CS162 Midterm 3 Review

Fall 2024

Disclaimer

This is not an exhaustive review of all topics that are in scope for the midterm.

“You are responsible for the sum total of human knowledge since the beginning of recorded history with particular emphasis on the contents of this course.”

~a certain wise guy at Berkeley

Outline

Scope of midterm: all lectures, homeworks, and projects.

This review session will focus on post midterm 2 material (which will be emphasized on midterm 3).

- I/O, Storage Devices, File Systems
- Durability, Reliability, Transactions
- Distributed Decision Making, Networking

If we have time at the end, we'll review a bit of queuing theory.

I/O

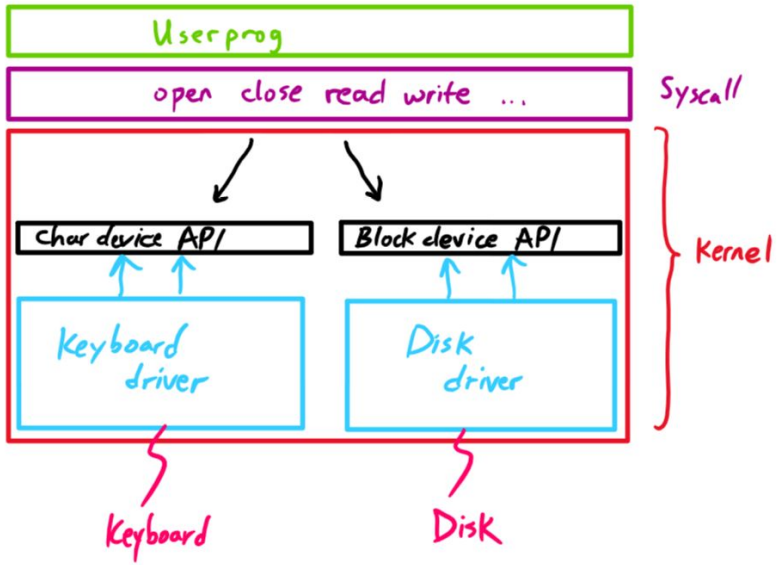
I/O

- In Linux, user programs communicate with devices via a standard file-like syscall interface (e.g. open, close, read, write).
- For instance, if we wanted to write something to a device, we would do

```
FILE fd = fopen("/dev/something", "rw");
for (int i = 0; i < 10; i++) {
  fprintf(fd, "Count %d\n", i);
}
close(fd);
```

I/O

- Kernel: translate these high-level syscalls into device instructions. Depending on the device, it will do so differently.



Device Drivers

How are device drivers implemented? Need a way for CPU to talk to device registers.

- CPU is connected to every device via data buses (like the PCI).
- Two main ways to communicate: **programmed I/O** vs. **DMA**.

Device Drivers

Programmed I/O:

- **Port-mapped I/O**: privileged in/out instructions to communicate with device registers.
- **Memory-mapped I/O**: registers appear in physical address space. Communicate with load/store instructions.
- CPU involved in every data transfer.
- Often requires polling! (so rather wasteful of CPU cycles)

Device Drivers

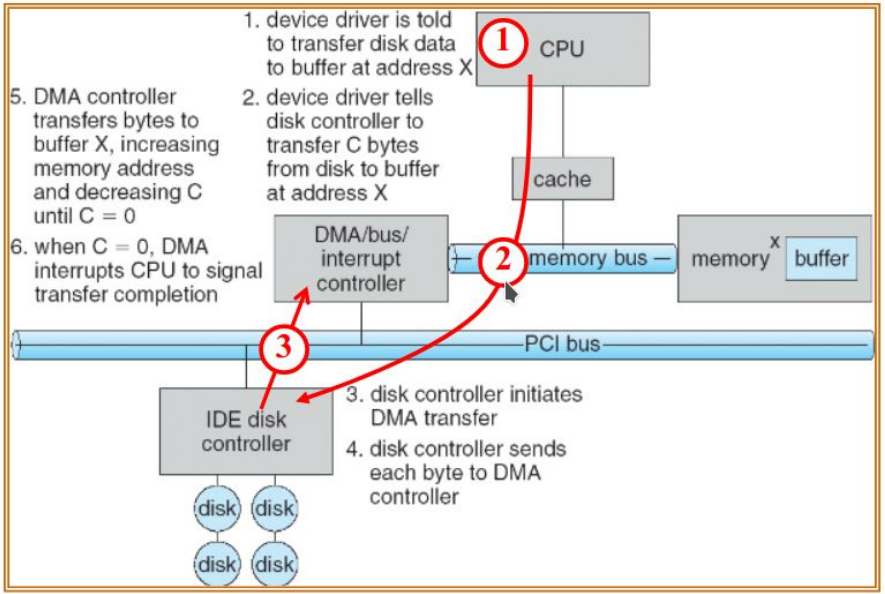
Example of port-mapped I/O: the speaker driver in Pintos.

```
/* Sets the PC speaker to emit a tone at the given FREQUENCY, in
   Hz. */
void
speaker_on (int frequency)
{
  if (frequency >= 20 && frequency <= 20000)
    {
      /* Set the timer channel that's connected to the speaker to
         output a square wave at the given FREQUENCY, then
         connect the timer channel output to the speaker. */
      enum intr_level old_level = intr_disable ();
      pit_configure_channel (2, 3, frequency);
      outb (SPEAKER_PORT_GATE, inb (SPEAKER_PORT_GATE) | SPEAKER_GATE_ENABLE);
      intr_set_level (old_level);
    }
  else
    {
      /* FREQUENCY is outside the range of normal human hearing.
         Just turn off the speaker. */
      speaker_off ();
    }
}
```

Device Drivers

Direct Memory Access:

1. CPU gives DMA request to device controller.
2. Device controller communicates with DMA controller and they work together to transfer data between device and memory.
3. When finished, DMA controller interrupts CPU.



Device Drivers

Note if a device driver uses DMA, then its code can be split into two parts:

1. Code that implements the device driver interface (e.g. for a disk driver, the block device interface) and starts the DMA request.
2. Code that executes once the DMA request finishes and the CPU is interrupted.

The former is called the **top half**, the latter is called the **bottom half**.

- If we are not using DMA or any interrupt-driven I/O, there would no bottom half.

I/O

Next up: storage devices.

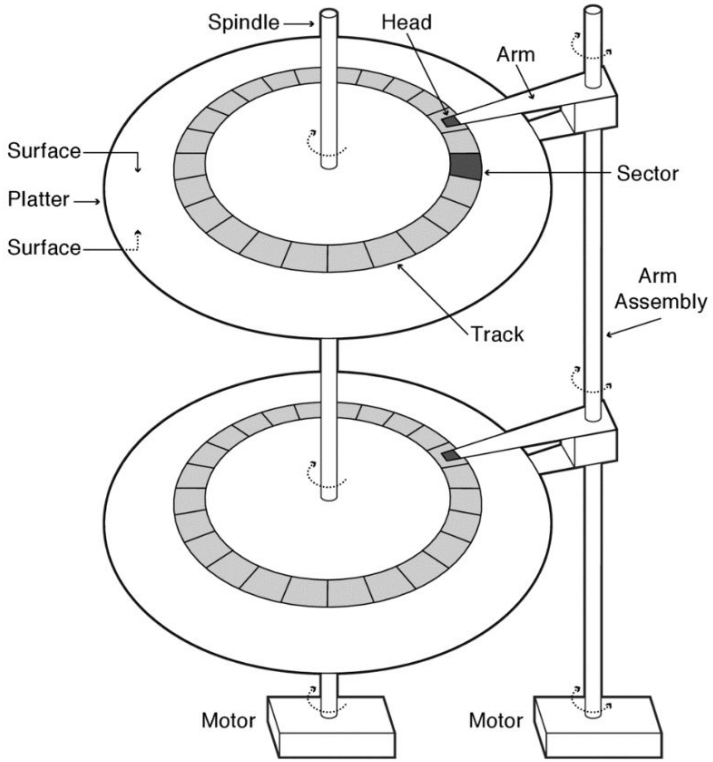
- The most important of devices.
- Performance is key! (Keep in mind)

Ways to measure performance:

- Latency = response time.
- Throughput = rate of which tasks are performed (e.g. operations per sec).

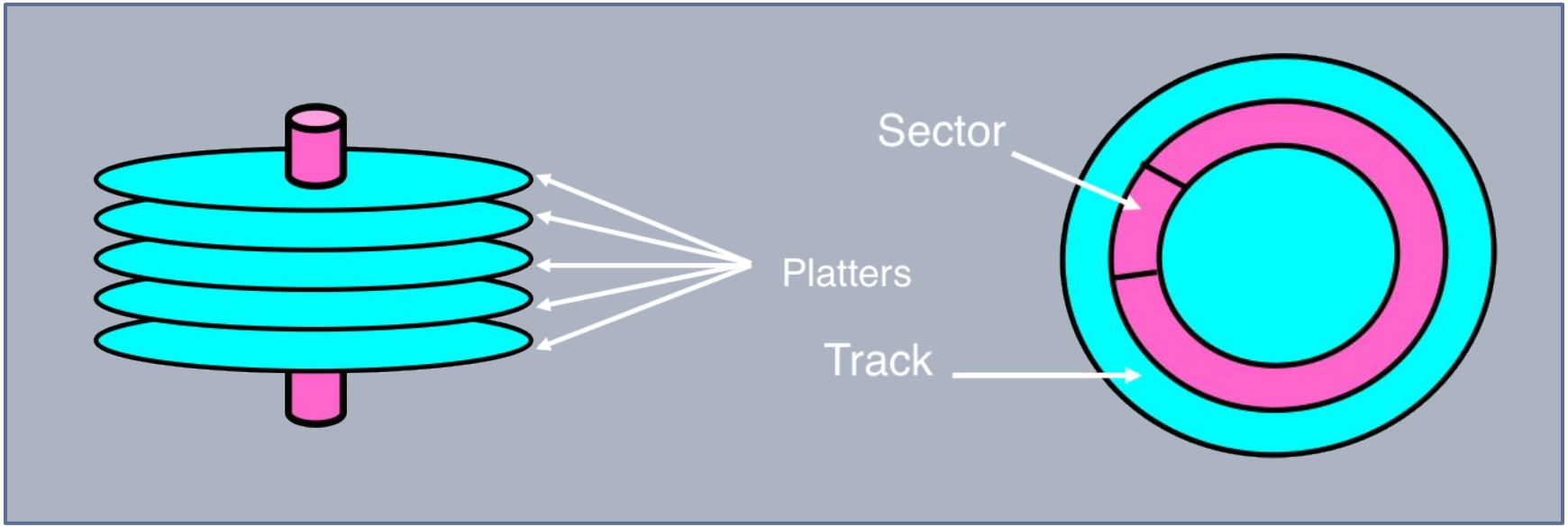
Storage Devices

HDDs


Spindle
Head
Arm
Surface
Platter
Surface
Sector
Arm
Assembly
Track
Motor
Motor

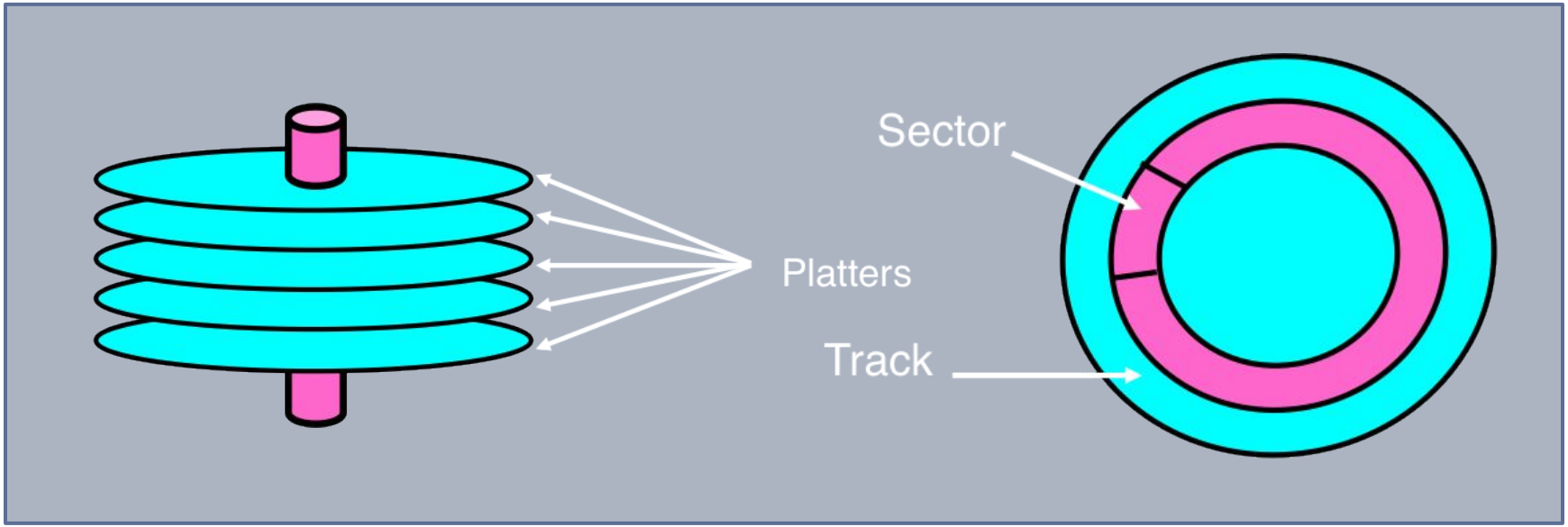
HDDs

- Read and write in sectors



HDDs

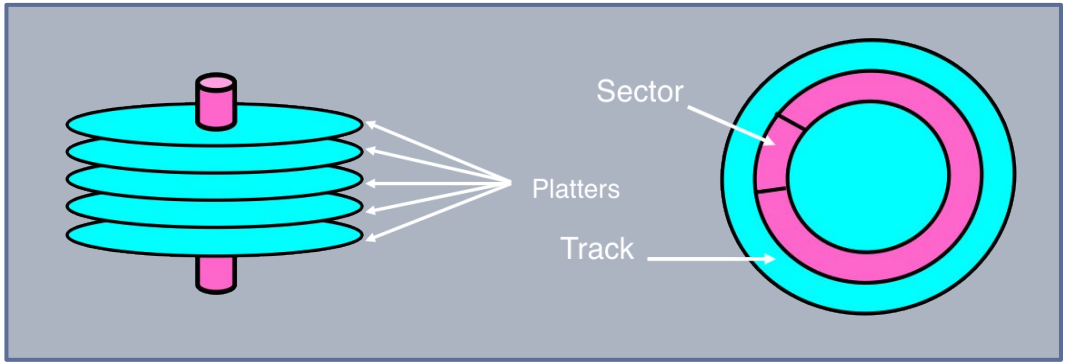
Latency = Queuing + Controller + Seek + Rotational + Transfer



HDDs

Latency = Queuing + Controller + Seek + Rotational + Transfer

- Seek: find desired track
- Rotational: wait for desired sector to rotate to you (~half a rotation)
- Transfer: read/write data at sector



HDDs

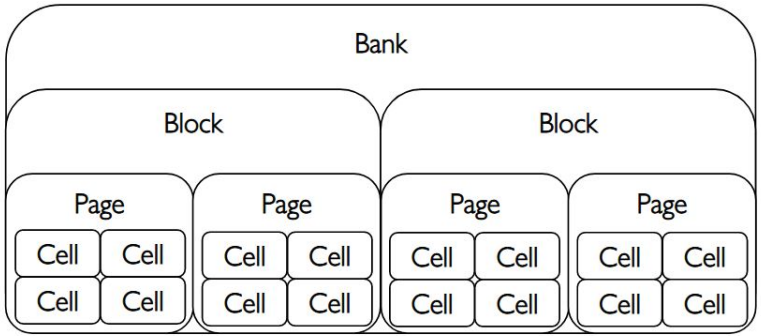
- Slow for random reads and writes, faster for sequential
- Cheap

SSDs

SSDs

- Have no moving parts. So no seek and rotational latency!
- **Latency = Queuing + Controller + Transfer**

- Read and write in pages (different from “pages” in virtual memory)
- Erase in erasure blocks (page << erasure block)



SSDs

- Fast for random reads and writes
- Erasing slow (and must erase before overwriting)
- More expensive than HDDs, erasing wears SSDs out

HDD Questions

We have a hard drive with the following specs:

- 4096 byte sectors
- 3,000,000 sectors/track
- 100 tracks/platter
- 2 platters*
- 5400 rpm
- 5.6ms seek time (avg)
- 1ms controller+queue time
- 140MB/s Transfer rate

1. How big is the drive?
2. What is the throughput for a 64KB read?
3. What is the throughput for a 64MB read?

Reminders:

- time = queue + controller + seek + rotation + transfer
- throughput = amount of memory read / time

*Assume (for simplicity) that data is only stored on ONE SIDE of a platter and all reads are sequential.

