0. 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
- Why? Moore's law => processing power doubles every 1.5 years, DRAM density increase over time, disk price (per GB) decreases, ...
  - But:
    - Not all aspects improve at the same pace
    - Moore's Law is plateauing
    - 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
- Relationship between these changes depending on the context
  - 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
  - Utilization sometimes makes bottleneck obvious (CPU is 100%
utilized vs. disk is 20% utilized), sometimes not (CPU and disk are 50% utilized, and at alternating times)
- Helpful to have a model in place: what do we expect from each component?
- When bottleneck is not obvious, use measurements to locate candidates for bottlenecks, fix them, see what happens (iterate)

4. How to relax the bottleneck
- Better algorithms, etc. These are application-specific. 6.033 focuses on generally-applicable techniques
- Batching, caching, concurrency, scheduling
- Examples of these techniques follow. The examples related to operating systems (that's what you know), but techniques apply to all systems

5. Disk throughput
- (HDDs, not SDDs -- those are coming later)
- How does a disk work?
  - Several platters on a rotating axle
  - Platters have circular tracks on either side, divided into sectors.
    - Cylinder: group of aligned tracks
  - 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
- How long does R/W take?
- Example disk specs:
  - Capacity: 400GB
  - # platters: 5
  - # heads: 10
  - # sectors per track: 567-1170 (inner to outer)
  - # bytes per sector: 512
  - Rotational speed: 7200 RPM => 8.3ms per revolution
  - 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
- So reading random 4KB block: 8.2ms + 4.1ms + ~.1ms = 12.4
  - 4096 B / 12.4 ms = 322KB/s
- => 99% of the time is spent moving the disk
- Can we do better?
  - Use flash? We'll get to that
  - 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: 512KB/9.1ms = 55MB/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

6. Caching
- Already saw in DNS. Common performance-enhancement for systems
  - How do we measure how well it works?
    - Average access time: hit_time * hit_rate + miss_time * miss_rate
  - Want high hit rate. How do we know what to put in the cache?
    - Can't keep everything
    - So really: how do we know what to *evict* from the cache?
  - Popular eviction policy: least-recently used
    - Evict data that was used the least recently
    - Works well for popular data
    - Bad for sequential access (think: sequentially accessing a dataset that is larger than the cache)
  - Caching is good when
    - All data fits in the cache
    - There is locality, temporal or spatial
  - Caching is bad for
    - Writes (writes have to go to cache and disk; cache needs to be consistent, but disk is non-volatile)
  - Moral: to build a good cache, need to understand access patterns
    - Like disk performance: to relax disk as bottleneck, needed to understand details of how it works

7. Concurrency/scheduling
- Suppose server alternates between CPU and disk:
  - CPU: --A-- --B-- --C--
  - Disk: --A-- --B-- --C--
- Apply concurrency, can get:
  - CPU: --A----B----C-- ...
  - Disk: --A----B-- ..
- This is a scheduling problem: different orders of execution can lead to different performance
- Example:
  - 5 concurrent threads issue concurrent reads to sectors 71, 10, 92, 45, and 29.
  - Naive algorithm: seek to each sector in turn
  - Better algorithm: sort by track and perform reads in order.
    - Gets even higher throughput as load increases
    - Drawback: it's unfair
- No one right answer to scheduling. Tradeoff between performance and fairness.

8. Parallelism
- Goal: have multiple disks, want to access them in parallel
- Problem: how do we divide data across the disks?
- Depends on bottleneck
  - Case 1: many requests for many small files. Limited by disk seeks. Put each file on a single disk, and allow multiple disks to seek multiple records in parallel
  - Case 2: few large reads. Limited by sequential throughput. Stripe files across disks.
- Another case: parallelism across many computers
  - Problem: how do we deal with machine failures?
  - (One) Solution: go to recitation tomorrow!

9. Alternative Technologies
- Talked a lot about HDDs. Why not use solid-state disks (SSDs)?
- SSD: (typically) flash memory that exports a disk interface
- Flash memory = no moving parts
- Is it better? Some specs:
  - Sequential read: 400 MB/sec.
  - Sequential write: 200-300 MB/sec.
  - Random 4K reads: 5700/sec (23MB/s)
  - Random 4K writes: 2200/sec (9MB/s)
- Conclusions:
  - Sequential access still much faster than random access.
  - Write performance is noticeably worse
    - Flash can only erase large units at a time. writing a small block = read large block, modify it, write it back
    - Modern SSDs have complex controllers that try to optimize this
  - SSDs are also more expensive
  - Many performance issues are the same
    - HDDs and SSDs are slower than RAM
  - Can still avoid small writes with batching
  - Lesson: even as technology improves, our performance techniques still apply. Understanding the details of your system (e.g., how the storage media works) is crucial.

10. Summary
- We can't magically apply any of the previous techniques. Have to understand what goes on underneath.
  - Batching: how disk access works
  - Caching: what is the access pattern
  - Scheduling/concurrency: how disk access works, how system is being used (the workload)
  - Parallelism: what is the workload

11. Useful numbers for your day-to-day-lives:
- Latency:
- 0.00000001ms: instruction time (1 ns)
- 0.0001ms: DRAM load (100 ns)
- 0.1ms: LAN network
- 10ms: random disk I/O
- 25-50ms: Internet east -> west coast

Throughput:
- 10,000 MB/s: DRAM
- 1,000 MB/s: LAN (or 100 MB/s)
- 100 MB/s: sequential disk (or 500 MB/s)
- 1 MB/s: random disk I/O