UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

EXAM 3: REVIEW

Questions answered in this lecture:

What are some useful things to remember about file systems?

ANNOUNCEMENTS

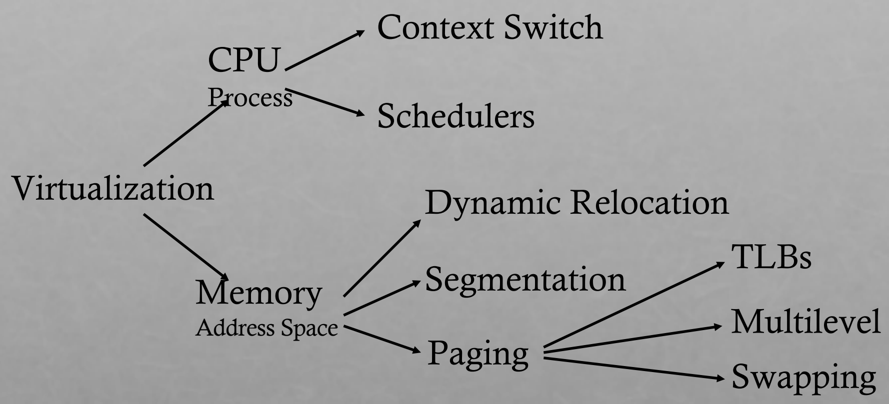
Project 5:

- Extension through Friday at 9pm; NOTHING LATER!

Final Exam – Saturday 10:05am – 12:05pm - Ingraham B10

- Bring #2 pencils and student id; All multiple choice
- Covers everything so far in course:
    - Lectures + Reading + Homework + Projects 1-5
    - 30% Old Material : 10% Virtualization, 20% Concurrency
    - New Material: File systems!
    - Look over sample exams
    - Homework simulations: RAID, VSFS, AFS
    - No question about Physical vs logical journals
- Office hours 10-11 Friday in CS 2310
  (room available for group study til noon)