HDD Questions

1. How big is the drive?

Each track contains 3,000,000 * 4096 B = 12.288 GB
For 100 tracks: 100 * 12.288 GB = 1228.8 GB
2 platters: 2 * 1228.8 GB = 2457.6 GB ~= 2.46 TB

HDD Questions

2. What is the throughput for a 64KB read?

Data: 64KB
Time: queue + controller + seek + rotation + transfer

queue + controller: 1 ms
seek: 5.6 ms
rotation: 1/5400 min = 11.11 ms (worst case) => 5.55 ms (average)
transfer time: 64KB / 140MB/s = 0.457 ms
total time: 1 + 5.6 + 5.55 + 0.457 = 12.60 ms
throughput = 64KB / 0.01260s = 5079 KB/s

HDD Questions

3. What is the throughput for a 64MB read?

Data: 64MB
Time: queue + controller + seek + rotation + transfer

queue + controller: 1 ms
seek: 5.6 ms
rotation: 1/5400 min = 11.11 ms (worst case) => 5.55 ms (average)
transfer time: 64MB / 140MB/s = 457 ms
total time: 1 + 5.6 + 5.55 + 457 = 475.9 ms
throughput = 64MB / 0.4759 s = 134454 KB/s

File Systems

Storage Device Abstraction

To the operating system, a storage device (whether HDD or SSD) is just a indexed array of **blocks**

- For HDDs, a block = a fixed number of sectors
- For SSDs, a block = usually the size of an erasure block

In UNIX systems, a block is often 4kb.

Storage Device Abstraction

To the operating system, a storage device (whether HDD or SSD) is just a indexed array of **blocks**

OS usually can only do two things:

- Read to block at any index i
- Write to block at any index i

It is the goal of the OS to transform this low level abstraction into a file system.

How do we store files?

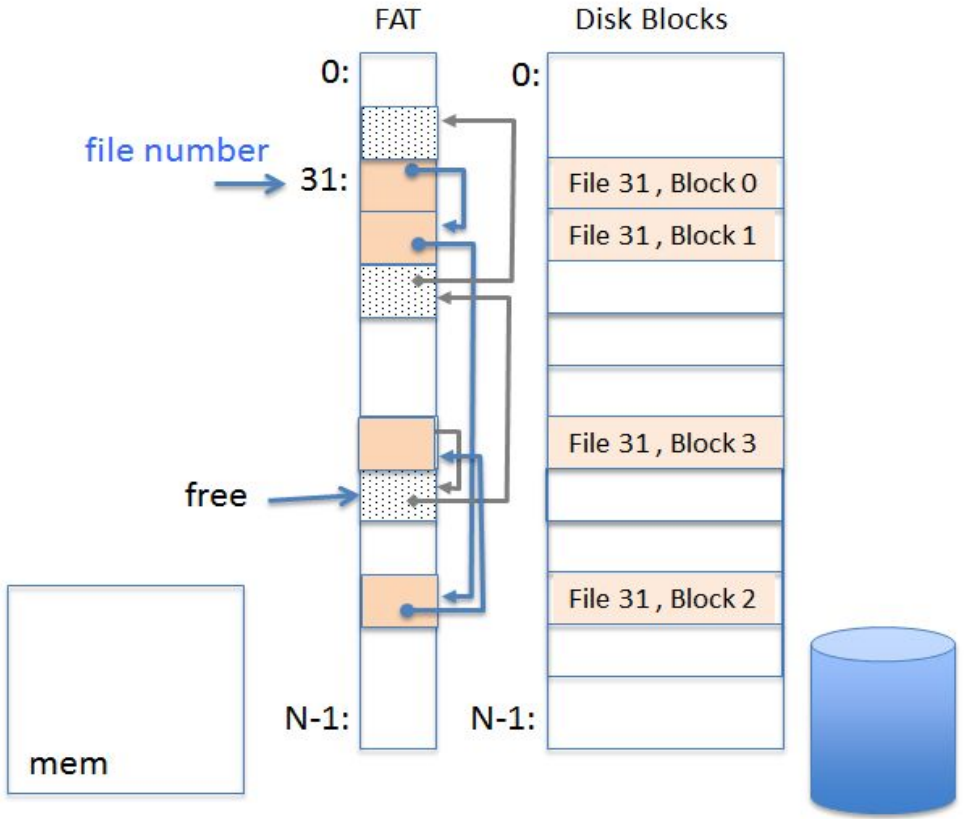
- We will look at three implementations: FAT, FFS, and NTFS
- In all three implementations, each file has a unique **file number**
  - To get the contents of a file, all you need is the file number
- In fact, file descriptions consist of a file number + offset

Implementation 1: FAT

- Developed by Microsoft in the 1970s
- File = linked list of blocks
- Bookkeeping done by the File Allocation Table (FAT)

Implementation 1: FAT

- FAT is table with entries that correspond one-to-one with blocks and stores linked list information
- File number is index of the first block of the file
- Follow block list to seek in file
- Grow file by appending to list
- Unused blocks in FAT are tracked via a free list (also linked)



FAT Assessment

How well does FAT work with:

- Sequential Access?
- Random Access?
- External Fragmentation?
- Internal Fragmentation? Small files?
- Big files?
- Locality for files and metadata?

FAT Assessment

- Sequential Access?
  - Good!
- Random Access?
  - Bad. Traverse linked list.
- External Fragmentation?
  - No external fragmentation.
- Internal Fragmentation? Small files?
  - Too many small files can lead to severe internal fragmentation
- Big files?
  - Usually not problem.
- Locality for files and metadata?
  - Bad, FAT stores the metadata separately from the data, files allocated as linked lists.

FAT Assessment

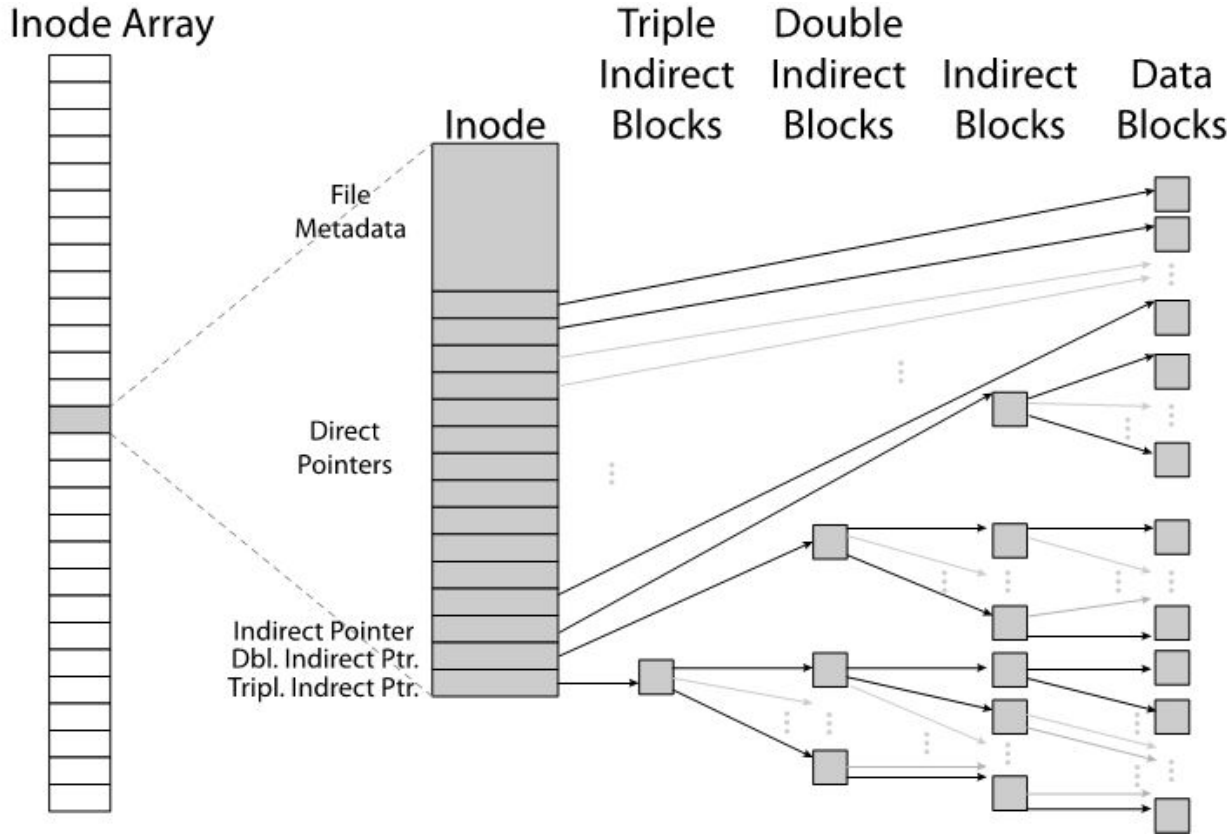
- Simple to implement, but poor performance and little security

Implementation 2: FFS

- Developed by Berkeley for BSD!
- File number is index into inode array
- Each inode corresponds to a file and contains its metadata (but not name)

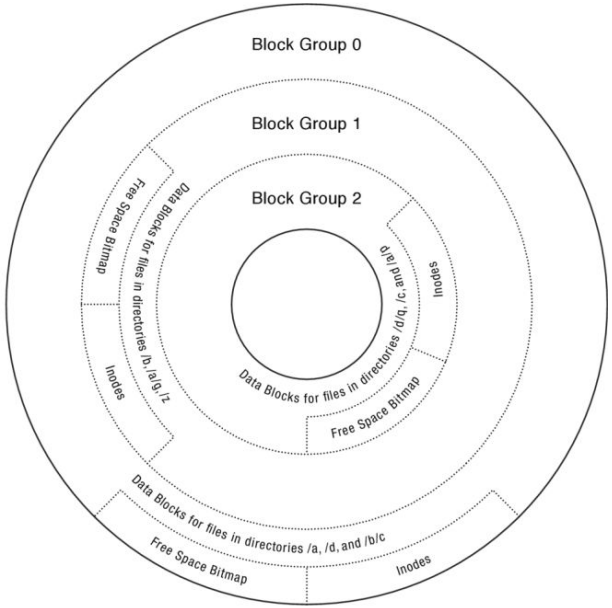
Beastie, the BSD daemon

Implementation 2: FFS



FFS Optimization for HDDs

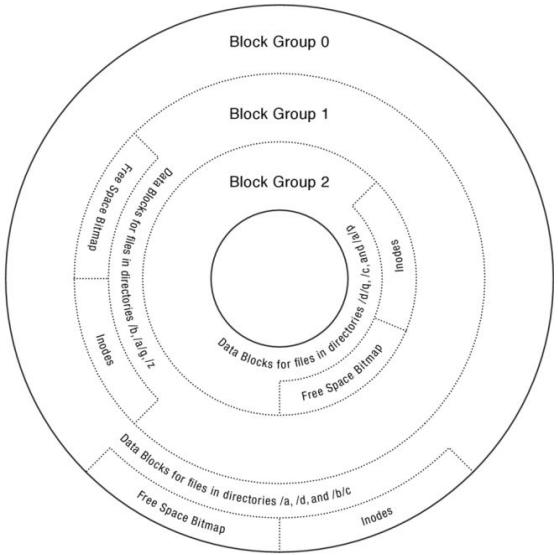
To optimize FFS for HDDs, we split the disk into block groups:



FFS Optimization for HDDs

To optimize FFS for HDDs, we split the disk into block groups.

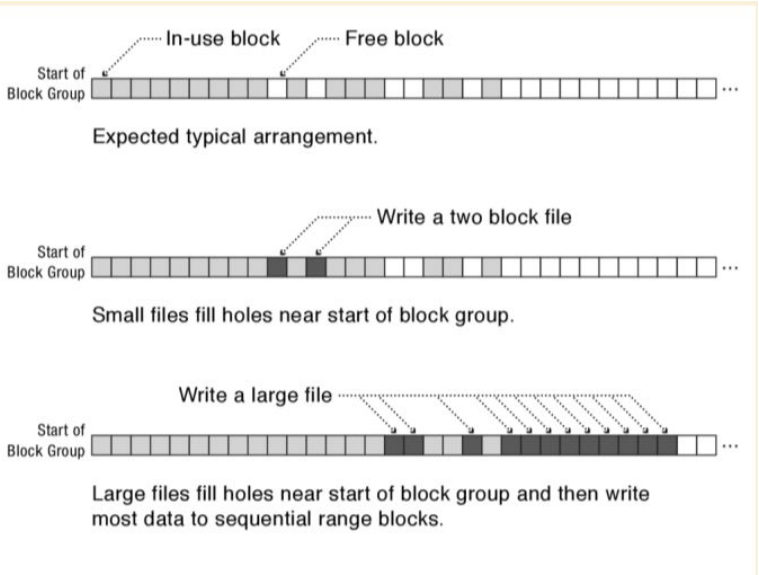
- Put directory (which is just a file) and its files in common block group



FFS Optimization for HDDs

To allocate new blocks when expanding a file, use a first-free heuristic.

- Reason: for large files, will help with locality.
- Important: keep 10% or more free



FFS Assessment

How well does FFS work with:

- Sequential Access?
- Random Access?
- External Fragmentation?
- Internal Fragmentation? Small files?
- Big files?
- Locality for files and metadata?

FFS Assessment

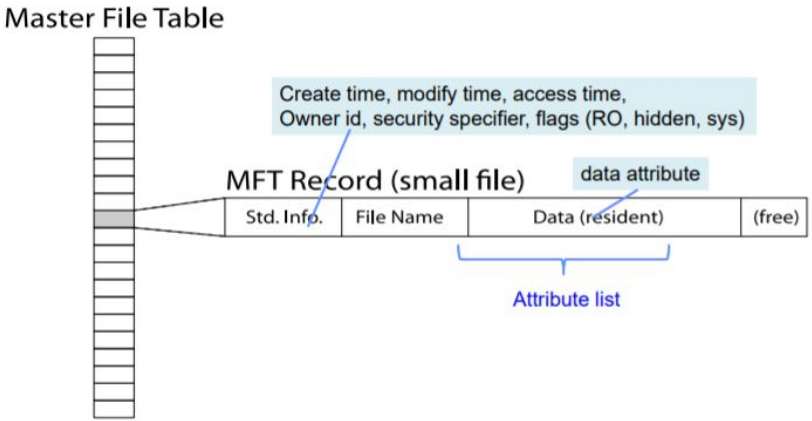
- Sequential Access?
  - Good!
- Random Access?
  - Good! Traverse pointers in the inode.
- External Fragmentation?
  - No external fragmentation.
- Internal Fragmentation? Small files?
  - Internal fragmentation for tiny files (an 8 byte file requires both an inode and a data block), but the direct pointers make it generally efficient for small files.
- Big files?
  - Good! Indirect pointers.
- Locality for files and metadata?
  - Yes. The layout (block groups) on disk tries to ensure that.

