Previously
- Enforced modularity on a single machine via virtualization
  - Virtual memory, bounded buffers, threads
- Saw monolithic vs. microkernels
  - Talked about VMs as a means to run multiple instances of an OS on a single machine with enforced modularity (bug in one OS won't crash the others)
  - Big thing to solve was how to implement the VMM. Solution: trap and emulate. How the emulation works depends on the situation.
    - Another key problem: how to trap instructions that don't generate interrupts.

1. What's left? Performance
- Performance requirements significantly influence a system's design
  - Today: general techniques for improving performance

2. Technique 1: buy new hardware
- But:
  - Not all aspects improve at the same pace
  - Hardware improvements don't always keep pace with load increases
- Conclusion: need to design for performance, potentially re-design as load increases

3. General approach
- Measure the system and find the bottleneck (the portion that limits performance)
- Relax (improve) the bottleneck

4. Measurement
- To measure, need metrics:
  - Throughput: number of requests over a unit of time
  - Latency: amount of time for a single request
  - As system becomes heavily-loaded:
    - Latency and throughput start low. Throughput increases as users enter, latency stays flat...
    - ...until system is at maximum throughput. Then throughput plateaus, latency increases
  - For heavily-loaded systems: focus on improving throughput
- Need to compare measured throughput to possible throughput: utilization
- Helpful to have a model in place: what do we expect from each component?

4. How to relax the bottleneck
- Better algorithms, etc. These are application-specific. 6.033 focuses on generally-applicable techniques
5. Disks
- HDDs (magnetic disks):
  - Several platters on a rotating axle
  - Platters have circular tracks on either side, divided into sectors.
  - Disk arm has one head for each surface, all move together
  - Each disk head reads/writes sectors as they rotate past. Size of a sector = unit of read/write operation (typically 512B)
- To read/write:
  - Seek arm to desired track
  - Wait for platter to rotate the desired sector under the head
  - Read/write as the platter rotates
- SSDs (solid state drives):
  - Organized into cells, each of which hold one (or 2, or 3) bits
  - Cells organized into pages; pages into blocks
  - Reads happen at page-level. Writes also at page-level, but to new pages (no overwrites of pages)
  - Erases (and thus overwrites) are at block-level
  - Takes a high voltage to erase
- How long does R/W take on HDD?
  - Seek time: Avg read seek 8.2ms, avg write seek 9.2ms
  - Given as part of disk specs
  - Rotation time: 0–8.3ms
    - Platters only rotate in one direction
  - R/W as platter rotates: 35–62MB/sec
    - Also given in disk specs (not on slides)
  - So reading random 4KB block: 8.2ms + 4.1ms + ~.1ms = 12.4
  - \[ \frac{4096 \text{ B}}{12.4 \text{ ms}} = 322 \text{KB/s} \]
  => 99% of the time is spent moving the disk
- Can we do better?
  - Batch individual transfers?
    - .8ms to seek to next track + 8.3ms to read entire track = 9.1ms
      - .8ms is single-track seek time for our disk (again, from specs)
    - 1 track contains \(~1000\)sectors * 512B = 512KB
      - throughput: \( \frac{512\text{KB}}{9.1\text{ms}} = 55\text{MB/s} \)
  - Lesson: avoid random access. Try to do long sequential reads.
  - But how?
    - If your system reads/writes entire big files, lay them out contiguously on disk. Hard to achieve in practice!
    - If your system reads lots of small pieces of data, group them
  - SSDs help with this too (no moving parts => no seeking), but still benefit from batching writes

6. OS Abstractions
- Filesystem abstracts a lot of this disk nonsense away, which is nice a lot of the time. Do filesystems ever get in the way?
- Consider a database
  - Series of tables of data; complex relationships between data
  - Database Management System (DBMS) has a good understanding of incoming queries and stored data
- If the database runs on top of a filesystem, how is it stored? Does it matter?
  - OSes often do per-file locks. All data in one file = no concurrency among database users. One file per cell = an enormous number of locks for each user to acquire to do a simple query.
  - There are other options (per-table, per-column,…), but it's not always a natural fit
- Additionally, how would caching work?
  - LRU policy works great in many cases, but not usually for database workloads
  - Moral of story: DBMS is *great* at predicting which byte it needs next; much better than OS. Filesystem/normal OS operations can get in the way
  - DBMSes can sometimes benefit from "block-level control", where they have direct access to blocks and don't need to go through a filesystem

7. Summary
- Performance matters in all parts of systems
- We look for generally-applicable performance-improving techniques (caching, batching, etc.)
- Sometimes abstractions/functionality that is useful for parts of our system can get in the way of other parts