REVIEW: EASY PIECE 1



Context Switch

CPU
Process

Schedulers

Virtualization

Dynamic Relocation

TLBs

Memory
Address Space

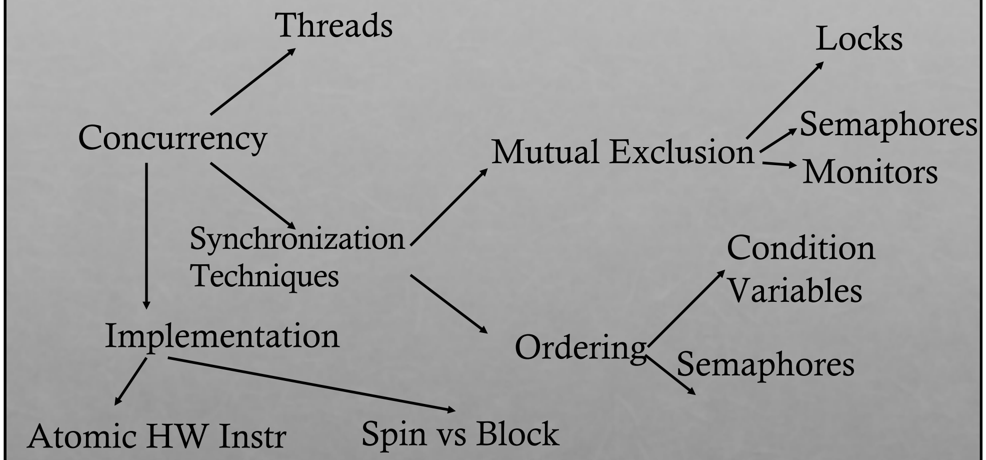
Segmentation

Multilevel

Paging

Swapping

REVIEW: EASY PIECE 2



Threads

Locks

Concurrency

Semaphores

Mutual Exclusion

Monitors

Synchronization
Techniques

Condition
Variables

Implementation

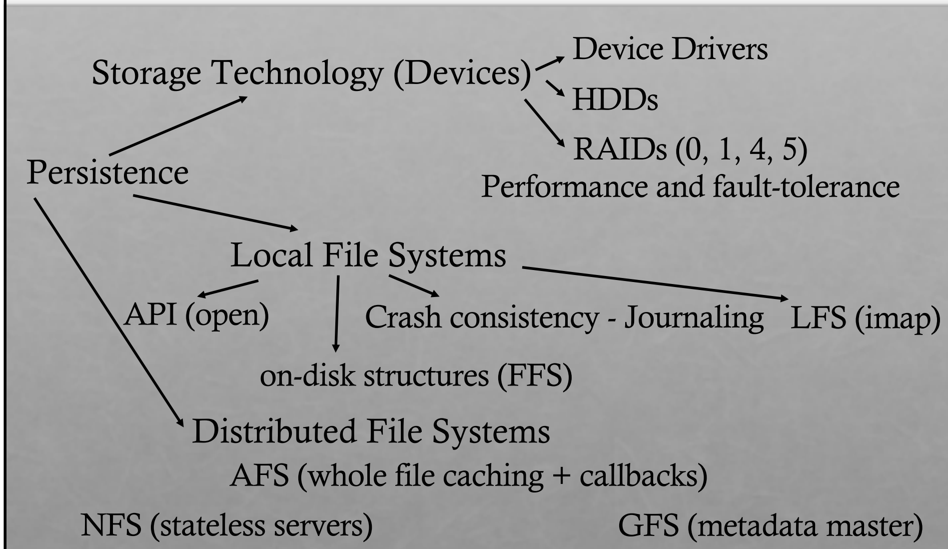
Ordering

Semaphores

Atomic HW Instr

Spin vs Block

REVIEW: EASY PIECE 3



Persistence

Storage Technology (Devices)

Device Drivers

HDDs

RAIDs (0, 1, 4, 5)

Performance and fault-tolerance

Local File Systems

API (open)

on-disk structures (FFS)

Crash consistency - Journaling

LFS (imap)

Distributed File Systems

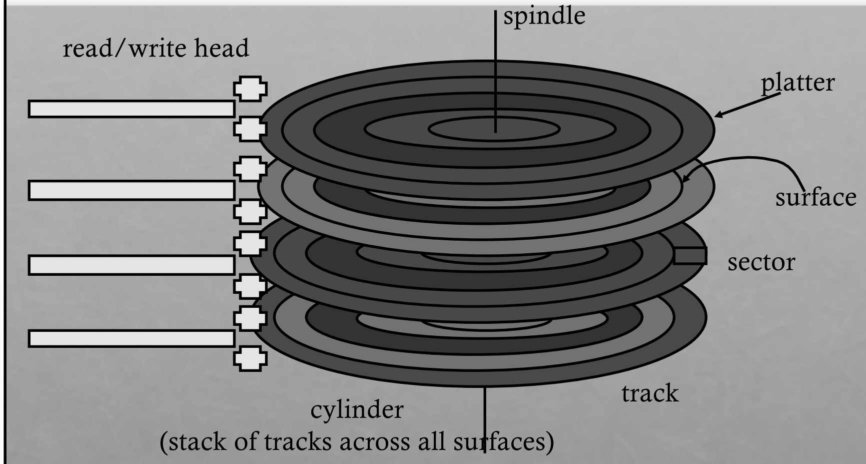
AFS (whole file caching + callbacks)

NFS (stateless servers)

GFS (metadata master)

WHAT QUESTIONS DID YOU ASK?

DISK TERMINOLOGY



RAIDS

RAID METRICS

Capacity: how much space is available to higher levels?

Reliability: how many disks can RAID safely lose?
(assume fail stop!)

Performance: how long does each workload take?

Normalize each to characteristics of one disk

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

RAID-0: STRIPING
4 DISKS

	Disk 0	Disk 1	Disk 2	Disk 4
	0	1	2	3
stripe:	4	5	6	7
	8	9	10	11
	12	13	14	15

Given logical address A, find:
Disk = …
Offset = …

Given logical address A, find:
Disk = A % disk_count
Offset = A / disk_count

RAID-0: ANALYSIS

What is capacity?	N * C
How many disks can fail (no loss)?	0
Latency	D
Throughput (sequential, random)?	**N*S , N*R**

Buying more disks improves throughput, but not latency!

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

Disk 0	Disk 1	Disk 2	Disk 4
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

RAID-1: MIRRORING

	Disk 0	Disk 1
2 disks	0	0
	1	1
	2	2
	3	3

Given logical address A, find:
Disk = …
Offset = …

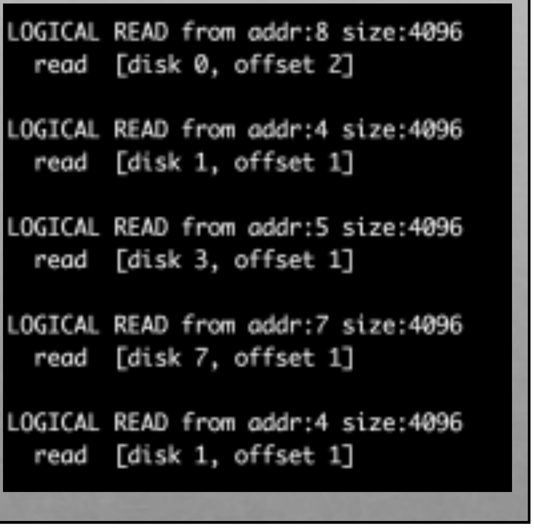
Disk = A % data_disk_count
Offset = A / data_disk_count