FFS Assessment

- More complicated than FAT, but has many performance advantages

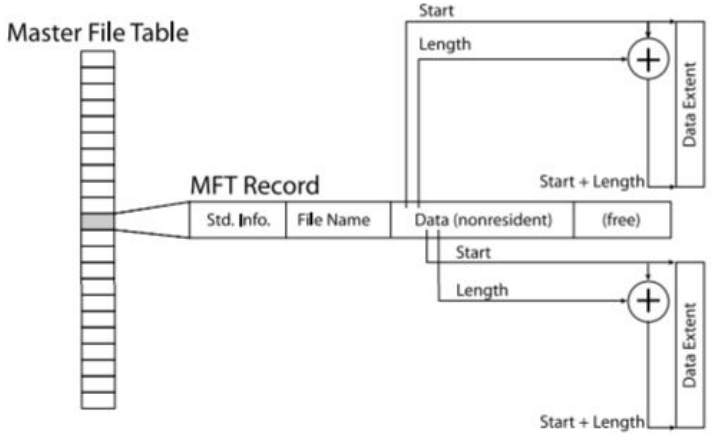
Implementation 3: NTFS

- The file system currently used by Windows
- Uses a Master File Table instead of inode array
- Index into the MFT with file number
- Each entry in MFT contains file metadata and data



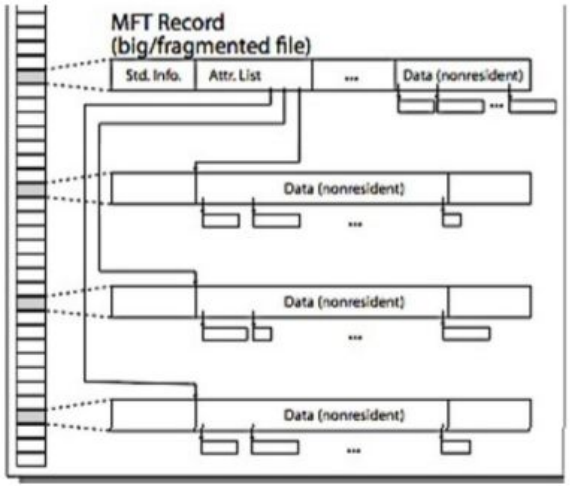
Implementation 3: NTFS

- If file is too big to fit in Master File Table, store some data in extents (variable length chunks of data)
- Analogous to single pointers in FFS



Implementation 3: NTFS

- If file is super big, include pointer to lists of extents (analogous to the double pointers in FFS)



NTFS vs FFS

- Most fundamental difference is that NTFS uses variable length extents while FFS uses fixed size blocks

What about Directories?

- A directory is just a file that contains **<name : file number>** mappings!
  - One entry corresponding to each file/subdirectory in the directory

- In FAT, file metadata is stored in the directory

More about Directories

- The **<name: file number>** pairs in the directory file are called **hard links**
  - Using the link() syscall, a file can have more than 1 hard link. (That is, a file can be part of two different directories).
  - We only use the term “hard link” if the file system supports multiple hard links to a file.
  - A file will not be removed until all the hard links to the file are removed (uses a reference count, so **FAT does not support hard links**!)
- A **soft link** is a special type of entry in the directory file. It is a entry of the form **<name: path>**. Every time the name is accessed, the OS will lookup the file corresponding to the path, and access that file.
  - Use the symlink() syscall to create soft links
  - No reference count
  - If path doesn’t exist, lookup will fail.

Durability

Durability

- How do we prevent loss of data due to disk failure?
- Two strategies:
  - RAID
  - Erasure codes

RAID

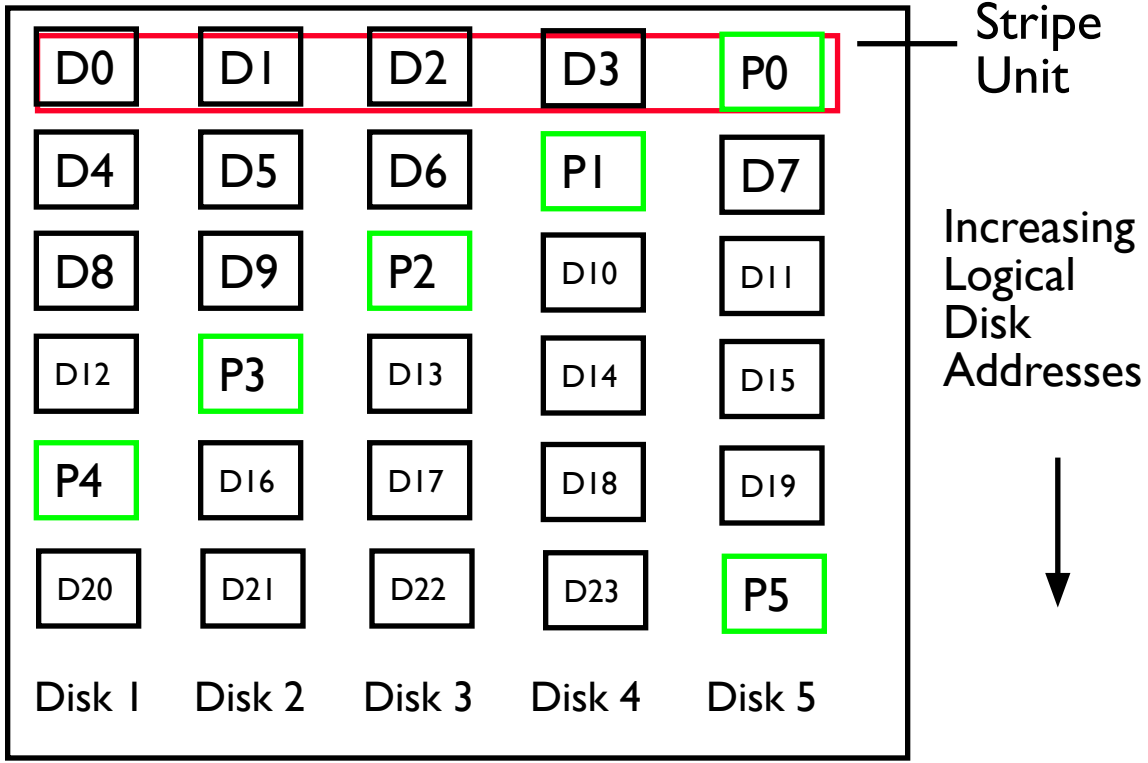
- RAID: redundant array of inexpensive disks
- Idea: use redundant arrays of inexpensive disks to make filesystems more durable
- Many different types of RAID:
  - We'll focus on RAID 1 and RAID 5+

RAID 1: Disk Mirroring/Shadowing

- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation fully synchronized (hard to do exactly)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk already attached to system to be used for immediate replacement

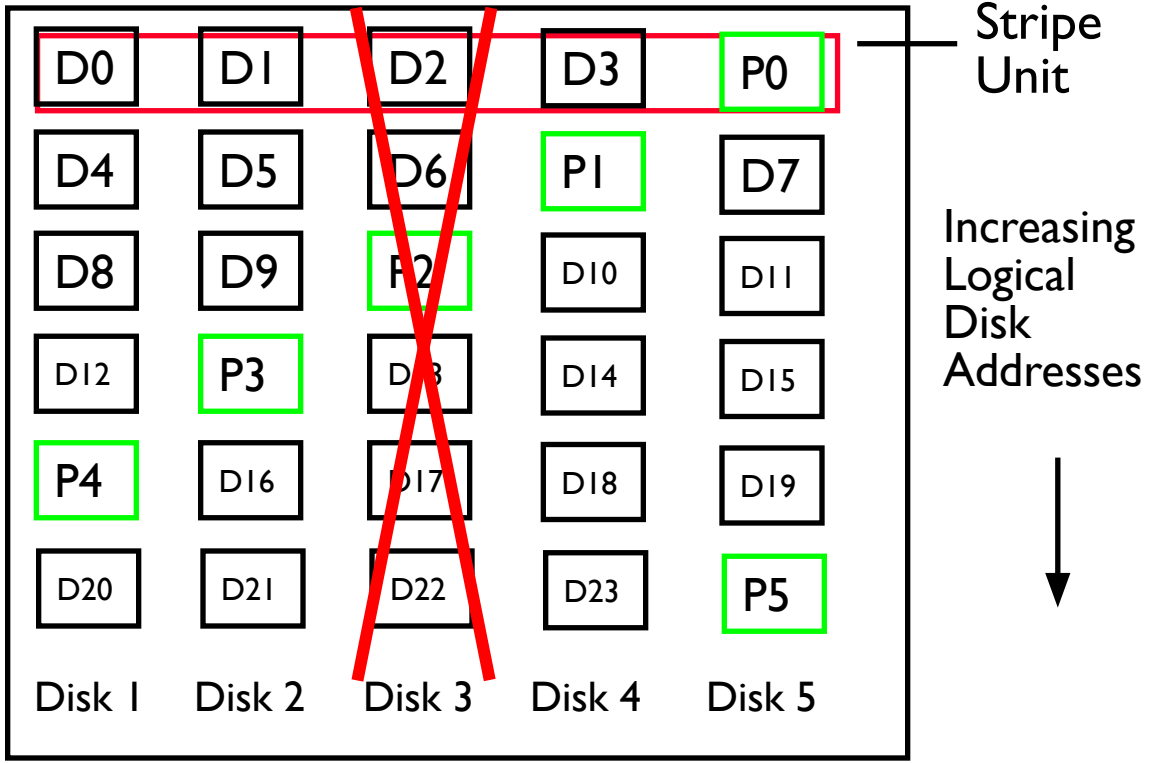
RAID 5+: High I/O Rate Parity

- Data striped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk



RAID 5+: High I/O Rate Parity

- Data striped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data blocks in stripe
  - P0=D0⊕D1⊕D2⊕D3
  - Can destroy any one disk and still reconstruct data
  - Suppose Disk 3 fails, then can reconstruct: D2=D0⊕D1⊕D3⊕P0



General Erasure Codes

- Reed-Solomon Error Correcting Codes
  - A bit mathy: has to do with polynomials of degree **m** over finite fields
  - The result: in **n** disks, can tolerate **n-m** failures
  - Incredibly durable!

Reliability

Reliability vs Durability

- Durability: being able to recover from a disk failure
- Reliability: after recovering from a disk failure, want data in a consistent state
  - For example: suppose a computer is transfering funds from one bank account to another
  - If the computer crashes after the withdrawal but before the deposit, we may be able to recover the data, but the data might be wrong

Reliability vs Durability

- Durability: being able to recover from a disk failure
- Reliability: after recovering from a disk failure, want data in a consistent state
  - For example: suppose a computer is transfering funds from one bank account to another
  - If computer crashes after the withdrawal but before the deposit, we may be able to recover the data, but the data might be wrong
- Basic idea of solution: want some operations (like withdrawal and deposit) to be atomic
  - Either both happen, or neither happen

Transactions

- A transaction is an atomic sequence of reads and writes that takes the system from one consistent state to another

- Transactions have four properties:
  - **A**tomicity (see above)
  - **C**onsistency (Consistency ensures that a transaction can only bring the database from one valid state to another)
  - **I**solation (concurrent execution of transactions is okay)
  - **D**urability (once a transaction is committed, it will not be erased in case of disk failure)

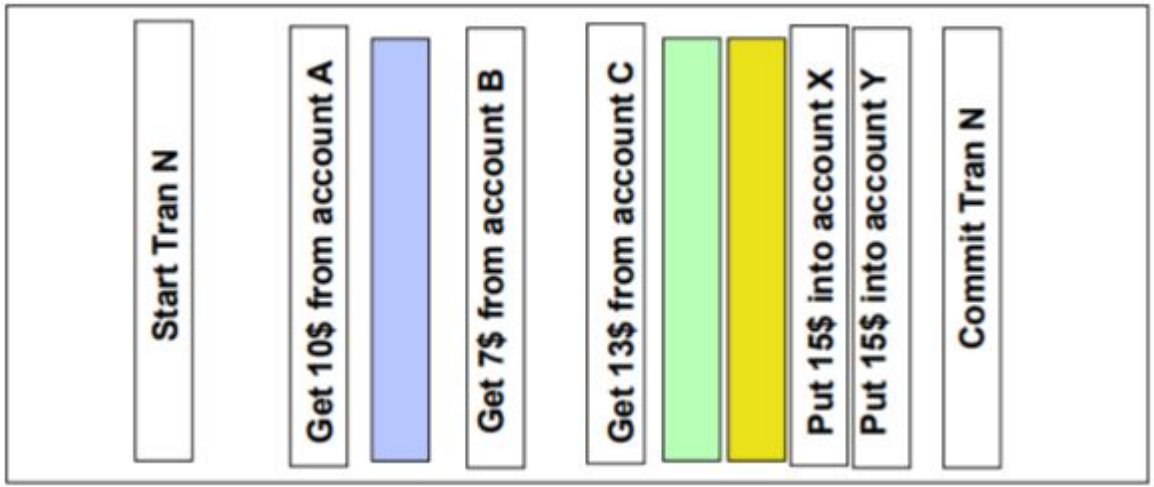
Transactions

- A transaction is an atomic sequence of reads and writes that takes the system from one consistent state to another

- Want to update the disk by transactions, not by individual reads and writes.
- Two ways to do this:
  - Journaling filesystem
  - Log structured file system
- Both implementations use a **log** (a buffer on the disk)

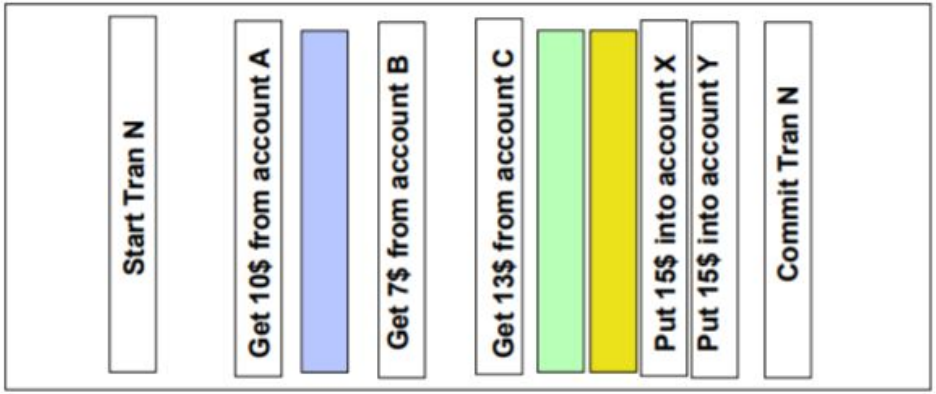
Logs

- Assuming read/write are atomic operations, can implement transactions by writing to a log:



Journaling Filesystem

- Instead of writing to the desired blocks on disk directly, write to log
- Once a whole transaction is written to log, the disk will apply the changes in the transaction to itself
- Once change is applied, remove transaction from log



Journaling Filesystem

- Instead of writing to the desired blocks on disk directly, write to log
- Once a whole transaction is written to log, the disk will apply the changes in the transaction to itself
- Once change is applied, remove transaction from log

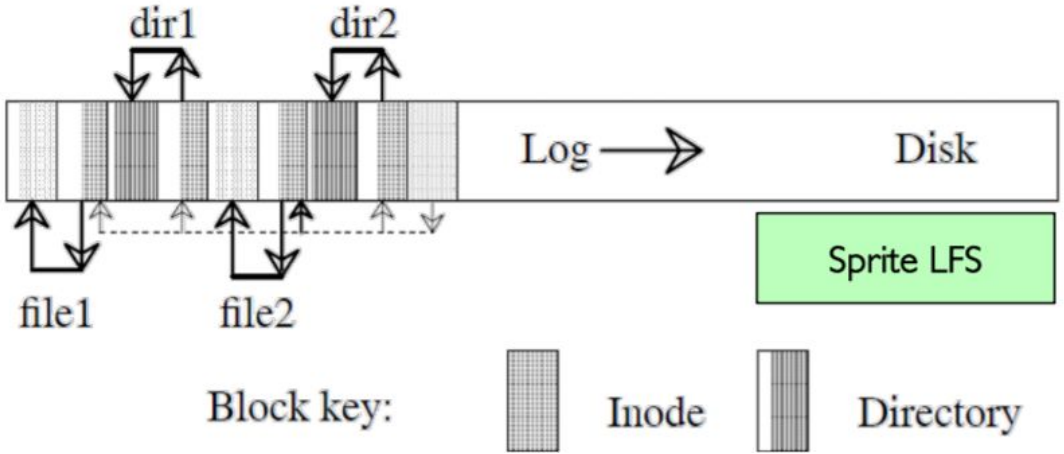
- If system crashes when writing to log, will get an incomplete transaction that will not be applied
- If system crashes when applying changes disk, we can simply re-apply the transaction (by looking at the log) after recovering

Journaling Filesystem

- The ext3 filesystem (which used to be the default filesystem for Linux) is basically FFS + logs

Log Structured File System

- Data stays in log form (basically, the file system is the log itself)
  - To create two files: dir1/file1, dir2/file2



Break!

