REVIEW, SUMMARY

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Project 6 grades

- Grade ranges

Midterm 3

HelioCampus feedback

Shivaram OH: 1-2pm Thu

RECAP

CHALLENGES IN DISTRIBUTED SYSTEMS

System failure: need to worry about partial failure

Communication failure: links unreliable

- bit errors
- packet loss
- node/link failure

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC

RAW MESSAGES: UDP

Advantages

- Lightweight
- Some applications make better reliability decisions themselves (e.g., video conferencing programs)

Disadvantages

- More difficult to write applications correctly
- messages may be lost
- messages may be reordered

RELIABLE MESSAGES: LAYERING STRATEGY

TCP: Transmission Control Protocol

Key features

- Ack: sender knows message was received
- Timeout: how long to wait to resend
- Sequence numbers: avoid duplicate / out-of-order messages

RPC

Machine A

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

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

client wrapper

Machine B

```
int foo(char *msg) {
        …
}

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

server wrapper

WRAPPER GENERATION

Wrappers must do conversions:

- client arguments to message
- message to server arguments
- convert server return value to message
- convert message to client return value

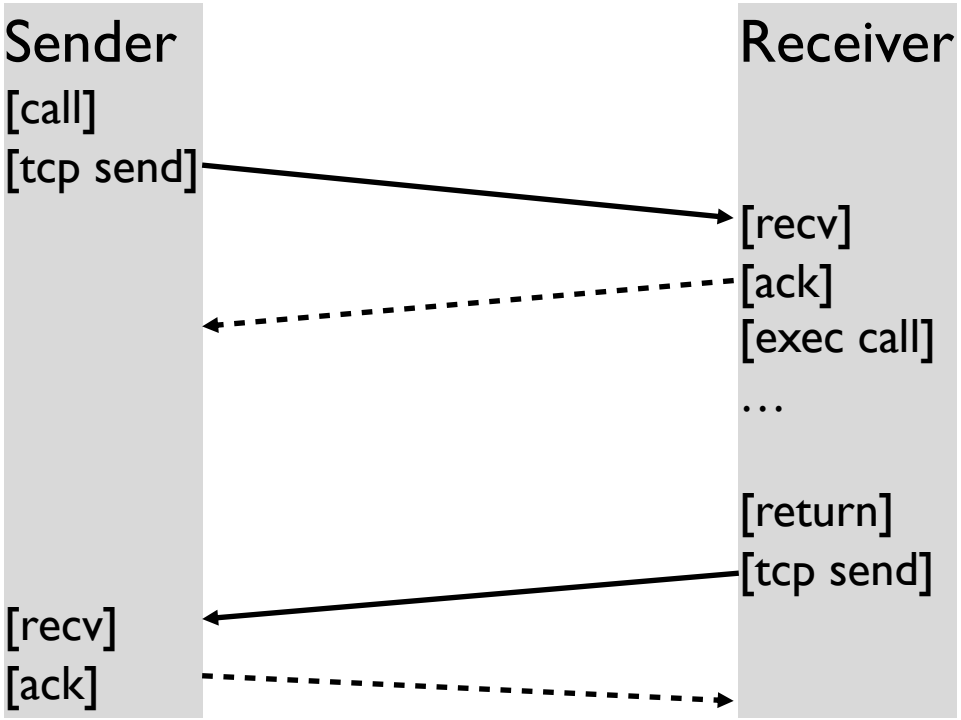
Need uniform endianness (wrappers do this)

Conversion is called marshaling/unmarshaling, or serializing/deserializing