To be more precise -- RAID-10:
Stripe of MIRRORS

	Disk 0	Disk 1	Disk 2	Disk 4
4 disks	0	0	1	1
	2	2	3	3
	4	4	5	5
	6	6	7	7

SAMPLE: RAID.PY

```
./raid.py -n 5 -L 1 -R 10 -D 8 -c
```



```
LOGICAL READ from addr:8 size:4096
  read  [disk 0, offset 2]

LOGICAL READ from addr:4 size:4096
  read  [disk 1, offset 1]

LOGICAL READ from addr:5 size:4096
  read  [disk 3, offset 1]

LOGICAL READ from addr:7 size:4096
  read  [disk 7, offset 1]

LOGICAL READ from addr:4 size:4096
  read  [disk 1, offset 1]
```

RAID-1: ANALYSIS

What is capacity?	**N/2 * C**
How many disks can fail?	**1 (or maybe N / 2)**
Latency (read, write)?	**D**

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

Disk 0	Disk 1	Disk 2	Disk 4
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

RAID-1: THROUGHPUT

What is steady-state throughput for

- random reads? **N * R**
- random writes? **N/2 * R**
- sequential writes? **N/2 * S**
- sequential reads? **Book: N/2 * S (other models: N * S)**

Disk 0	Disk 1	Disk 2	Disk 4
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

UPDATING PARITY: XOR

If write "0110" to block 0, how should parity be updated?

One approach: read all other blocks in stripe and calculate new parity

Second approach: Read old value at block 0
- 1100

Read old value for parity
- 0101

Calculate new parity
- 1111
- Write out new parity
- → 2 reads and 2 writes (1 read and 1 write to parity block)

RAID-4 PARITY DISK

	Disk0	Disk1	Disk2	Disk3	Disk4
Stripe:	001	110	101	011	001
					(parity)

RAID-4: ANALYSIS

What is capacity? **(N-1) * C**

How many disks can fail? **1**

Latency (read, write)? **D, 2*D (read and write parity disk)**

Disk0	Disk1	Disk2	Disk3	Disk4
3	0	1	2	6
				(parity)

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

RAID-4: THROUGHPUT

What is steady-state throughput for

- sequential reads? **(N-1) * S**
- sequential writes? **(N-1) * S (parity calculated for full stripe)**
- random reads? **(N-1) * R**
- random writes? **R/2 (read and write parity disk)**

how to avoid parity bottleneck?

Disk0	Disk1	Disk2	Disk3	Disk4
3	0	1	2	6
				(parity)

RAID-5

Disk0	Disk1	Disk2	Disk3	Disk4
-	-	-	-	P
-	-	-	P	-
-	-	P	-	-

…

Rotate parity across different disks
Where exactly do individual data blocks go?

LEFT-SYMMETRIC RAID-5

D0	D1	D2	D3	D4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Pattern repeats…

RAID-5: ANALYSIS

What is capacity?	**(N-1) * C**
How many disks can fail?	**1**
Latency (read, write)?	**D, 2*D (read and write parity disk)**

This metrics same as RAID-4…

Disk0	Disk1	Disk2	Disk3	Disk4
-	-	-	-	P
-	-	-	P	-
-	-	P	-	-

…

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

RAID-5: THROUGHPUT

Steady-state throughput for RAID-4:

- sequential reads?	**(N-1) * S**
- sequential writes?	**(N-1) * S**
- random reads?	**(N-1) * R**
- random writes?	**R/2 (read and write parity disk)**

Disk0	Disk1	Disk2	Disk3	Disk4
3	0	1	2	6
				(parity)

What is steady-state throughput for RAID-5?

- sequential reads?	**(N-1) * S**
- sequential writes?	**(N-1) * S**
- random reads?	**(N) * R**
- random writes?	**N * R/4**

Disk0	Disk1	Disk2	Disk3	Disk4
-	-	-	-	P
-	-	-	P	-
-	-	P	-	-

…

FILE API

RENAME

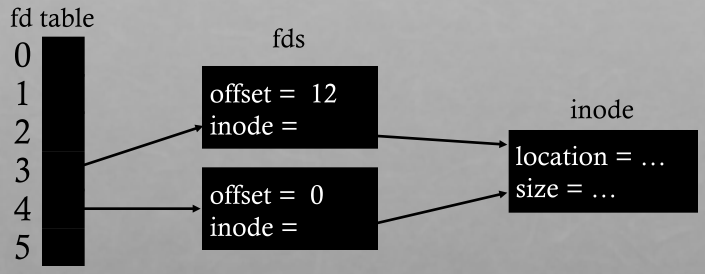
rename (char *old, char *new):

- deletes an old link to a file
- creates a new link to a file

Just changes name of file, does not move data

Even when renaming to new directory (unless…?)

