
C3
D3
Disk 0
Disk 1
Disk 2
Disk 3