Performance: why am I lecturing about it?
often has huge effect on design
often forces major re-design as loads grow
faster CPUs haven't "solved" performance
problems have grown too, e.g. Internet-scale
disk and net latency hasn't improved

Performance Introduction
your system is too slow
perhaps too many users, and they are complaining
what can you do?
1. measure to find bottleneck
could be any of the components incl client
you hope to find a dominating bottleneck!
2. relax bottleneck
increase efficiency or add hardware

Decide what you mean by "performance"
throughput: requests/second (for many users)
latency: time for a single request
sometimes inverses:
if it takes 0.1 second of CPU, and one CPU, then throughput = 10/sec
often not inverses
w/ 2 CPUs, latency still 0.1, but throughput 20/sec
queuing and pipelining

I will focus on throughput, appropriate for heavily loaded systems
most systems gets slow as # of users goes up
at first, each new user uses some otherwise idle resources
then they start to queue
[graph: # users, reply/sec, linear up, hits a plateau]
[graph: # users, delay, stays at zero then linear up (queuing delay)]

How to find bottleneck?
1. measure, perhaps find that e.g. cpu is 100% used
   but maybe not, e.g. cpu 50% use and disk 50% used, just at different
times
2. model
   net should take 10ms, 50ms CPU, 10ms disk
3. profile
   tells you where CPU time goes
4. guess
   you will probably have to do this anyway
   test hypothesis by fixing slowest part of system
   can be difficult:
   if disk busy all the time, should you buy a faster disk / two disks?
   or more RAM?

You may need to make application logic more efficient
fewer features, better algorithms
I cannot help you with that -- application-specific
but there are general-purpose techniques
The main performance techniques
1. caching
2. I/O concurrency
3. scheduling
4. parallel hardware (two disks, two CPUs, &c)
   these are most useful when many waiting requests
   but that will often be the case if your server is heavily loaded

I'm going to focus on disk bottlenecks
Every year it gets harder to be CPU-bound
what tricks for good disk performance?

hitachi 7K400: 400 GB. 7200. 8.5ms r seek, 9.2ms w.
567 to 1170 s/t. 10 heads. abt 87000 cylinders.

primer on disks
[disk slide]
physical arrangement
   rotating platter: 7,200 RPM, 120/sec, 8.3 ms / rotation
   continuous rotation
   circular tracks, 512-byte sectors, about 1000 sectors/track
   perhaps 100,000 tracks, concentric rings
   multiple platters, 5 for 7K400, so 10 surfaces
   cylinder: set of vertically aligned tracks
   one disk arm, head per surface, they all move together
can only read from one head at a time
three movements required:
"seek" arm to desired track: varies w/ # tracks, 1 to 15 ms
wait for disk to "rotate" desired sector under head: 0 to 8.3ms
read bits as they rotate under head at 7200

disk performance?
big multi-track sequential transfers:
one track per rotation
512*1000 / 0.0083
62 megabytes/second
that is fast! unlikely to be a bottleneck for most systems
small transfers, e.g. 5000-byte web page:
from *random* disk location
seek + rotate + transfer
avg seek: 9 ms
avg rotate: 1/2 full rotation for random block
transfer: size / 62 MB/sec
9 + 4 + 0.1 = 13.1ms
rate = 5000 / 0.0131 = 0.4 MB/sec
i.e. 1% of big sequential rate. this is slow.
sadly this is typical of real life

Lesson: lay your data out sequentially on disk!

caching disk blocks
use some of RAM to remember recently read disk blocks
   this is often the main purpose of RAM...
your o/s kernel does this for you
table:
   | BN   | DATA   |


read(bn):
    if block in cache, use it
    else:
        evict some block
        read bn from disk
        put bn, block into cache
hit cost: about 0 s to serve a small file from RAM
miss cost: 0.010 to read a small file from disk

eviction / replacement policies
important: don't want to evict something that's about to be used!
least-recently-used (LRU) usually works well
    if it's been used recently, will be used again soon
LRU bad for huge sequential data that doesn't fit
    if you read it over and over (or even only once)
    if it's been used recently, won't be used again for a while!
don't want to evict other useful stuff from cache
random? MRU?