Logging Tradeoffs

- Is it faster to write 100 random 512 byte sectors in a normal file system, or in a log-structured file system ?

- Is it faster to write 100 1MB sequential sectors in a normal file system, or in a log-structured file system?

Logging Tradeoffs

- Is it faster to write 100 random 512 byte sectors in a normal file system, or in a log-structured file system ?
  - The log-structured file system might perform better because writes are written sequentially in the log. During the write-back phase, better scheduling could lead to faster write-back than 100 random writes.

- Is it faster to write 100 1MB sequential sectors in a normal file system, or in a log-structured file system ?
  - The normal file system might perform better because the costs of logging dominate. In the non-log-structured file system the writes are largely sequential.

Demand Paging

When you run out of physical memory for applications: store memory pages in disk to make space, pull them out of disk when necessary.

Ideally, store pages on disk that will not be used for a long time ⇒ keep most used pages in memory

Think of physical memory as a “cache” for disk.

Terminology

Working set of a process: Pages that are used by a process within a given time interval.

Resident set of a process: Set of pages that are actually loaded into physical memory.

Ideally, the working set is a subset of the resident set. Not usually the case, but we can try to maximize the overlap.

Page Replacement Policies

Much like caches, need to evict unused pages.

Ideally, MIN:
replace the page that will be referenced furthest in the future or not at all

LRU? Good idea, but implementing LRU is inefficient in hardware.
Approximating LRU is good enough: Clock

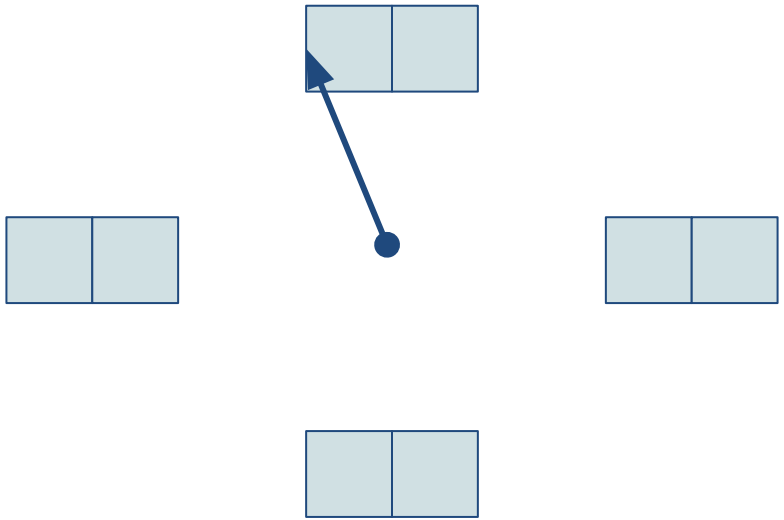
Key idea: Replace an old page, not the oldest page.

Clock Algorithm

- Each cache entry (or PTE) keeps track of extra bit called use bit (a.k.a. clock bit, reference bit)
  - Use bit 1 means young; use bit 0 means old
- Upon hit, set the entry's use bit to 1
- Upon miss
  - Clock hand sweeps over entries until finding one to evict:
    - If entry has use bit 1, clear it (set it to 0)
    - If entry has use bit 0, evict this entry
  - Sets use bit of new entry to 1

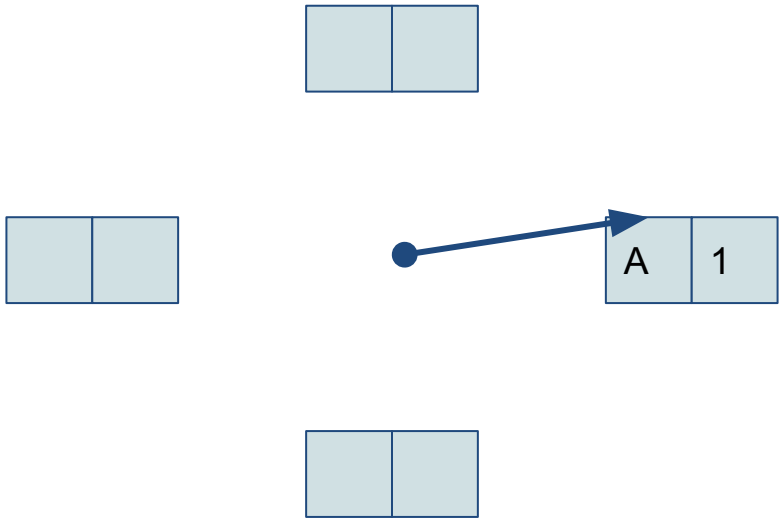
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



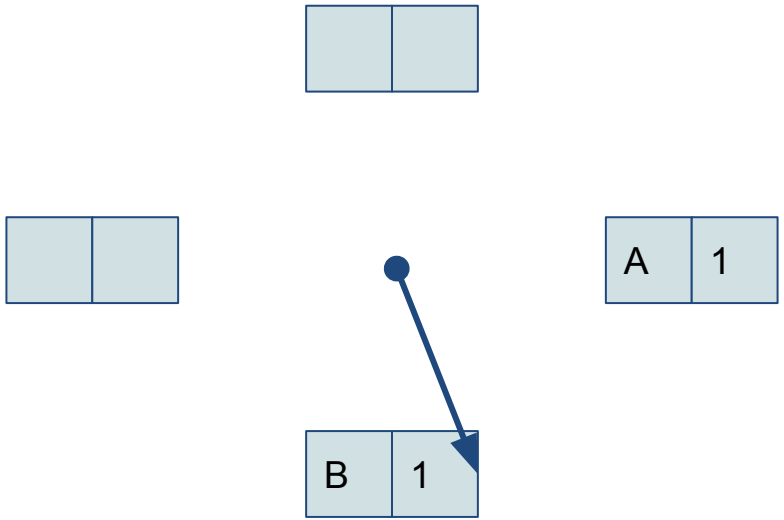
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



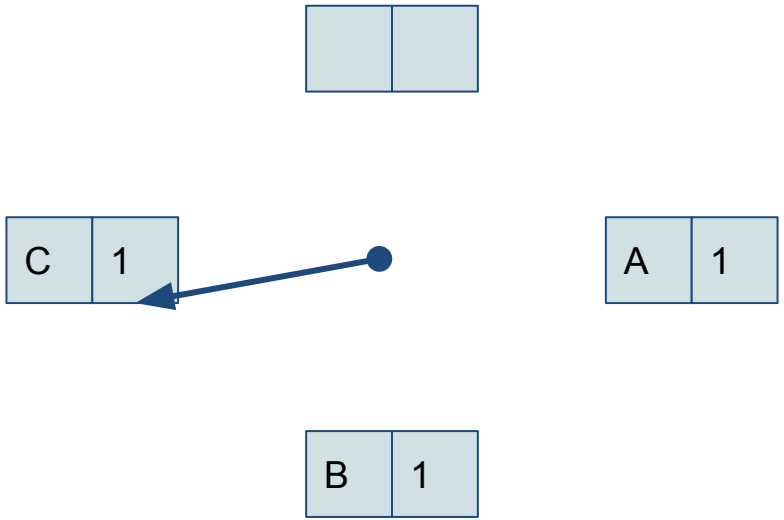
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



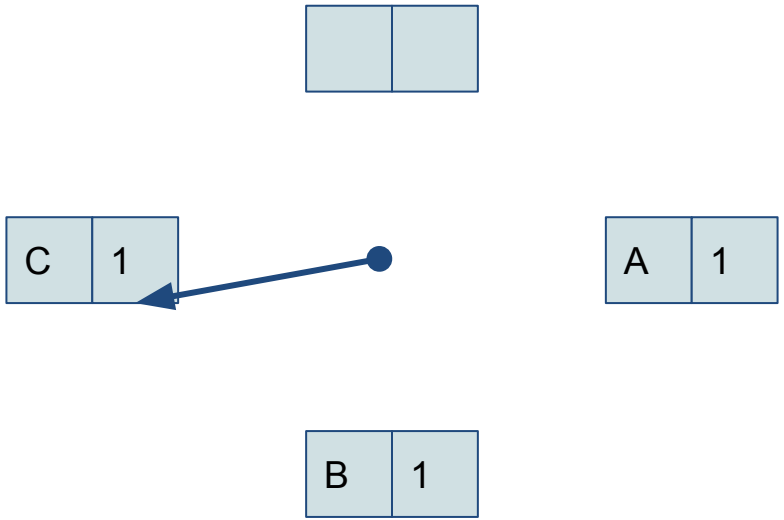
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



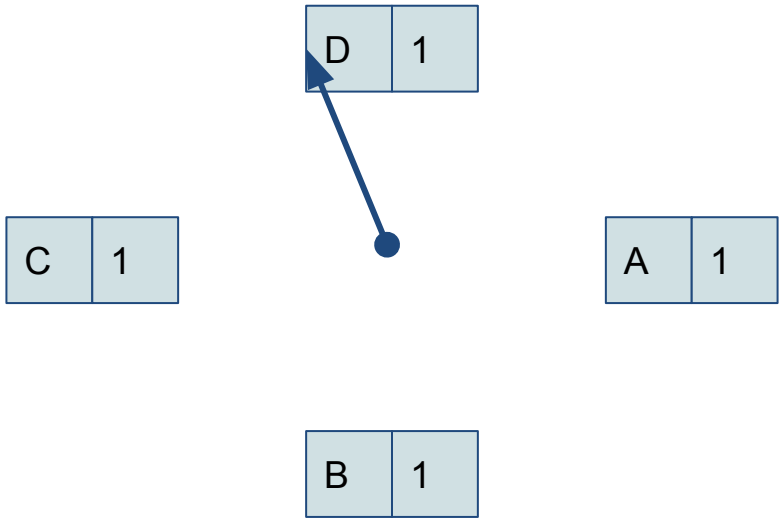
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



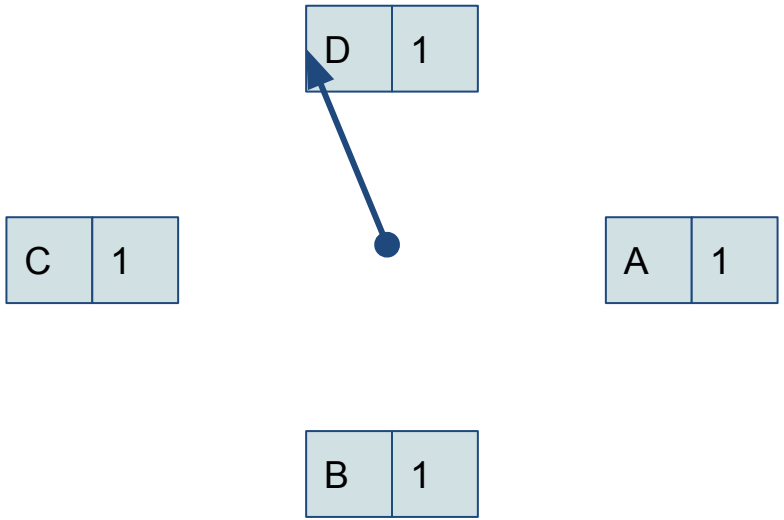
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



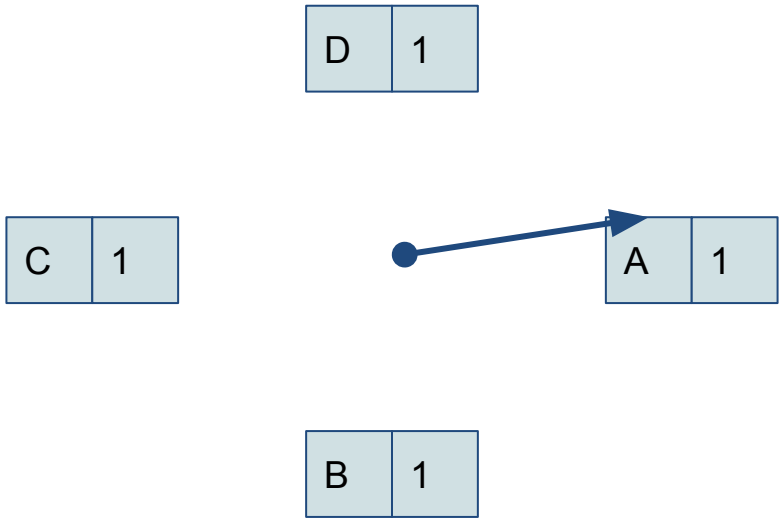
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



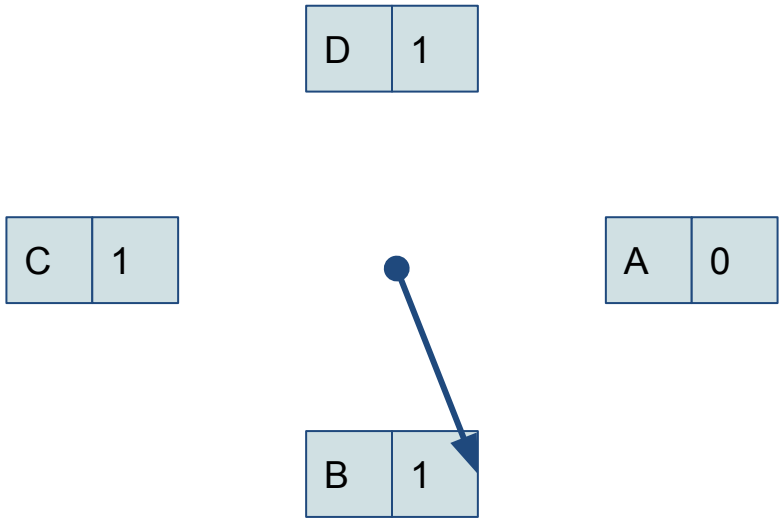
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



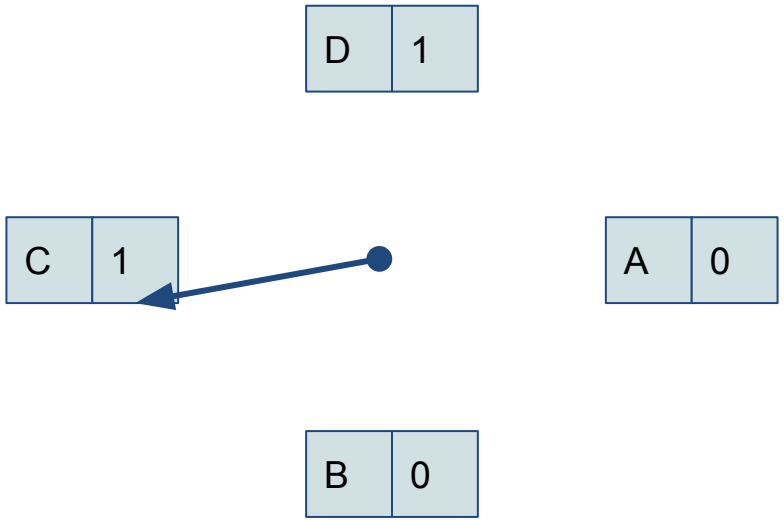
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



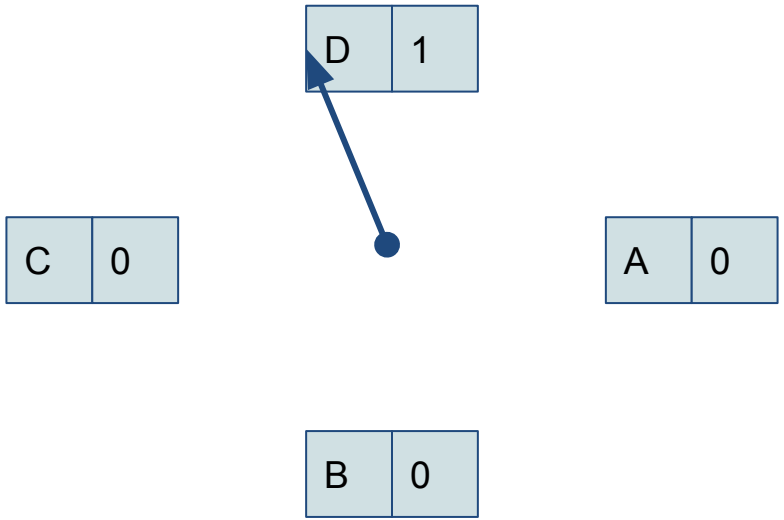
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