Why are pointers problematic?

Address passed from client not valid on server

RPC OVER TCP?


Sender
[call]
[tcp send]
[recv]
[ack]
Receiver
[recv]
[ack]
[exec call]
…
[return]
[tcp send]

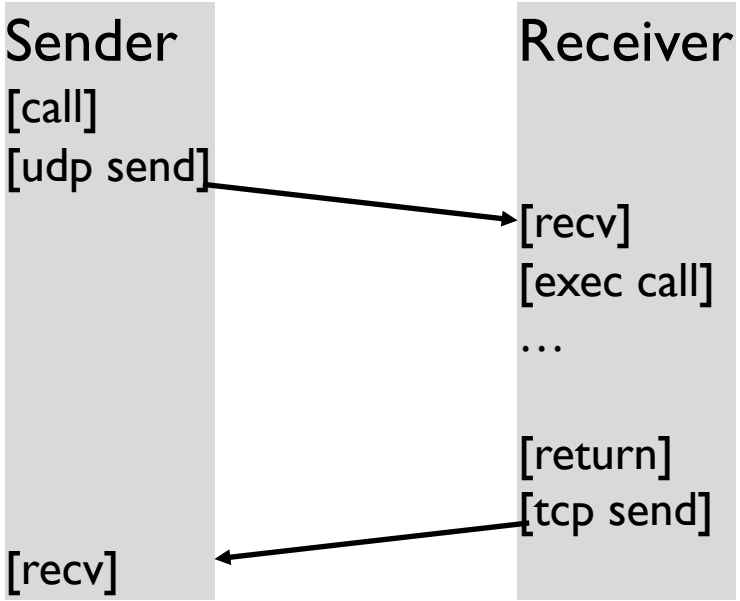
RPC OVER UDP

Strategy: use function return as implicit ACK

Piggybacking technique

What if function takes a long time?

then send a separate ACK



MIDTERM 3: PERSISTENCE, ADVANCED TOPICS

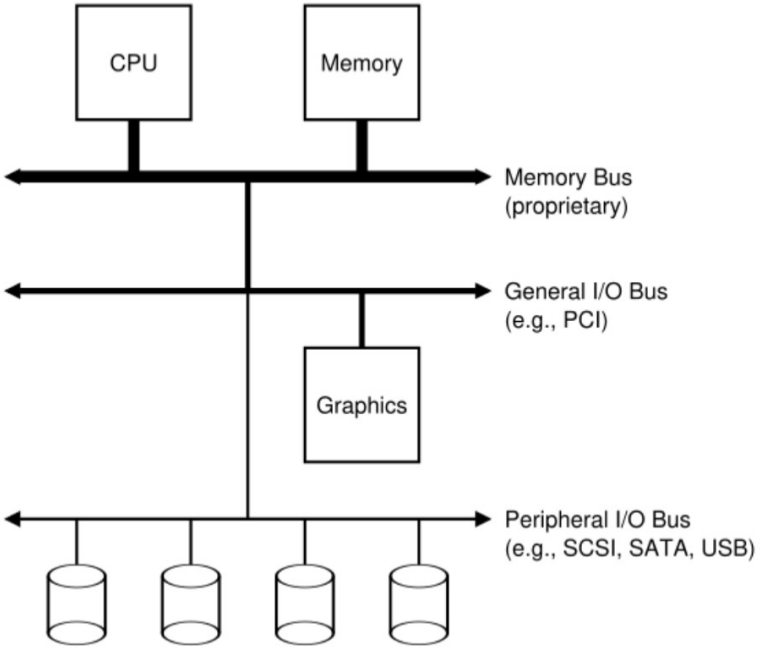
IO Devices
- Disks
- SSDs

File API

Filesystems
- Simple FS, FFS
- Journaling
- Log Structured FS

Virtual Machines
Multiprocessor Scheduling
Distributed Systems

REVIEW: IO DEVICES



Polling vs. Interrupt

Using DMA for memory copy

Instructions for I/O or memory-mapped

SEEK, ROTATE, TRANSFER

Seek cost: Function of cylinder distance

- Not purely linear cost
- Must accelerate, coast, decelerate, settle
- Settling alone can take 0.5 - 2 ms

Entire seeks often takes 4 - 10 ms

Average seek = 1/3 of max seek

Depends on rotations per minute (RPM)

- 7200 RPM is common, 15000 RPM is high end

Average rotation?

Pretty fast: depends on RPM and sector density.

100+ MB/s is typical for maximum transfer rate

I/O SCHEDULERS

Given a stream of I/O requests, in what order should they be served?

- First come first served
- SSTF (Shortest SEEK Time First): choose request that requires least seek time
- SCAN or Elevator Algorithm: Sweep back and forth, from one end of disk other,

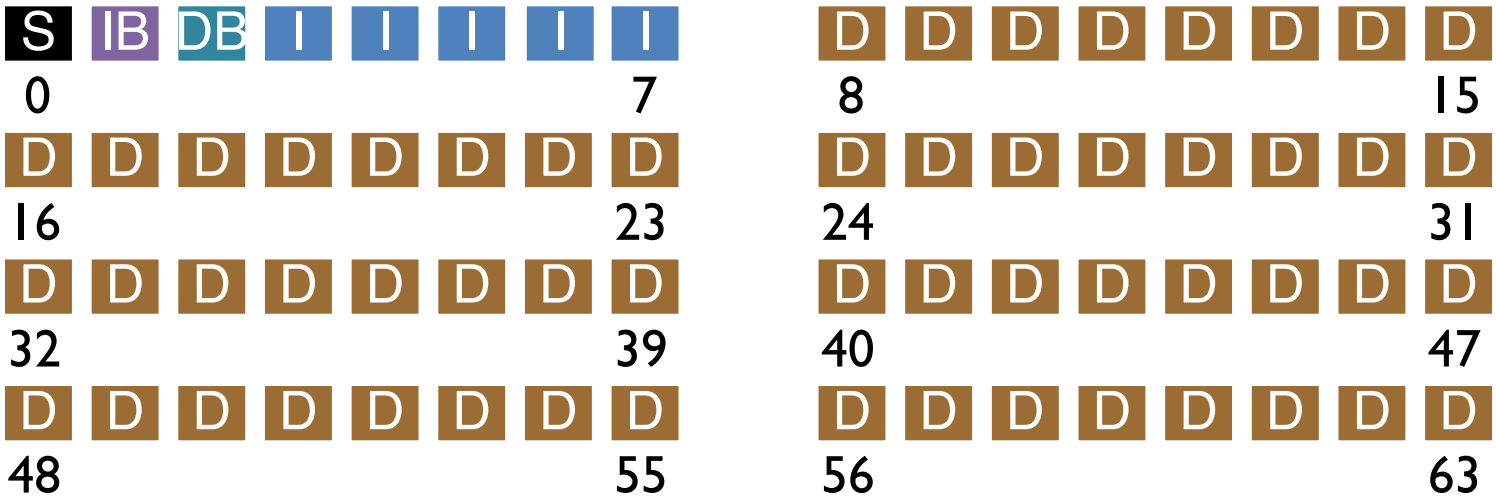
FILE API