FILE DESCRIPTORS



```
int fd1 = open(“file.txt”); // returns 3
read(fd1, buf, 12);
int fd2 = open(“file.txt”); // returns 4
```

STEPS TO CREATING/WRITING FILES

create /foo/bar

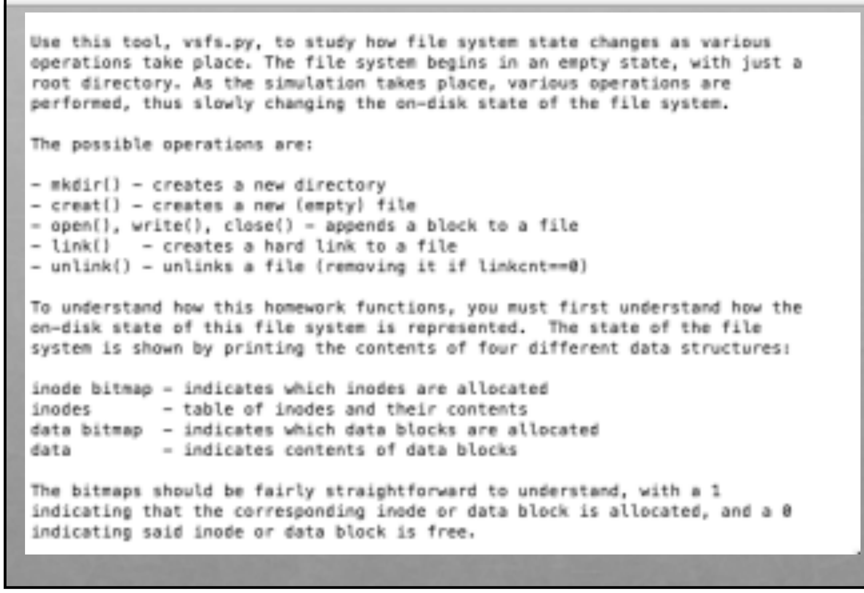
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read
	read write					
				read write		
			write			
						write

Update inode (e.g., size) and data for directory append to /foo/bar

[bar inode in mem]

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							
				write			
							write

SAMPLE HW: VSFS



```
Use this tool, vsfs.py, to study how file system state changes as various
operations take place. The file system begins in an empty state, with just a
root directory. As the simulation takes place, various operations are
performed, thus slowly changing the on-disk state of the file system.

The possible operations are:

- mkdir() - creates a new directory
- creat() - creates a new (empty) file
- open(), write(), close() - appends a block to a file
- link()   - creates a hard link to a file
- unlink() - unlinks a file (removing it if linkcnt==0)

To understand how this homework functions, you must first understand how the
on-disk state of this file system is represented.  The state of the file
system is shown by printing the contents of four different data structures:

inode bitmap - indicates which inodes are allocated
inodes       - table of inodes and their contents
data bitmap  - indicates which data blocks are allocated
data         - indicates contents of data blocks

The bitmaps should be fairly straightforward to understand, with a 1
indicating that the corresponding inode or data block is allocated, and a 0
indicating said inode or data block is free.
```

MOTIVATION FOR JOURNALING

File system is appending to a file and must update 3 blocks:

- inode
- data bitmap
- data block

What happens if crash after only updating some blocks?

a) **bitmap**: lost block

b) **data**: nothing bad

c) **inode**: point to garbage (what?), **another file may use**

d) **bitmap** and **data**: lost block

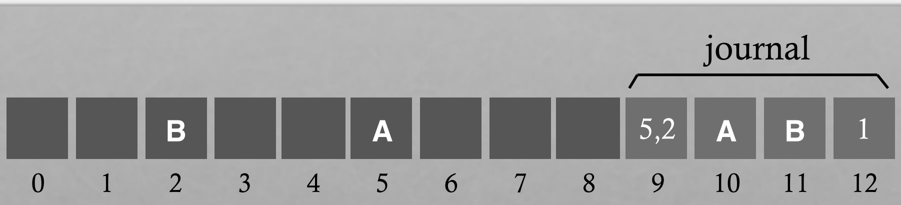
e) **bitmap** and **inode**: point to garbage