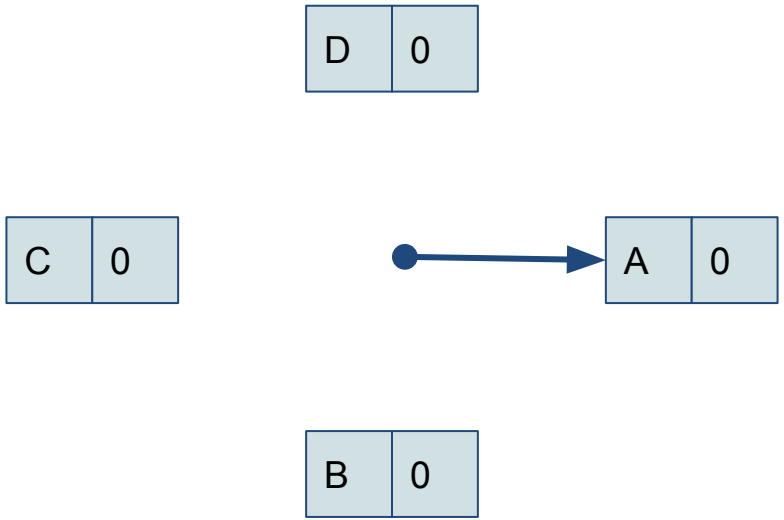
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



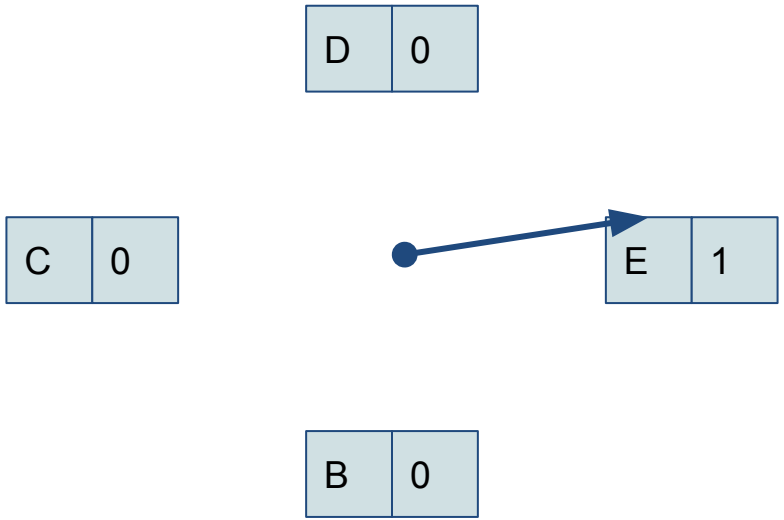
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



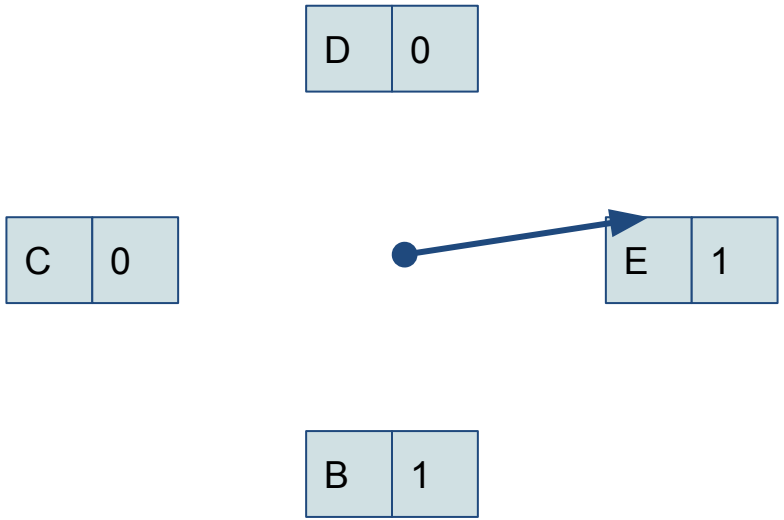
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



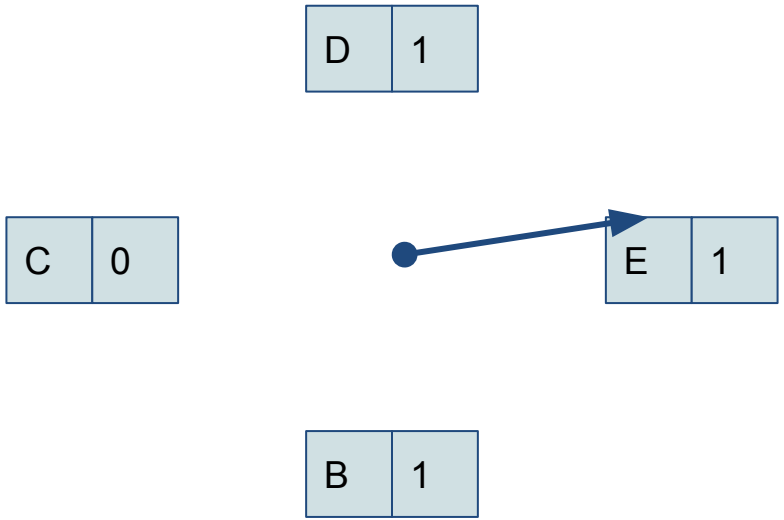
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



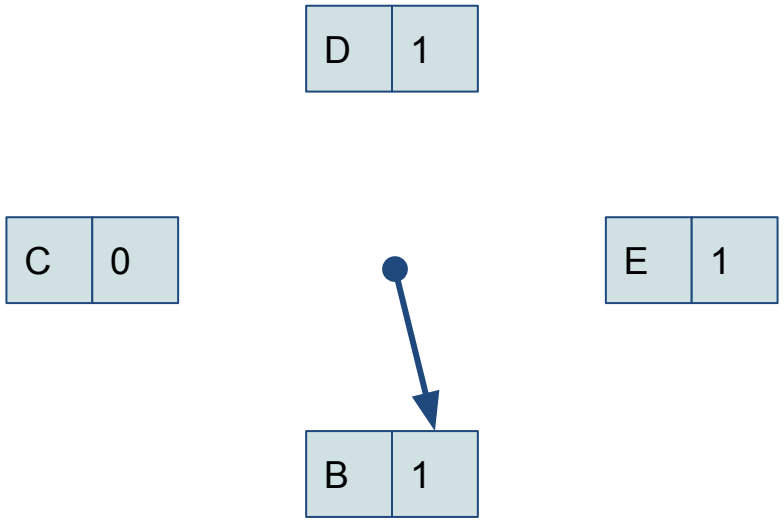
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



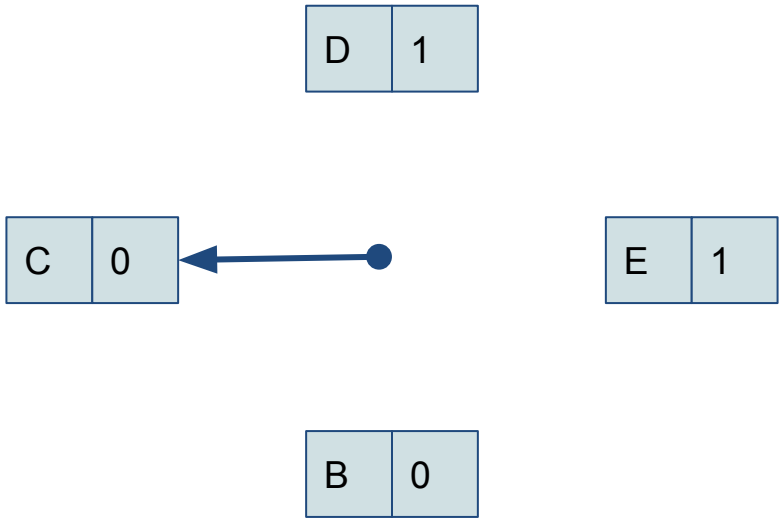
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



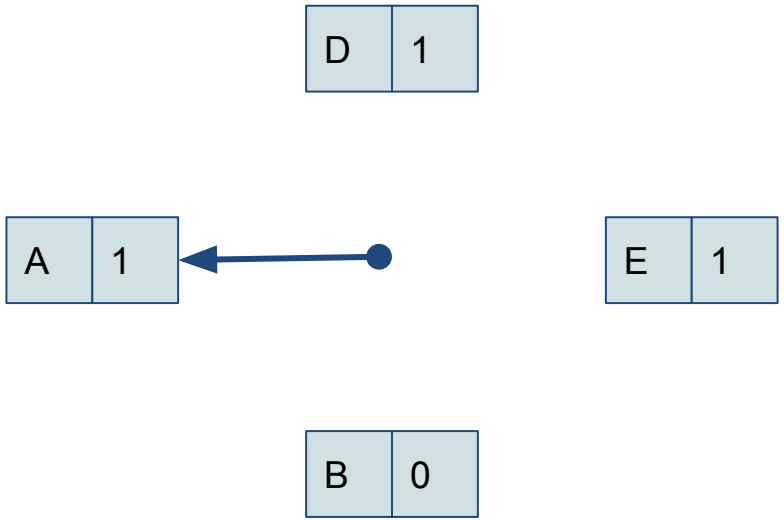
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



Networks and Distributed Systems

Distributed Consensus Making

Problem

- Previously, we talked about transactions and logging on a single machine
  - Consistency!
- What if we want to coordinate multiple computers, i.e.
  - Ensure consistent transactions in a sharded database

Why we care:

- For instance, for durability, we could duplicate a lot of machines, each with a journaling/log-structured file system.
- Then keeping the machines consistent with each other is an important problem

Solution

Two phase commit:

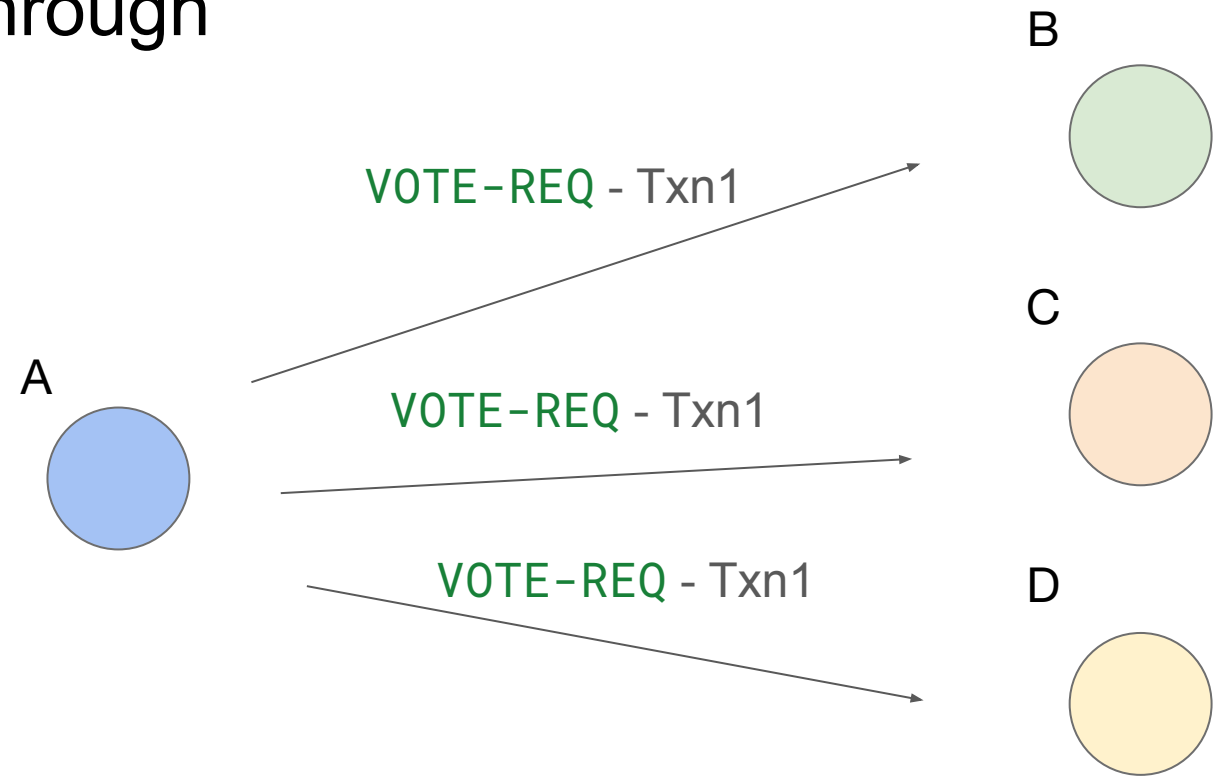
- One machine is designated as the coordinator, everyone else is a participant
- The coordinator requests that all participants vote to commit or rollback the transaction
- Participants record vote in its own log, then tell coordinator
  - If anyone votes to abort, coordinator writes "Abort" in its log and tells everyone to abort; each records "Abort" in log
- After all participants respond that they are prepared, then the coordinator writes "Commit" to its log
  - Then asks all nodes to commit; they respond with ACK

2PC Walkthrough (Successful `COMMIT`)

Txn 1
`INSERT INTO…`

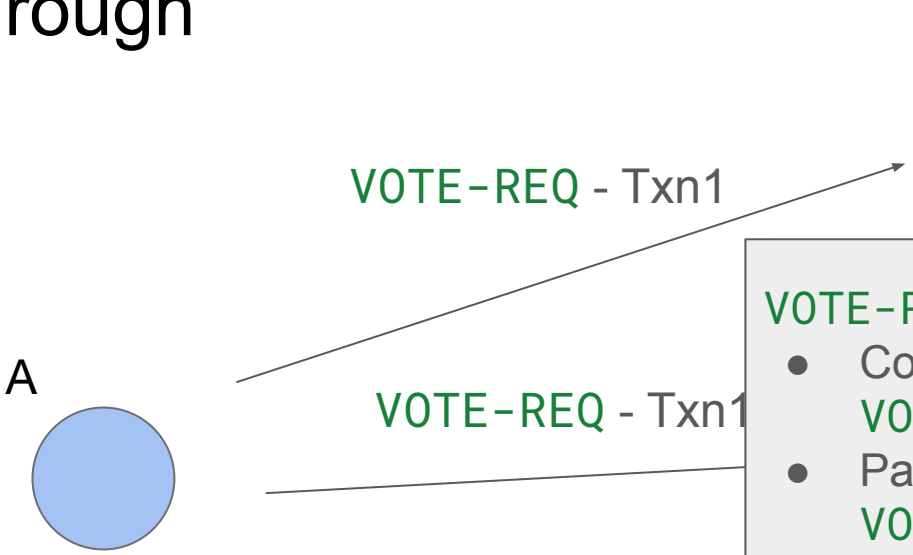
2PC Walkthrough

Txn 1
INSERT
INTO…



2PC Walkthrough

Txn 1
`INSERT INTO…`

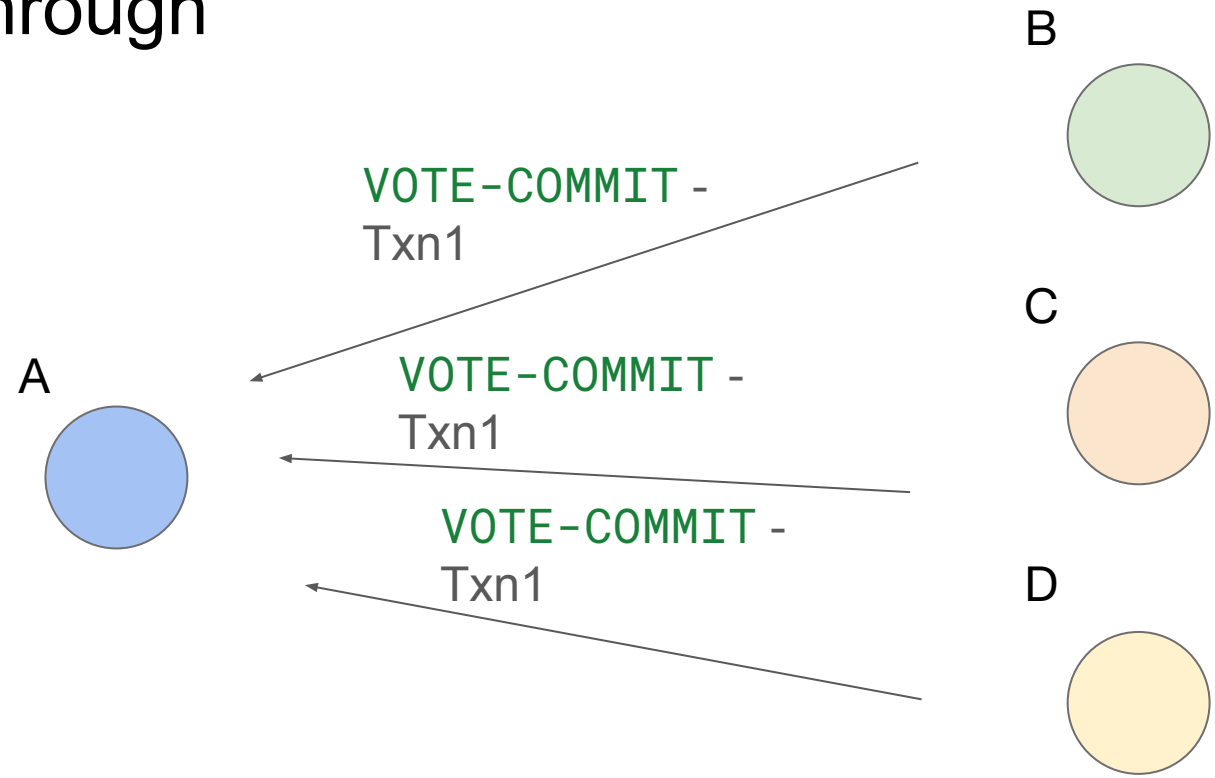


`VOTE-REQ`

- Coordinator sends `VOTE-REQ` message
- Participants generate `VOTE-COMMIT` or `VOTE-ABORT`, flush to disk
- Respond yes/no
- **What happens if participant recovers and sees no `VOTE-REQ` record?**