```
int fd = open(char *path, int flag, mode_t mode)
read(int fd, void *buf, size_t nbyte)
write(int fd, void *buf, size_t nbyte)
close(int fd)
```

advantages:

- string names
- hierarchical
- traverse once
- offsets precisely defined

FS STRUCTS: SUPERBLOCK



INODE

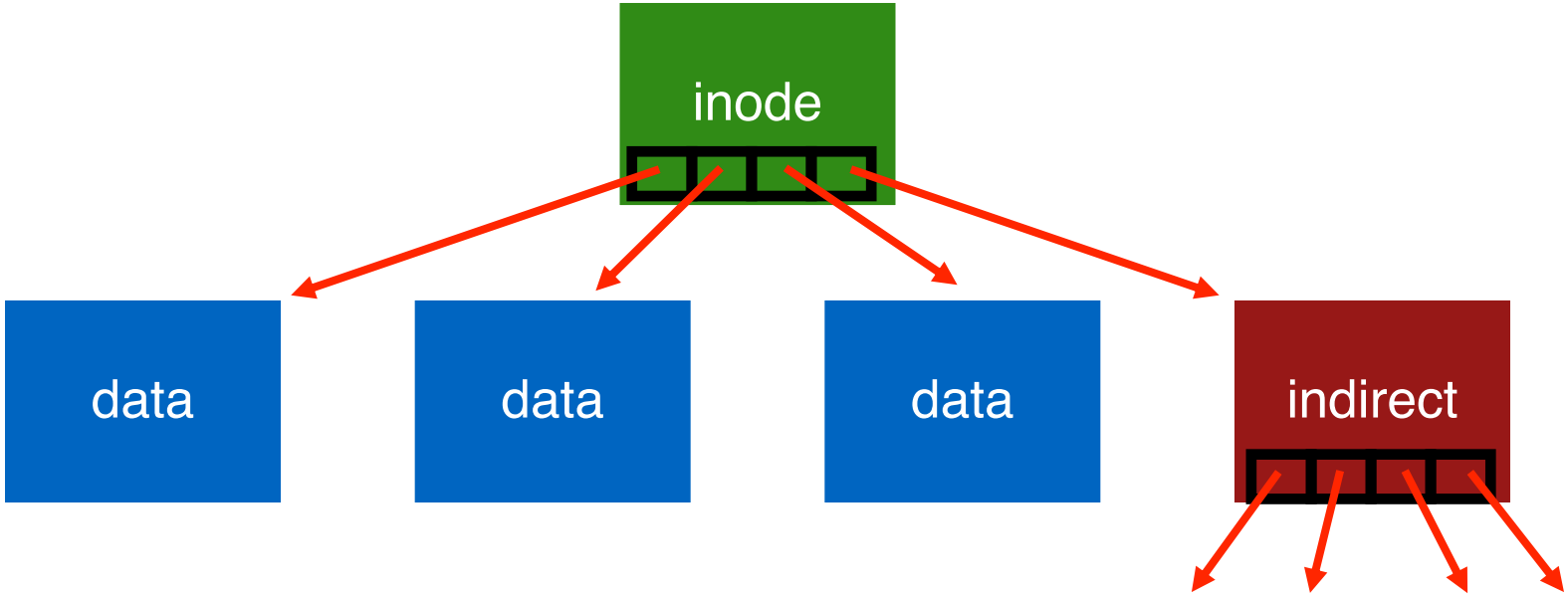
type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

Assume single level (just pointers to data blocks)

What is max file size?

Assume 256-byte inodes
(all can be used for pointers)
Assume 4-byte addrs

How to get larger files?



Better for small files!

How to handle even larger files?

An inode has three fields: type (f for file and d for directory), address of data block (either -1 if no data or a single integer address for the data block), and a reference count saying how many directory entries there are to this inode.

Data blocks are indicated by matching square brackets and can contain file data (a single character) or directory information (name-inode pairs). Directory data always fits into a single data block.