f) **data** and **inode**: **another file may use same data block**

BASIC JOURNALING

transaction: write A to block 5; write B to block 2

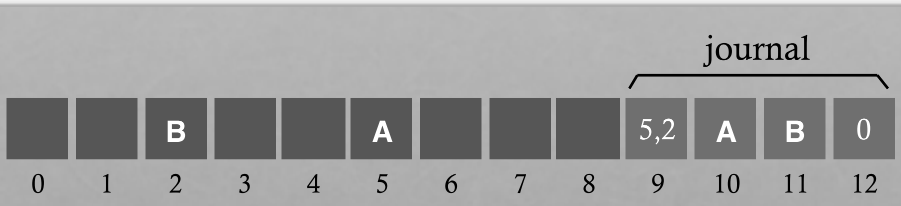
NEW LAYOUT

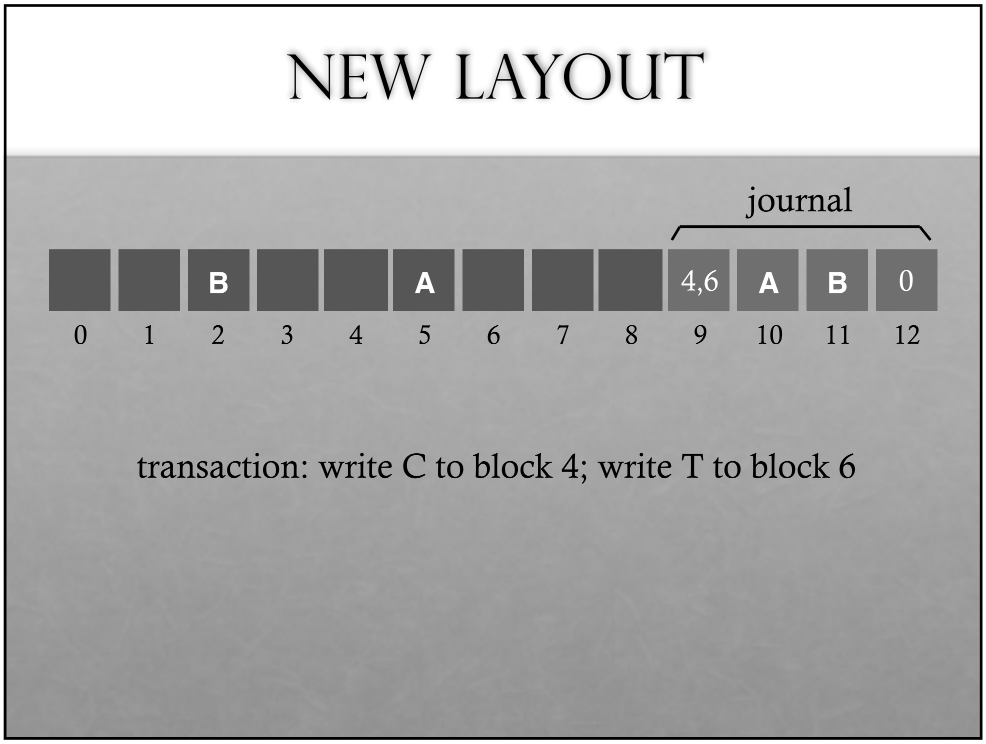


transaction: write A to block 5; write B to block 2

Checkpoint: Writing new data to in-place locations

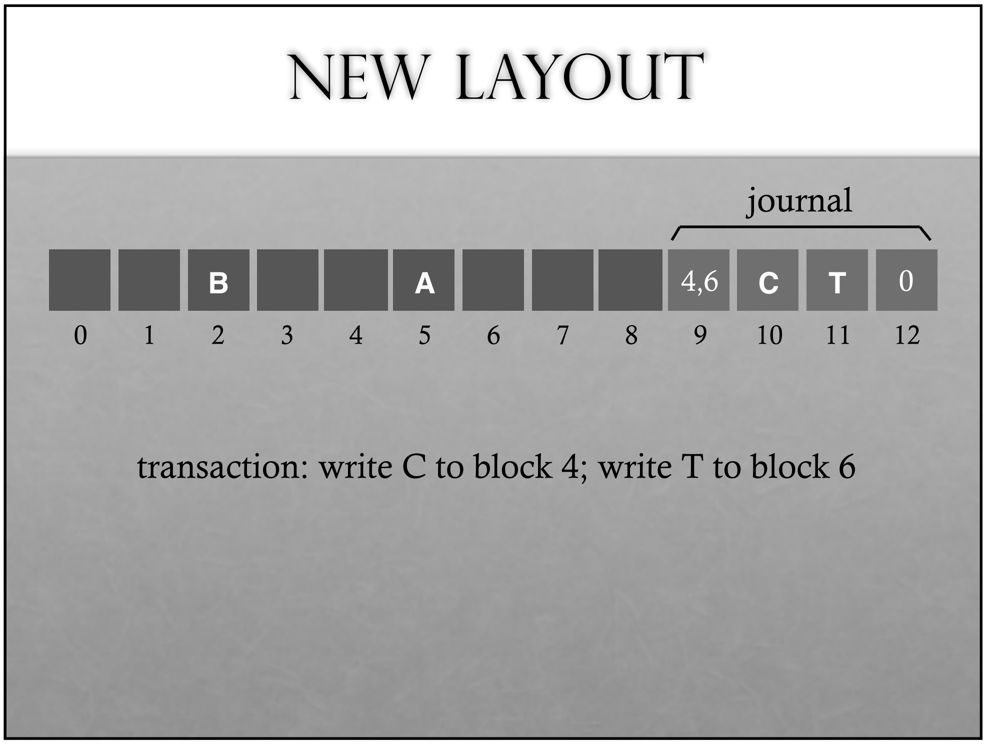
NEW LAYOUT





NEW LAYOUT

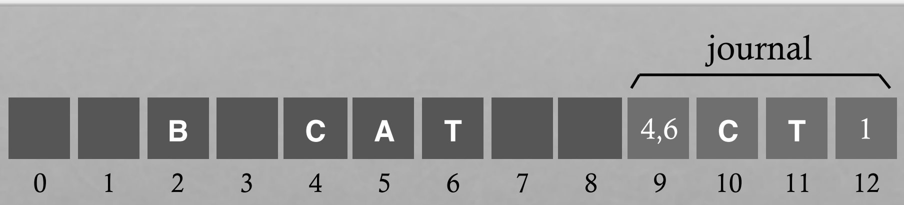
transaction: write C to block 4; write T to block 6



NEW LAYOUT

transaction: write C to block 4; write T to block 6

NEW LAYOUT

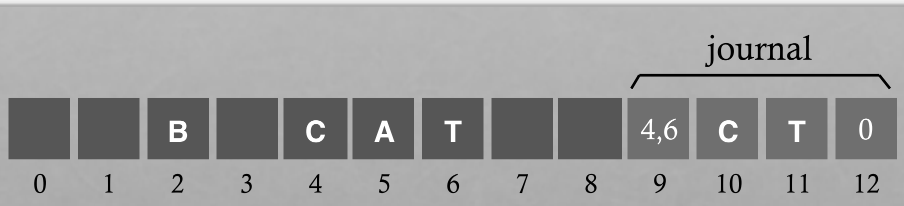


		B		C	A	T			4,6	C	T	1
0	1	2	3	4	5	6	7	8	9	10	11	12

transaction: write C to block 4; write T to block 6

Checkpoint: Writing new data to in-place locations

NEW LAYOUT



		B		C	A	T			4,6	C	T	0
0	1	2	3	4	5	6	7	8	9	10	11	12

transaction: write C to block 4; write T to block 6

JOURNALING: STATES AFTER CRASH

a) No transactions replayed during recovery; file system in old state

b) No transactions replayed during recovery; file system in new state

c) Transaction replayed during recovery; file system in old state

d) Transaction replayed during recovery; file system in new state

e) Transaction replayed during recovery; file system in unknown state

Which of these 2 are not possible?

LFS

