0. Introduction
- Done with OSes, networking
- Now: how to systematically deal with failures, or build "fault-tolerant" systems
  - We'll allow more complicated failures and also try to recover from failures
- Thinking about large, distributed systems. 100s, 1000s, even more machines, potentially located across the globe.
- Will also have to think about what these applications are doing, what they need

1. Building fault-tolerant systems
- General approach:
  1. Identify possible faults (software, hardware, design, operation, environment, ...)
  2. Detect and contain
  3. Handle the fault
     - do nothing, fail-fast (detect and report to next higher-level), fail-stop (detect and stop), mask, ...
- Caveats
  - Components are always unreliable. We aim to build a reliable system out of them, but our guarantees will be probabilistic
  - Reliability comes at a cost; always a tradeoff. Common tradeoff is reliability vs. simplicity.
  - All of this is tricky. It's easy to miss some possible faults in step 1, e.g. Hence, we iterate.
  - We'll have to rely on *some* code to work correctly. In practice, there is only a small portion of critical code
    We have stringent development processes for those components.

2. Quantifying reliability
- Goal: increase availability
- Metrics:
  - $\text{MTTF} = \text{mean time to failure}$
  - $\text{MTTR} = \text{mean time to repair}$
  - $\text{MTBF} = \text{mean time between failures} = \text{MTTF} + \text{MTTR}$
  - $\text{availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$
- Example: Suppose my OS crashes once every month, and takes 10 minutes to recover.
  - MTTF = 30 days = 720 hours = 43,200 minutes
  - MTTR = 10 minutes
  - availability = $\frac{43,200}{43,210} = .9997$
- This is one way to think about reliability! Different systems will get more specific
3. Reliability via Replication
   - To improve availability, add redundancy
   - One way to add redundancy: replication
     - Today: replication within a single machine to deal with disk failures
     - Tomorrow in recitation: replication across machines to deal with machine failures.

4. Dealing with disk failures
   - Why disks?
     - Starting from single machine because we want to improve reliability there first before we move to multiple machines
     - Disks in particular because if disk fails, your data is gone. Can replace other components like CPU easily. Cost of disk failure is high.
   - Are disk failures frequent?
     - Manufacturers claim high MTBF, but use AFR to give a more reasonable datapoint (see slides)
     - Even if probability of a specific disk failing is low, the probability of *some* disk failing in a big system is high

5. Whole-disk failures
   - General scenario: entire disk fails, all data on that disk is lost. What to do? RAID provides a suite of techniques.
     - RAID 1: Mirror data across 2 disks.
       - Pro: Can handle single-disk failures
       - Pro: Performance improvement on reads (issue two in parallel), not a terrible performance hit on writes (have to issue two writes, but you can issue them in parallel too)
       - Con: To mirror N disks' worth of data, you need 2N disks
     - RAID 4: With N disks, add an additional parity disk. Sector i on the parity disk is the XOR of all of the sector i's from the data disk.
       - Pro: Can handle single-disk failures (if one disk fails, xor the other disks to recover its data)
       - Can use same technique to recover from single-sector errors
       - Pro: To store N disks' worth of data we only need N+1 disks
       - Pro: Improved performance if you stripe files across the array. E.g., an N-sector-length file can be stored as one sector per disk. Reading the whole file means N parallel 1-sector reads instead of 1 long N-sector read.
       - This is really a result of having N disks of data, not of RAID specifically
       - RAID is a system for reliability, but we never forget about performance, and in fact performance influenced much of the design of RAID.
       - Con: Every write goes to the parity disk.
- RAID 5: Same as RAID 4, except intersperse the parity sectors amongst all N+1 disks to load balance writes. (see slide for diagram)
  - You need a way to figure out which disk holds the parity sector for sector i, but it's done systematically; controller can handle it
- RAID 5 used in practice, but falling out in favor of RAID 6, which uses the same techniques but provides protection against two disks failing at the same time.

6. Your future
- RAID doesn't solve everything.
  - E.g., what about failures that aren't independent?
- We are going to work to develop abstractions that allow us to be more systematic about handling failures.
- Lectures will tackle these abstractions on larger and larger systems.