```
inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []
```

Initial state

Initial state

```
inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []
```

```
inode bitmap  11000000
inodes        [d a:0 r:3][d a:1 r:2][][][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (k,1)][(.,1) (..,0)][][][][][][]
```

```
inode bitmap  11100000
inodes        [d a:0 r:3][d a:1 r:2][f a:-1 r:1][][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (k,1)][(.,1) (..,0) (h,2)][][][][][][]
```

```
inode bitmap 11100000
inodes       [d a:0 r:3][d a:1 r:2][f a:-1 r:2][][][][][]
data bitmap  11000000
data         [(.,0) (..,0) (k,1) (h,2)][(.,1) (..,0) (h,2)][][][][][][]
```

Course feedback:
https://heliocampusac.wisc.edu

FFS PLACEMENT GROUPS

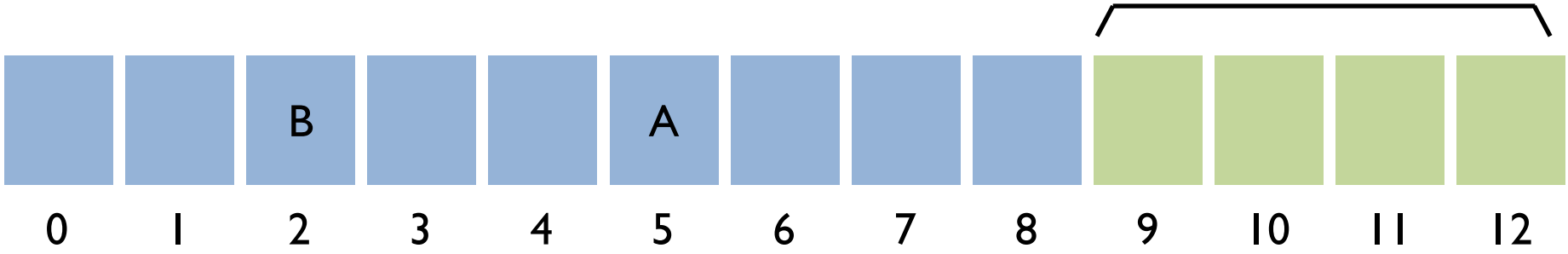
Key idea: Keep inode close to data

Use groups across disks;

Strategy: allocate inodes and data blocks in same group.

ORDERING FOR CONSISTENCY

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



write order

9,10,11

12

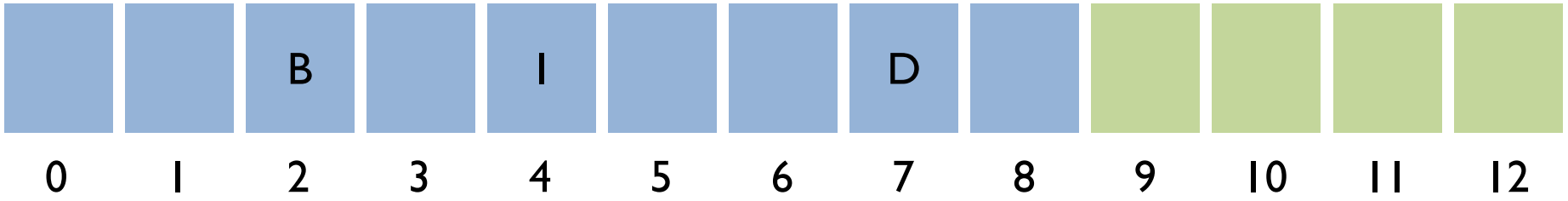
4,6

Barriers

1) Before journal commit, ensure journal entries complete
2) Before checkpoint, ensure journal commit complete
3) Before free journal, ensure in-place updates complete

ORDERED JOURNALING

Still only journal metadata. But write data **before** the transaction!



What happens if crash in between?

write order

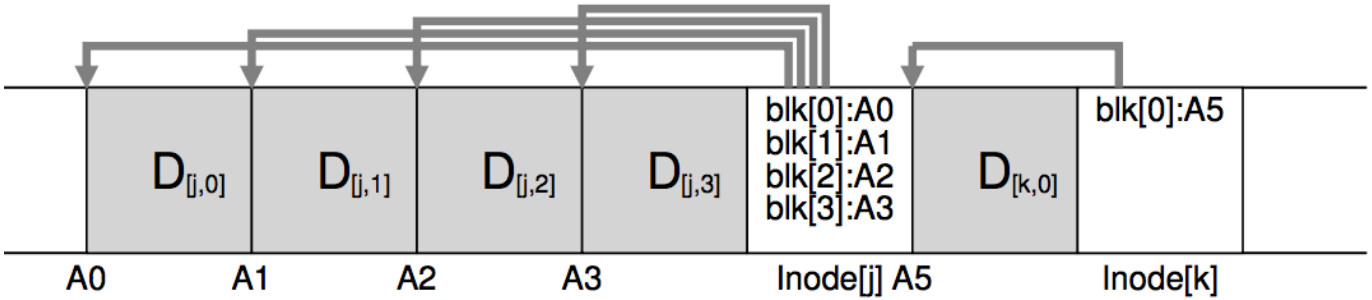
7

9, 10, 11

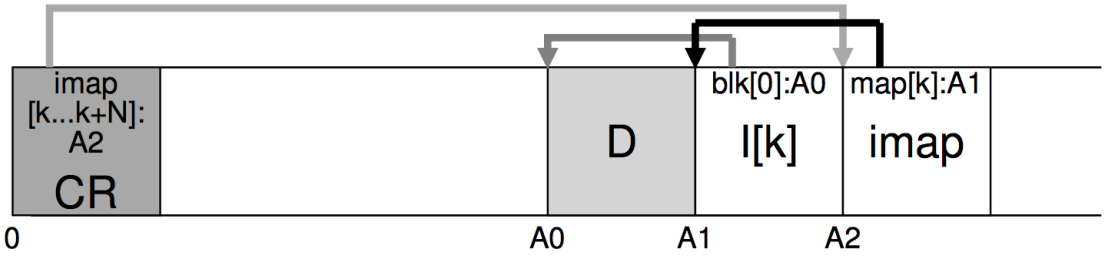
12

2, 4

BUFFERED WRITES



READING IN LFS



1. Read the Checkpoint region
2. Read all imap parts, cache in mem
3. To read a file:
    1. Lookup inode location in imap
    2. Read inode
    3. Read the file block

GARBAGE COLLECTION

Need to reclaim space:

1. When no more references (any file system)