2PC Walkthrough

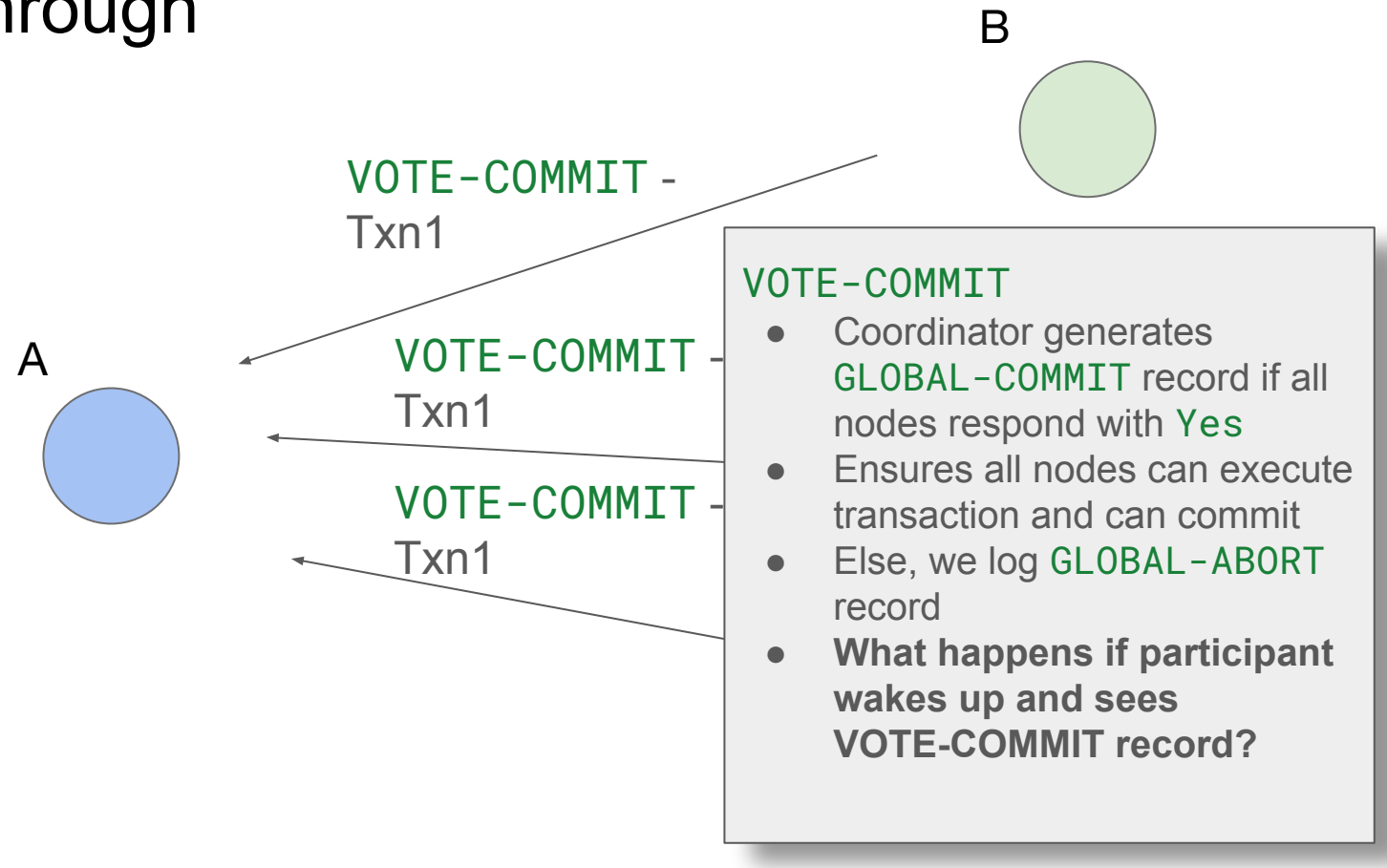
Txn 1
INSERT
INTO…



2PC Walkthrough

Txn 1
INSERT
INTO…

COMMIT



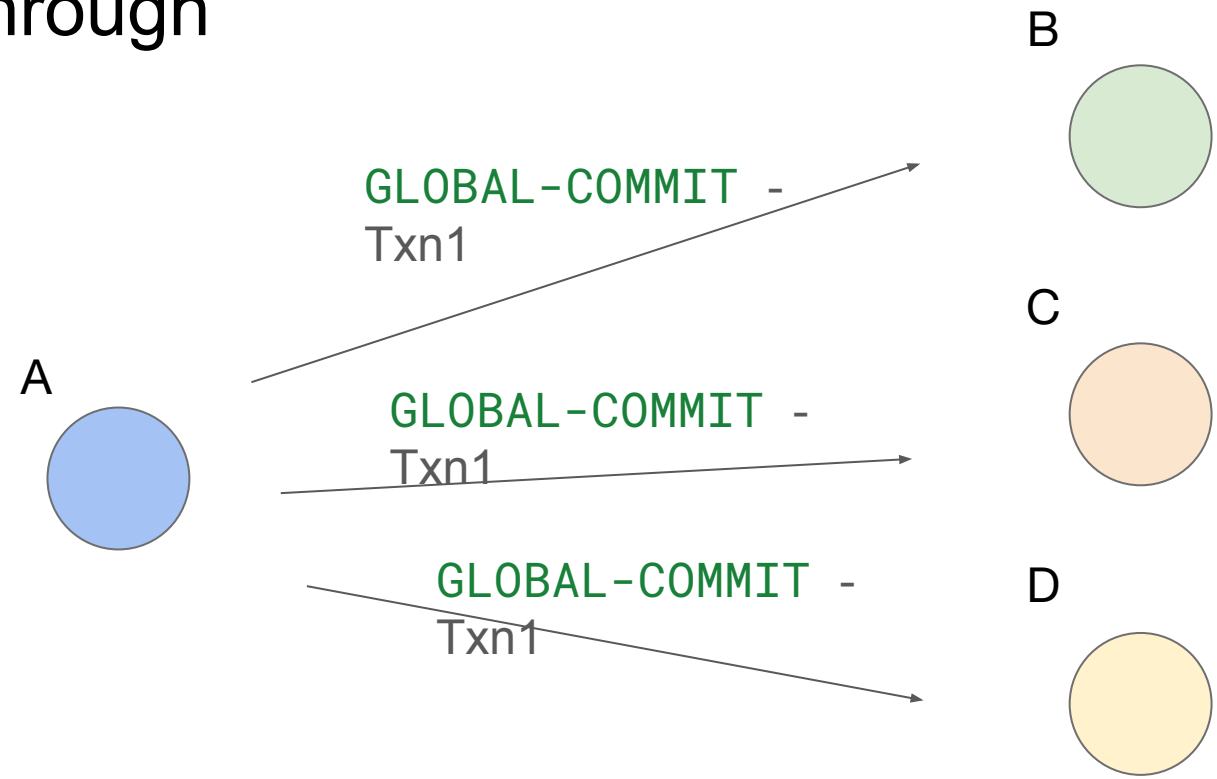
VOTE-COMMIT

- Coordinator generates GLOBAL-COMMIT record if all nodes respond with Yes
- Ensures all nodes can execute transaction and can commit
- Else, we log GLOBAL-ABORT record
- **What happens if participant wakes up and sees VOTE-COMMIT record?**

2PC Walkthrough

Txn 1
INSERT
INTO…

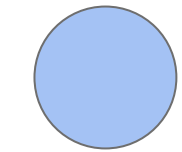
COMMIT



2PC Walkthrough

Txn 1
INSERT
INTO…

COMMIT



GLOBAL-COMMIT
Txn1

GLOBAL-COMMIT
Txn1

GLOBAL-COMM
Txn1

B

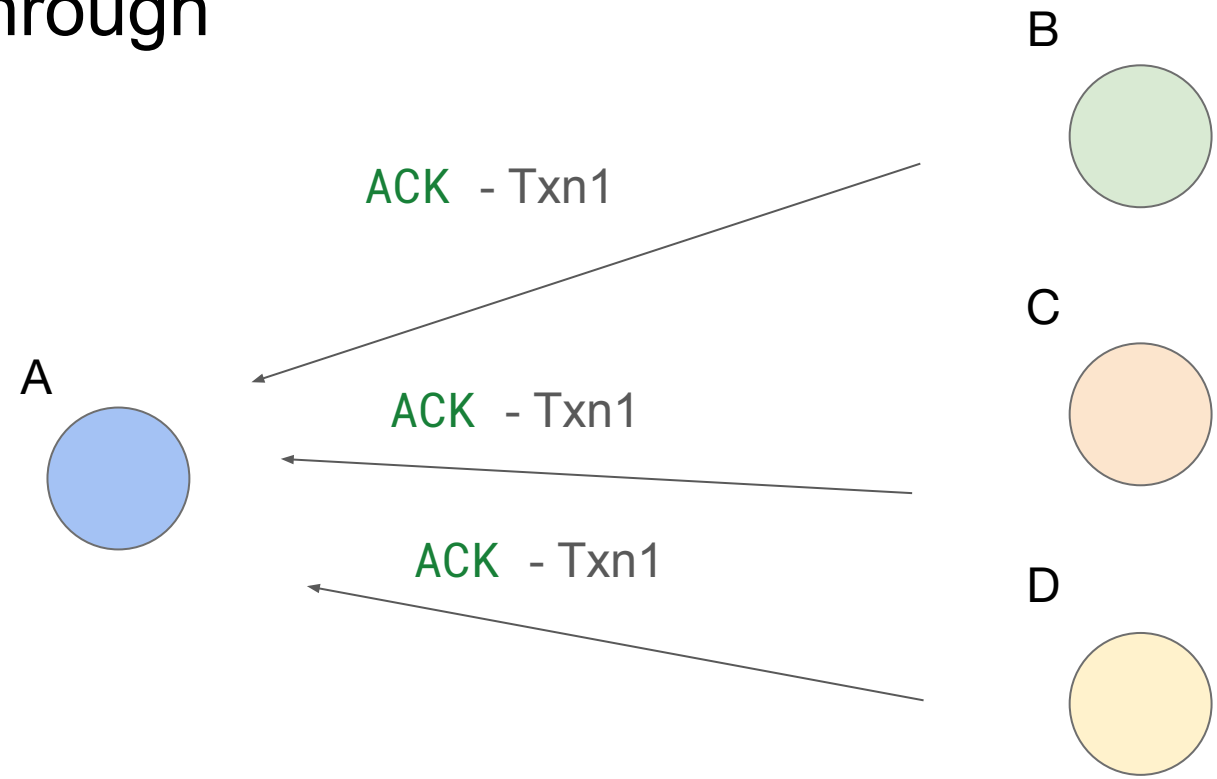
GLOBAL-COMMIT

- Coordinator broadcasts result of vote COMMIT/ABORT
- Participants log result, and respond with ACK
- **What happens if coordinator/participant wakes up and sees GLOBAL-COMMIT record?**

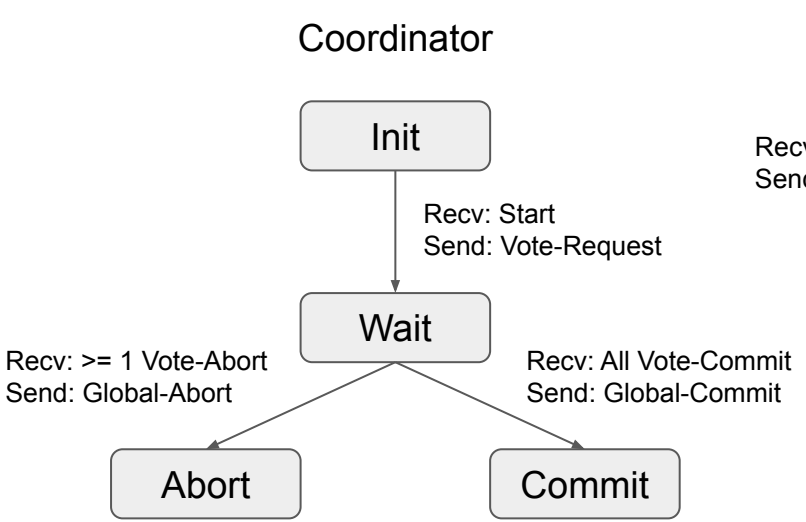
2PC Walkthrough

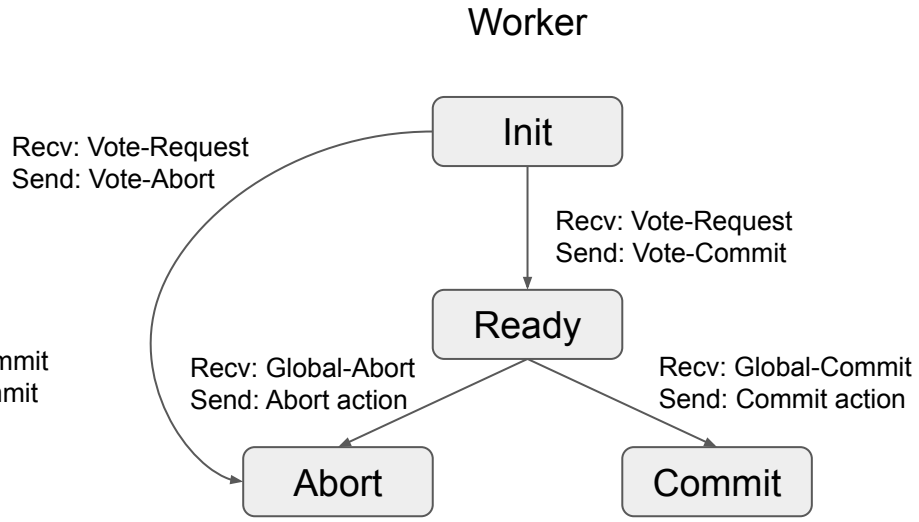
Txn 1
INSERT
INTO…

COMMIT



2PC State Machines





If a timeout occurs while coordinator is in wait state, then Global-Abort is sent to all workers.

If worker times out in ready, then that means the coordinator crashed. This is solved in one of two ways:

1. Can block and continue to ask coordinator for Global-*.
2. (Optimization) Can ask other workers what message they got, if any, and decide on what to do.