When LFS writes a new copy of a **data block** to a segment, it also writes a new copy of the **inode** that points to that data block.

- *True; since the data is in a new location in the log, the pointers to that location stored in the inode also have to change; LFS does not overwrite inodes (which would be a random write) and instead writes a new copy of the inode to the log.*

When LFS writes a new copy of **inode** to a segment, it also writes a new copy of the **directory** that points to that inode.

- *False; LFS handles the fact that the location of the inode changes by having an imap to track the current location of each inode.*

LFS periodically **checkpoints imaps** to a known location on disk (alternating between two locations to withstand crashes).

- *False; LFS checkpoints pointers to portions of the imap in known locations; the modified imaps themselves are written out to each segment.*

When performing **garbage collection**, LFS determines that an **inode** is valid by verifying that the corresponding entry in the imap points to this location.

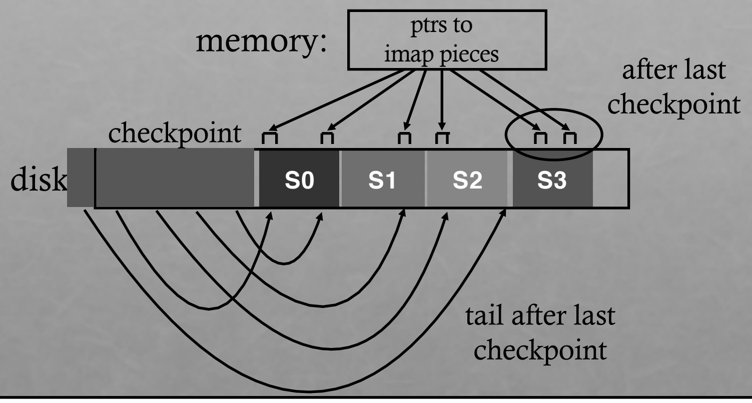
- *True, this is what it does.*

When performing **garbage collection**, LFS determines that a **data block** is valid by scanning all valid inodes from the imap and verifying that one of the valid inodes points to this location.