2. After newer copy is created (COW file system)

LFS reclaims **segments** (not individual inodes and data blocks)

- Want future overwites to be to sequential areas
- Tricky, since segments are usually partly valid

SSD OPERATIONS

Read a page: Retrieve contents of entire page (e.g., 4 KB)

- Cost: 25—75 microseconds
- Independent of page number, prior request offsets

Erase a block: Resets each page in the block to all 1s

- Cost: 1.5 to 4.5 milliseconds
- Much more expensive than reading!
- Allows each page to be written

Program (i.e., write) a page: Change selected 1s to 0s

- Cost is 200 to1400 microseconds
- Faster than erasing a block, but slower than reading a page

FLASH TRANSLATION LAYER

1. Translate reads/writes to logical blocks into reads/erases/programs

2. Reduce write amplification (extra copying needed to deal with block-level erases)

3. Implement wear leveling (distribute writes equally to all blocks)

Typically implemented in hardware in the SSD, but in software for some SSDs

FIND THE ERRORS

```
inode bitmap 11100000
inodes       [d a:0 r:3][d a:1 r:2][f a:2 r:2][][][][][]
data bitmap  11100000
data         [(.,0) (..,0) (k,1) (h,2)][(.,0) (..,1) (h,2)][u][][][][][]
```

```
inode bitmap 11000000
inodes       [d a:0 r:3][d a:1 r:2][f a:2 r:2][][][][][]
data bitmap  11100000
data         [(.,0) (..,0) (k,1) (h,2)][(.,1) (..,0) (h,2)][u][][][][][]
```

LOOKING BACK, LOOKING FORWARD

1. What was one idea or concept that you learnt in this course that you appreciated the most?

2. What are some future opportunities that you look forward to based on content from 537?

NEXT COURSES

CS 640: Computer Networks

CS 736: Advanced Operating Systems

CS 739: Advanced Distributed Systems

CS 744: Big Data Systems

THANK YOU!