2PC Problem

A Two-Phase Commitment: You maintain a distributed key-value store with one coordinator server (C1) and 2 worker servers (W1 and W2).

Consider the following series of events for adding a K-V pair (Obi, 1) under a perfect execution of two-phase commit protocol.

1	C1: Log (BEGIN)
2	C1: Log (PREPARE PUT "Obi”, “1")
3	C1 -> W1: VOTE-REQ (PUT "Obi”, “1”)
4	W1: Log (PREPARE PUT "Obi”, “1")
5	W1 -> C1: Send (VOTE-COMMIT)
6	C1 -> W2: VOTE-REQ (PUT "Obi”, “1”)
7	W2: Log (PREPARE PUT "Obi”, “1")
8	W2 -> C1: Send (VOTE-COMMIT)
9	C1: Log (GLOBAL-COMMIT)

10	C1 -> W1: Send (GLOBAL-COMMIT)
11	W1: Log (COMMIT)
12	W1: PUT "Obi”, “1"
13	W1 -> C1: Send (ACK)
14	C1 -> W2: Send (GLOBAL-COMMIT)
15	W2: Log (COMMIT)
16	W2: PUT "Obi”, “1"
17	W2 -> C1: Send (ACK)
18	C1: PUT "Obi, “1"
19	C1: Log (END)

- Server C1 crashes right after step 10. What actions will it take upon recovery? Assuming no other crashes, will the key-value pair be added to any of the servers and why?
- Give a timestep right after which a single server crash could block the entire system (no progress is made) until it is brought back online.

1	C1: Log (BEGIN)
2	C1: Log (PREPARE PUT "Obi", “1")
3	C1 -> W1: VOTE-REQ (PUT "Obi", “1”)
4	W1: Log (PREPARE PUT "Obi", “1")
5	W1 -> C1: Send (VOTE-COMMIT)
6	C1 -> W2: VOTE-REQ (PUT "Obi", “1”)
7	W2: Log (PREPARE PUT "Obi", “1")
8	W2 -> C1: Send (VOTE-COMMIT)
9	C1: Log (GLOBAL-COMMIT)

10	C1 -> W1: Send (GLOBAL-COMMIT)
11	W1: Log (COMMIT)
12	W1: PUT "Obi", “1"
13	W1 -> C1: Send (ACK)
14	C1 -> W2: Send (GLOBAL-COMMIT)
15	W2: Log (COMMIT)
16	W2: PUT "Obi", “1"
17	W2 -> C1: Send (ACK)
18	C1: PUT "Obi, “1"
19	C1: Log (END)

- Server C1 crashes right after step 10. What actions will it take upon recovery? Assuming no other crashes, will the key-value pair be added to any of the servers and why?
- C1 finds GLOBAL-COMMIT in its log, so it resends the GLOBAL-COMMITs. The KV pair will be added.

1	C1: Log (BEGIN)
2	C1: Log (PREPARE PUT "Obi", “1")
3	C1 -> W1: VOTE-REQ (PUT "Obi", “1”)
4	W1: Log (PREPARE PUT "Obi", “1")
5	W1 -> C1: Send (VOTE-COMMIT)
6	C1 -> W2: VOTE-REQ (PUT "Obi", “1”)
7	W2: Log (PREPARE PUT "Obi", “1")
8	W2 -> C1: Send (VOTE-COMMIT)
9	C1: Log (GLOBAL-COMMIT)

10	C1 -> W1: Send (GLOBAL-COMMIT)
11	W1: Log (COMMIT)
12	W1: PUT "Obi", “1"
13	W1 -> C1: Send (ACK)
14	C1 -> W2: Send (GLOBAL-COMMIT)
15	W2: Log (COMMIT)
16	W2: PUT "Obi", “1"
17	W2 -> C1: Send (ACK)
18	C1: PUT "Obi, “1"
19	C1: Log (END)

- Give a timestep right after which a single server crash could block the entire system (no progress is made) until it is brought back online.
- C1 crashes after step 8 or 9.

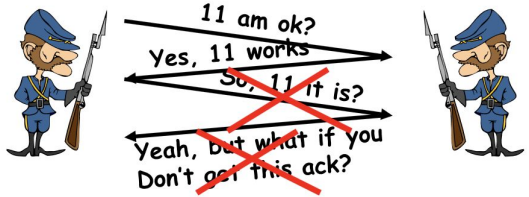
Problems and Paradoxes

- Active recall: Explain the General's Paradox and why it doesn't apply to 2 phase commit.

General's Paradox (con't)

Can messages over an unreliable network be used to guarantee two entities do something simultaneously?

– Remarkably, "no", even if all messages get through



– No way to be sure last message gets through!

– In real life, use radio for simultaneous (out of band) communication

So, clearly, we need something other than simultaneity!

Problems and Paradoxes

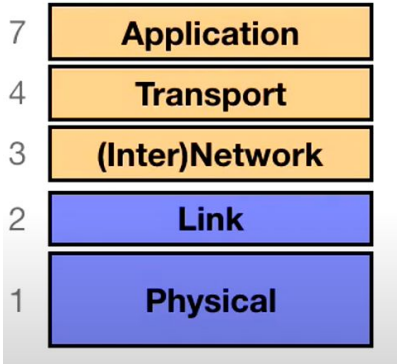
General's paradox:

- Messages over an unreliable network cannot guarantee entities to do something simultaneously
- Does not apply to 2PC because the machines are not trying to agree to do something simultaneously in 2PC.

Networks

The Internet

- Internet: used to move data from one location to another
- To do so, need protocols (agreements on how to communicate)
- The protocol for communicating on the internet has a layered structure:



- Each layer provides more services upon the previous layer

The Internet

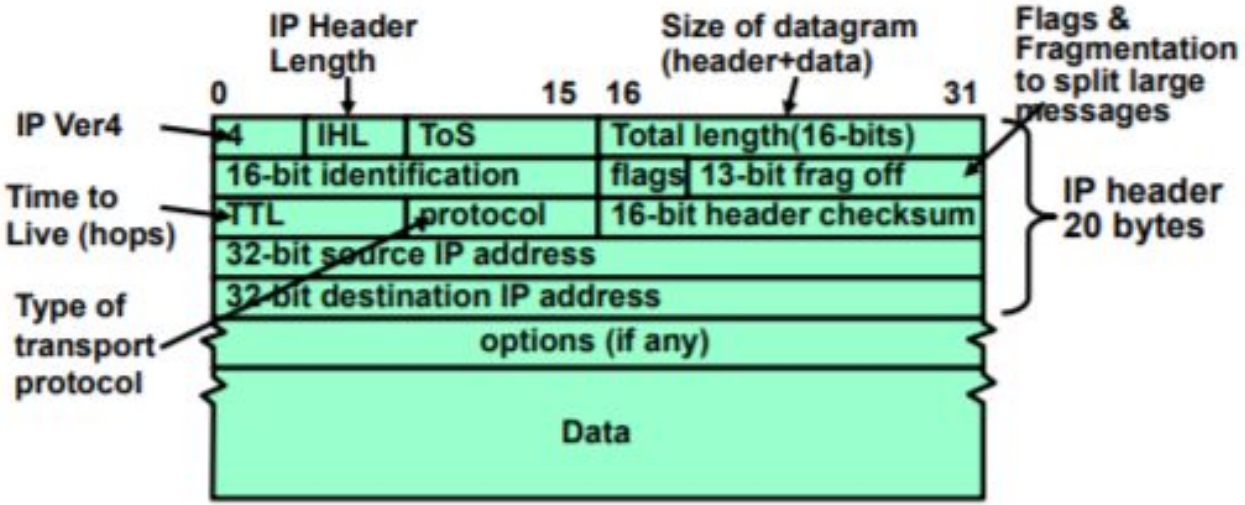
This is all very abstract, and in this class we only really care about the **network** and **transport** layers, so in this review session, we're going to adopt a simplistic view of the internet:

- The basic unit of communication on the internet is a network **packet**
- A packet consists of a chunk of data, some metadata, a source IP address, and a destination IP address (and is limited size)
- When computer **A** wants to communicate with computer **B**, it sends a bunch of network packets with computer **B**'s IP as destination. The packets then travel through a bunch of routers, eventually arriving at computer **B.**
    - Routers try to send the packet to a router closer to the destination IP.

The Internet

IPv4 Packet Format

- IP Packet Format:



The Internet

Problems:

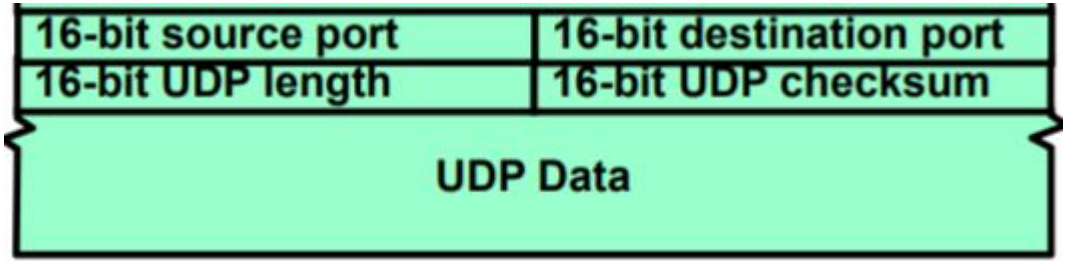
- Packets do not support ports.
- Packets are unreliable (may get dropped when jumping through routers, or get corrupted, or may be received in a different order than sent)

Solution is to build the transport layer on top of the packets:

- UDP: “best-effort delivery”
- TCP: reliable in-order delivery (but higher overhead than UDP)

UDP

In UDP, the **data** portion of a packet is formatted like:

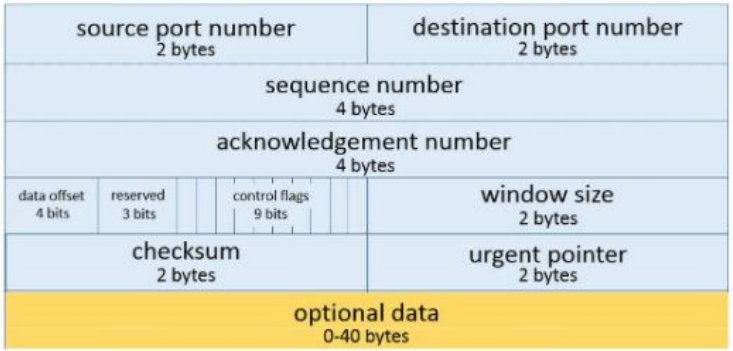


Notice we can now use ports.

However, this still doesn’t solve the problem of dropped/out-of-order packets

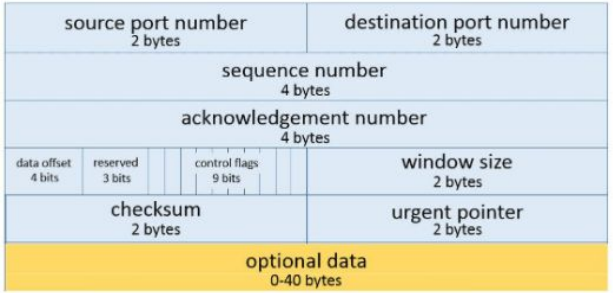
TCP

In TCP, the **data** portion of a packet is formatted like:



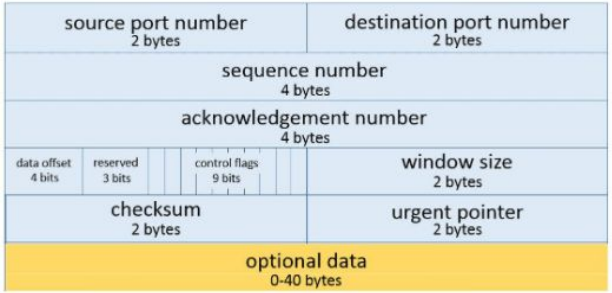
The additional fields allow us to do many things.

TCP

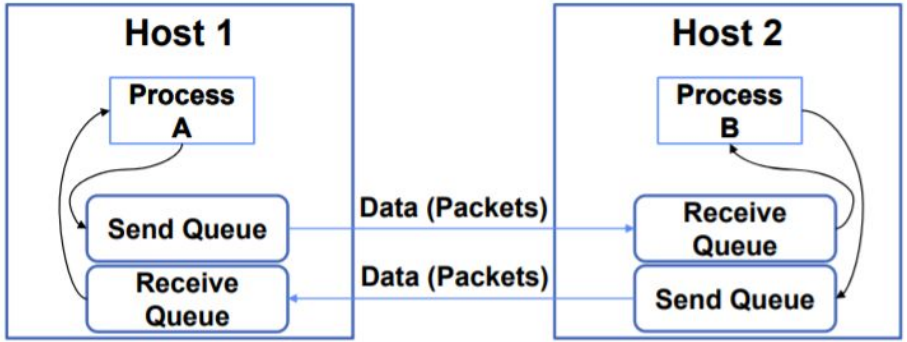


- Sequence number: tells you the original order the packets were sent (allows receiver to reconstruct this order)
  - Starting at a random number (determined by TCP handshake), index every byte of message
  - Sequence number is the index of the first data byte in the packet
- ACK number: after receiving some packets, receiver sends a packet back with ACK number being the next sequence number receiver is expecting
- Sender then uses the ACK number to determine which packets have been dropped, and resends these packets

TCP

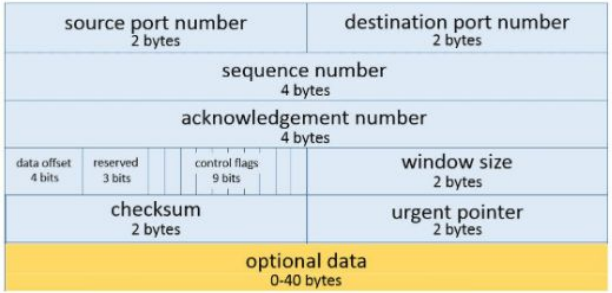


- What's window size?
- Let's say two computers are communicating with TCP:



- A host's window size for a TCP connection is how much remaining space it has in its receive queue
- TCP sender ensures it doesn't overwhelm the receiver by sending more bytes than the receiver's window size (unless resending old packets)

TCP



How do you determine how much data to send before waiting for ACKs?

Goal: want to avoid too much congestion.

Solution:

- Start small, increasing sending size slowly
- If timeout, cut sending size by half
- “Additive Increase, Multiplicative Decrease”

UDP and TCP

- What is provided by TCP that is NOT provided by UDP?
- When might a TCP sender send a packet that has more bytes than the receiver’s advertised window?
- Name one important feature of TCP/UDP that is NOT provided by plain IP.

UDP and TCP (soln.)

- What is provided by TCP that is NOT provided by UDP? Reliable inorder delivery and congestion avoidance.
- When might a TCP sender send a packet that has more bytes than the receiver's advertised window? When re-sending packets that were lost. The advertised window is for *new data*.
- Name one important feature of TCP/UDP that is NOT provided by plain IP. Multiplexing of different processes via port number.

Midterm Practice

Do take a look at the TCP window question on FA2020 M3 (if you have time).

Distributed File Systems

Distributed File Systems

- In distributed file systems, part of the file system is stored on a remote server.
- For instance, /home/oksi/162/ on your laptop actually refers to /users/oski on campus file server

Distributed File Systems

How do they work?