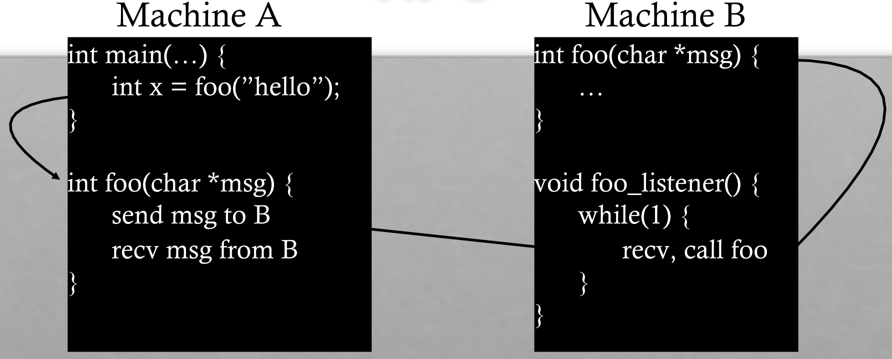
- *False; this would be way too slow. Instead, LFS writes segment summary info to each segment that describes each updated data block (i.e., the inode that points to it and its offset in the file).*

LFS IMAP

In log-structured based file system there is a checkpoint region (CR) which contains pointers to the inode map which contains pointer to the inode which points to the data. Won't this affect performance to read 4 times from disk to get the data from 1 inode?



RPC



```
int main(…) {
    int x = foo(”hello”);
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

```
int foo(char *msg) {
    …
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

Actual calls

RAW MESSAGES: UDP

UDP : User Datagram Protocol

API

- reads and writes over socket file descriptors
- messages sent from/to ports to target a process on machine

Provide minimal reliability features:

- messages may be lost
- messages may be reordered
- messages may be duplicated
- only protection: checksums to ensure data not corrupted

RELIABLE MESSAGES: LAYERING STRATEGY

TCP: Transmission Control Protocol

Using software, build reliable, logical connections over unreliable connections

- **Make sure each message is received**
- Make sure messages are received in order
- Make sure no duplicates are received

Techniques:

- Acknowledgment (ACK)
- Time-outs with retransmit
- Sequence numbers

NFS SUMMARY

NFS handles client and server crashes very well;
robust APIs

- **stateless**: servers don't remember clients or open files

- **idempotent**: repeating operations gives same results

Details:

- Writes: Clients flush writes on file close
- Reads: Check if can re-use data blocks for file every 3 seconds (getattr to server)

Problems:

- Consistency model is odd (client may not see updates until 3 seconds after file is closed)
- Scalability limitations as more clients call stat() (getattr) on server

AFS SUMMARY

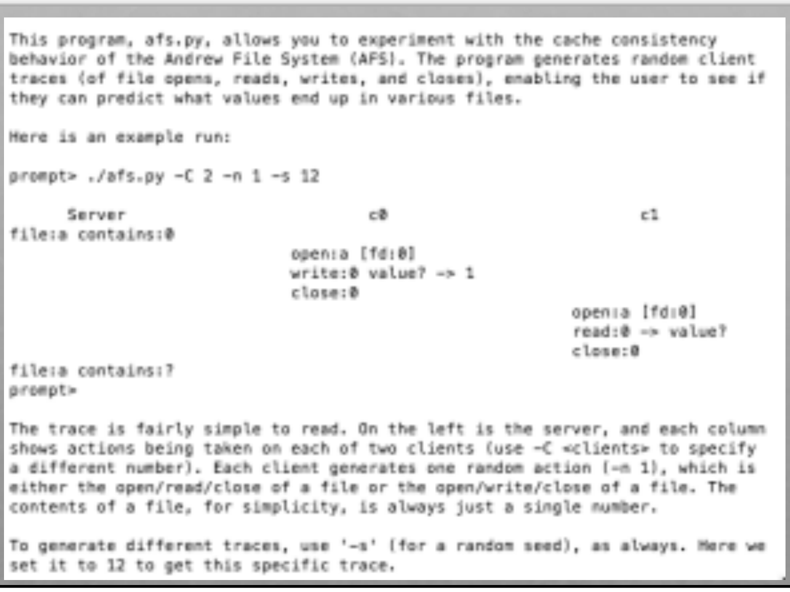
Whole-file caching

- Upon open, AFS client fetches whole file (unless have fetched before…), storing in local memory or disk
- More intuitive semantics (see version of file that existed when file was opened)
- Upon close, client flushes file to server (if file was written)

State is useful for **scalability**, but makes handling crashes hard

- Server tracks callbacks for clients that have file cached
- Lose callbacks when server crashes…
- If file is changed, notify all clients that have that cached so won't re-use NEXT TIME client calls open()

HOMEWORK: AFS



This program, afs.py, allows you to experiment with the cache consistency behavior of the Andrew File System (AFS). The program generates random client traces (of file opens, reads, writes, and closes), enabling the user to see if they can predict what values end up in various files.

Here is an example run:

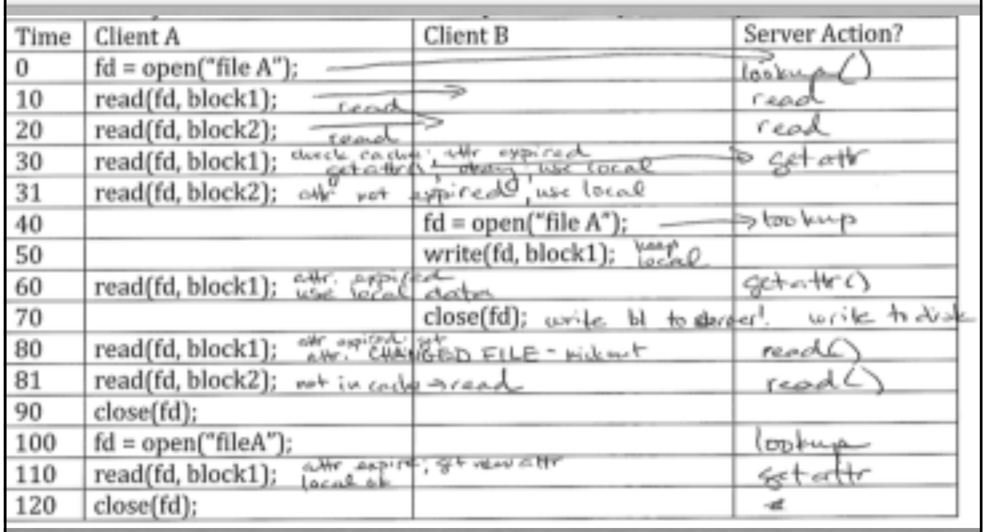
```
prompt> ./afs.py -C 2 -n 1 -s 12

      Server                         c0                          c1
file:a contains:0
                             open:a [fd:0]
                             write:0 value? -> 1
                             close:0
                                                         open:a [fd:0]
                                                         read:0 -> value?
                                                         close:0
file:a contains:?
prompt>
```

The trace is fairly simple to read. On the left is the server, and each column shows actions being taken on each of two clients (use -C <clients> to specify a different number). Each client generates one random action (-n 1), which is either the open/read/close of a file or the open/write/close of a file. The contents of a file, for simplicity, is always just a single number.

To generate different traces, use '-s' (for a random seed), as always. Here we set it to 12 to get this specific trace.

NFS PROTOCOL



Time	Client A	Client B	Server Action?
0	fd = open("file A");		lookup()
10	read(fd, block1); read		read
20	read(fd, block2); read		read
30	read(fd, block1); check cache; attr expired getattr; ~~okay~~ use local		get attr
31	read(fd, block2); attr not expired, use local		
40		fd = open("file A");	lookup
50		write(fd, block1); keep local	
60	read(fd, block1); attr: expired use local data		getattr()
70		close(fd); write bl to server!	write to disk
80	read(fd, block1); attr expired: get attr: CHANGED FILE - kick out		read()
81	read(fd, block2); not in cache → read		read()
90	close(fd);		
100	fd = open("fileA");		lookup
110	read(fd, block1); attr expire, get new attr local ok		getattr
120	close(fd);		

AFS PROTOCOL

Time	Client A	Client B	Server Action?
0	fd = open("file A");		setup callback for A
10	read(fd, block1);	send all of	file A
20	read(fd, block2); local!!		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	setup call back
50		write(fd, block1); send	all of A
60	read(fd, block1); local		
70		close(fd); send back	changes of A
80	read(fd, block1); local		break callbacks
81	read(fd, block2); local		
90	close(fd); nothing changed		
100	fd = open("fileA"); no callback!! need to	fetch A again	
110	read(fd, block1);		
120	close(fd);	send A	

GOOD LUCK!