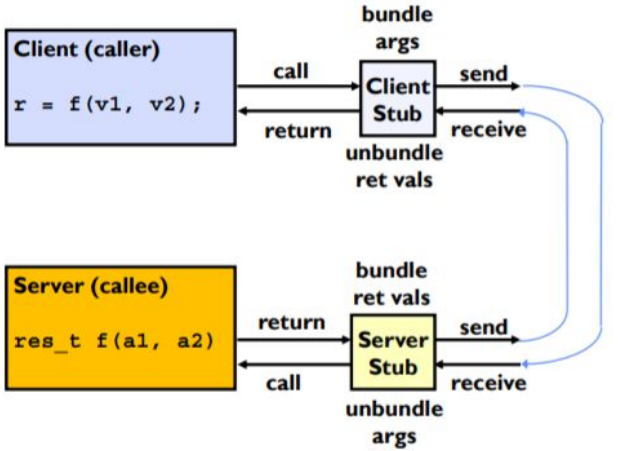
- So far, we've gone over:
  - File systems
  - Distributed systems + networks
- We're going to try to put all these together to build distributed file systems.
- Need one more concept: RPCs

RPC

- Basic idea: want to call a procedure on another machine (i.e. server)
- RPC (remote procedure calls) allow you to do this, and bundles all the technicalities (like transforming between big and little-endian, pointers, networking) behind a simple interface
- So for instance, if the client calls
  - **remoteFileSys->Read("lalala");**
- It gets translated automatically into call on server:
  - **fileSys->Read("lalala");**
- To do all this, RPCs have to marshal data into some canonical form

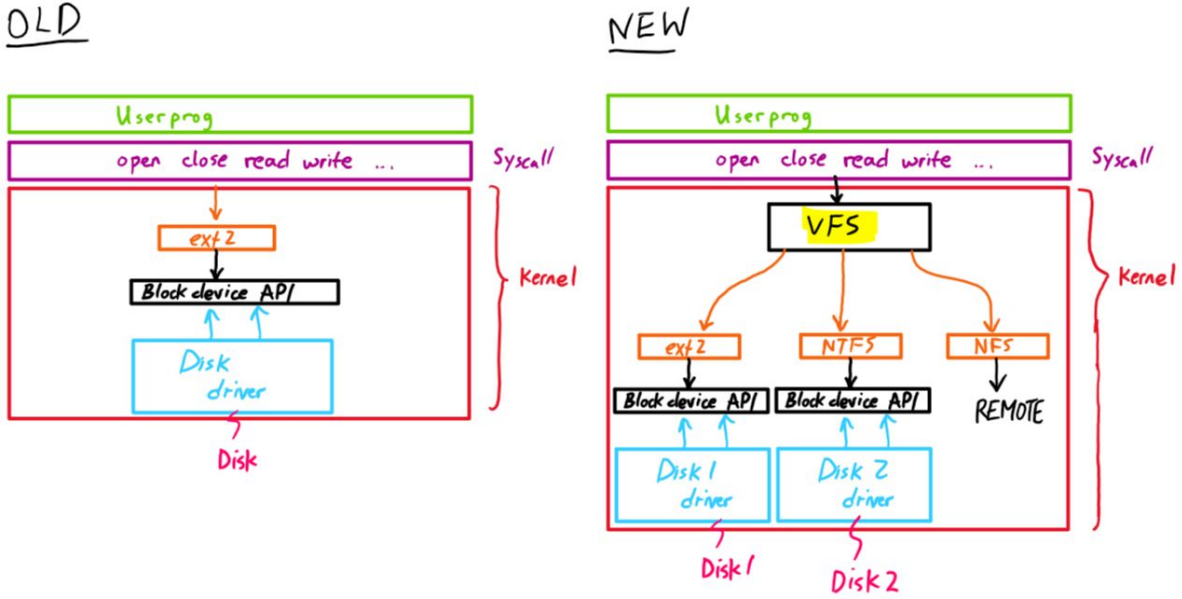
RPC

- Basic idea: want to call a procedure on another machine (i.e. server)
- RPC (remote procedure calls) allow you to do this, and bundles all the technicalities (like transforming between big and little-endian, pointers, networking) behind a simple interface



Building a Distributed FS

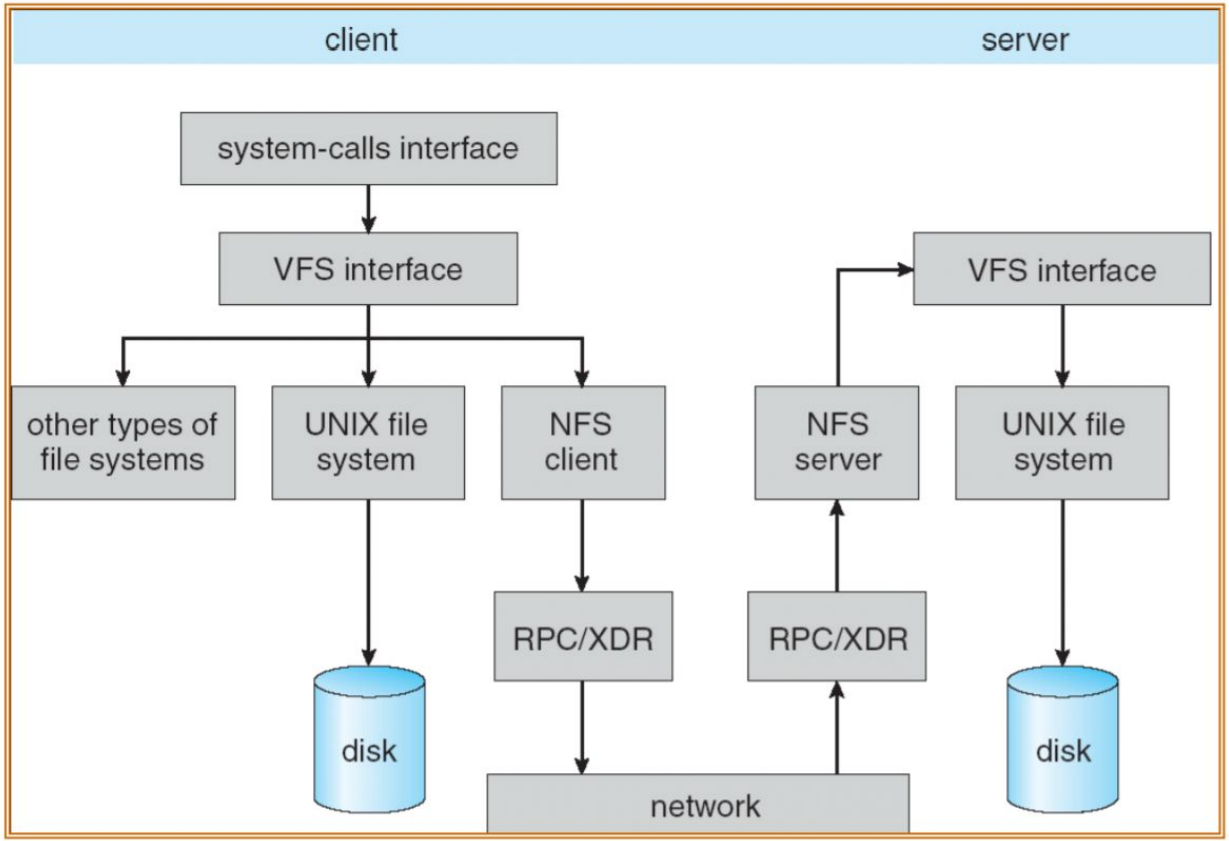
- Step 1: distinguish between remote files and local files.
- Solution: add a layer of abstraction with the **virtual file system (VFS)**.



Building a Distributed FS

- Step 2: handle remote files.
- We will be using **NFS**, the network file system.
- Basic idea: NFS translates read and write calls from VFS into RPCs (which are then executed on the remote file system).
- The RPCs are **stateless** (and **idempotent**)
  - e.g. reads include information for entire operation, such as **ReadAt(inumber, position)**, not **Read(openfile)**
  - no **open()** and **close()** RPCs
- Cache in local computer, poll server periodically to check for changes (server can be bottleneck).
- **Write-through caching**: modified data is committed to server's disk before results are returned to the client.
- If multiple clients write to the same file, results are arbitrary.

NFS


client
server
system-calls interface
VFS interface
other types of file systems
UNIX file system
NFS client
RPC/XDR
disk
network
VFS interface
NFS server
RPC/XDR
UNIX file system
disk

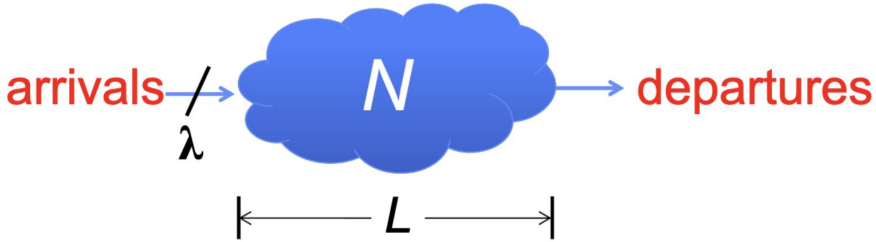
Queueing Theory

Terminology

Model arrivals to a queue probabilistically
Mean (m): Average value the distribution takes on.
Variance: (σ^2): Measure of average spread of values.
C: σ^2/m^2: Squared coefficient of variance:

Important values: C = 0: constant distribution (fully deterministic)
C = 1: Memoryless

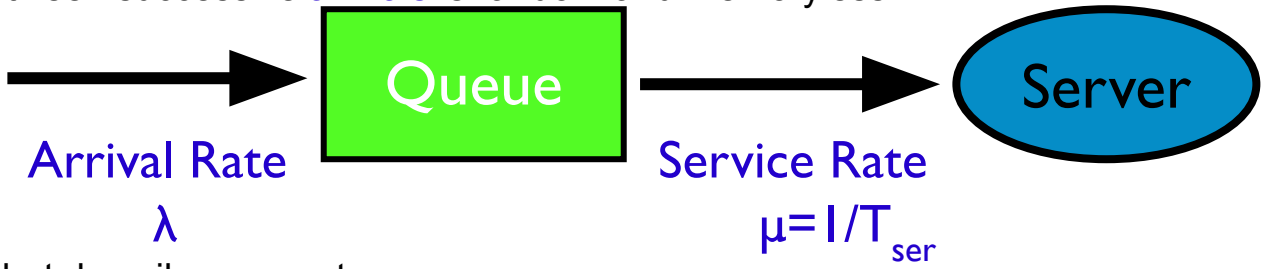
Little's Law



- In any *stable* system
  - Average arrival rate = Average departure rate
- The average number of tasks in the system (N) is equal to the λ arrival time times the response time (L)
  - Arrival time = throughput = λ
- N (jobs) = λ (jobs/s) x L (s)
- Regardless of structure, bursts of requests, variation in service
  - Instantaneous variations, but it washes out in the average

A Little Queuing Theory: Some Results

- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless



- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)

- Results (M: Poisson arrival process, 1 server):
  - Memoryless service time distribution (C = 1): Called an M/M/1 queue
    - » $T_q = T_{ser} \times u/(1 - u)$
  - General service time distribution (no restrictions): Called an M/G/1 queue
    - » $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u))$

Modeling Queues - M/M/1, M/G/1, M/M/m

- first letter refers to distribution of arrivals: “Memoryless” exponential distribution - rate at any time is constant
- second letter refers to service distribution: M: memoryless, G: general distribution w/ own mean, variance
- last letter - number of “servers”

Queuing Qn

- McDowell’s is a famous family restaurant in Queens, New York, known for its fine dining and rapid service. Let us say that a customer enters the drive-through every 5 seconds. From the moment the customer enters the drive-through, he will be able to leave the drive-through, with his food, in 60 seconds. How many people are waiting in the drive-through at any given time?

Queuing Qn

- McDowell's is a famous family restaurant in Queens, New York, known for its fine dining and rapid service. Let us say that a customer enters the drive-through every 5 seconds. From the moment the customer enters the drive-through, he will be able to leave the drive-through, with his food, in 60 seconds. How many people are waiting in the drive-through at any given time?

Use Little's Law.
Length of queue = Arrival rate (or Throughput) * Average Time Spent in Queue
= (1 person / 5 seconds) * 60 seconds # of people in the drive through
= 12 people in drive-thru on average.

This logically makes sense, because when a 13th person enters the queue, the first person would have been waiting for 60 seconds and departs. Hence, the length is on average 12 people (and it is always 12 people if each person arrives exactly 5 seconds apart and spends exactly 60 seconds in the queue)

Queuing Qn [cntd]

Utilization is defined as arrival rate/service rate. However, this means that utilization can easily exceed 1 if T_{ser} (or lambda) is large enough. Why do we limit our analysis of queues to utilization in the range [0, 1]?

u = utilization = λ / μ where λ is the arrival rate, μ (service rate) = $1 / T_{ser}$

Queuing Qn [cntd]

Utilization is defined as arrival rate/service rate. However, this means that utilization can easily exceed 1 if T_{ser} (or lambda) is large enough. Why do we limit our analysis of queues to utilization in the range [0, 1]?

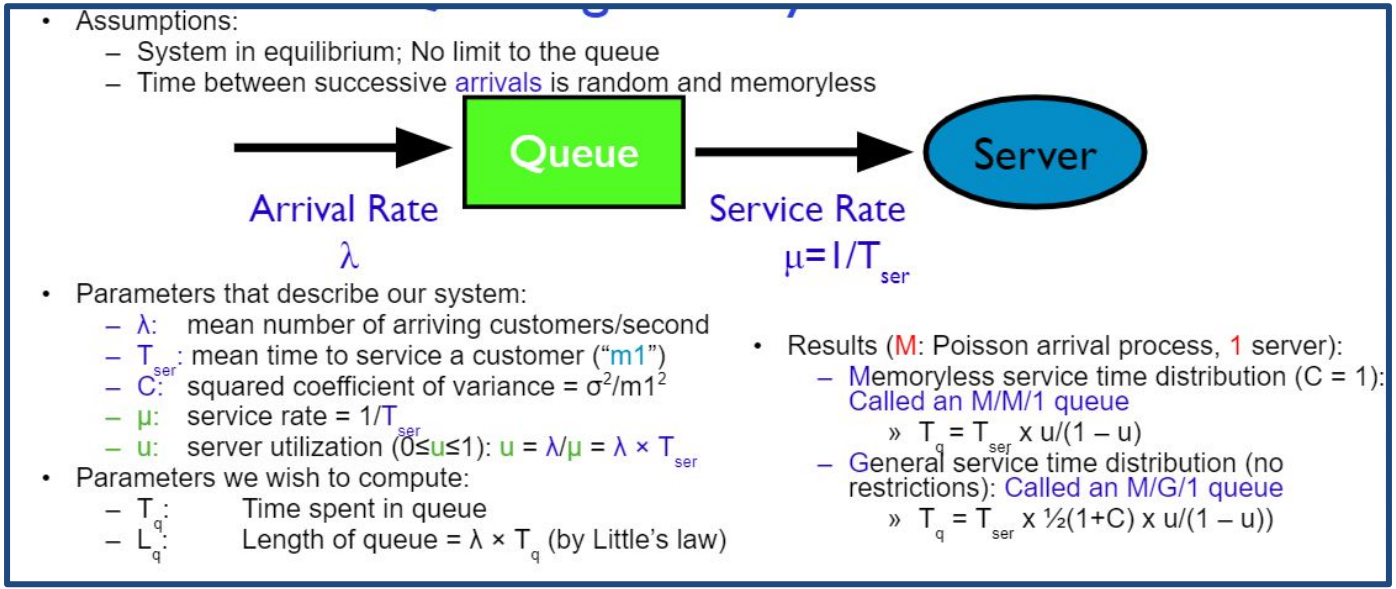
If the arrival rate exceeds the service rate, then on average, with each customer, the length of the queue will grow without bound. Thus we are dealing with an "infinite" queue, and it is pointless to examine the system. This is why utilization is restricted to values of [0, 1].

Remember the system has to be stable !

Queueing theory applies to a system where the long term, steady behavior approaches "average arrival rate = average departure rate"

Queueing Qn [cntd]

Consider the scenario where one customer walks into McDowell's per minute. On average, it takes 30 seconds to serve the customer at the front of the line. Calculate the average length of the line assuming that the system is memoryless.



Queuing Qn [cntd]

Consider the scenario where one customer walks into McDowell's per minute. On average, it takes 30 seconds to serve the customer at the front of the line. Calculate the average length of the line assuming that the system is memoryless.

$T_{Queue} = T_{Served} * (u/ 1 - u)$
$u = \text{arrival rate} * T_{Served}$

u = 1 customer / minute * 0.5 minutes = 0.5
T_{Queue} = 0.5 minutes * 0.5 / 0.5 = 0.5
Length of Queue = Arrival Rate (or Throughput) * Average time spent in queue (or response time)
Length of Queue = 1 person / minute * 0.5 minutes in queue = 0.5 persons.

This means on average, there are 0.5 persons in the queue. Keep in mind that this is an *average* (which is why the number is a fraction). It could be that for the most part, there are no people in the queue, but occasionally, it takes the cashier far longer than 30 seconds to serve the customer at the front of the line, creating a temporary backlog.

Good Luck!!