how to decide if caching is likely to work?
productive to think about working set size vs cache size
you have 1 GB of data on disk and 0.1 GB of RAM
will that work out well?
maybe yes:
    if small subset used a lot (popular files)
    if users focus on just some at a time (only actively logged in users)
    if "hit" time << "miss" e.g. disk cache
    if disk I/O signific fraction of overall time
maybe no:
    if data used only once
    if more than 0.1 GB read before re-use (i.e. people read whole GB)
    if people read random blocks, then only 10% hit probability
    if hit time not much less than miss time
    e.g. caching results of computation

i/o concurrency
what if most requests hit but some have to go to disk?
    and you have lots of requests outstanding
    [time diagram: short short long short short]
we don't want to hold up everything waiting for disk
idea: process multiple requests concurrently
    some can be waiting for disk, others are quicker from cache
    can use threads
note: threads handy here even if only one CPU
special case: prefetch (works w/ only one thread)
special case: write-behind

scheduling
sometimes changing order of execution can increase efficiency
if you have lots of small disk requests waiting
sort by track
results: short seeks (1ms instead of 8ms)
the bigger the backlog, the smaller the average seek
system gets more efficient as load goes up!
higher throughput
"elevator algorithm"
maybe you can sort rotationally also, but that's hard
cons: unfair -- increases delay for some

if you cannot improve efficiency
buy faster CPU/net/disk if you can
otherwise buy multiple servers (CPUs + disks) -- but how to use?
strict partition of work: easiest
users a-m on server 1 (cpu+disk)
users n-z on server 2 (cpu+disk)
no interaction, so e.g. no need for locks, no races
hard if e.g. some operations involve multiple users -- bank xfers
perhaps factor out computing from storage
front end / back end
front end can get data for all needed users from back end
works well if CPU-bound or PEs can cache

Pragmatics
  programmer time costs a lot
  hardware is cheap
  programming most worthwhile if allows you to use more h/w

****

show slide
going to look at a quiz question from a few years ago
as practice, and to illustrate performance ideas
quizzes often present some new pretend system, ask lots of
  questions about it and effects of changes
worth practicing a bit (old exams), takes some getting used to

OutOfMoney
serving movie files, to clients, over net
single CPU, single disk
each file is 1 GB, split into 8 KB blocks, randomly scattered

Q1: 1153 seconds
seek + half rotation + (8192 / 10 MB/sec)
0.005 + 0.003 + 0.0008 = 0.0088 sec/block
times 131072 = 1153

if layout were better, how long would it take?
mark adds a one-GB whole-file cache

Q2: 1153 seconds
does that mean the caching scheme is bad?

Q3: B
what behavior would we expect to see for each of those reasons if it were true?
in what circumstance would that caching scheme work well?
how could it be improved?

Threads: what is the point? do we expect threads to help?

Why do we think it is telling us where it calls yield? What is the significance of those points? (I/O concurrency)

Why might non-pre-emptive be important?

new caching code:
  1. 4 GB
  2. reads each block independently: *block* cache, not whole-file

Why might per-block caching be important? Probably second-order, to fix a bug in the question: that otherwise GET_FILE_FROM_DISK might prevent all other activity.

Q4: 100% hit rate.

Maybe an artifact of having sent a first non-concurrent request. What if he had started by sending many requests in parallel?

Q5: Ben is right. One CPU, threads are non-pre-emptive.

What if two CPUs, or pre-emption? Would this lock be enough?

Might need to protect all uses of cache, disk driver, network s/w, maybe best done inside each of these modules.

Might want to do something to prevent simultaneous disk read of same block, i.e. make IF, GET, and ADD indivisible. Though a spin-lock is probably not the right answer. Per-block busy flag, wait(), notify().

Q6: 0.9 * 0 + 0.1 * 1153 = 115.3

Q7: E

The cache is bigger than the total data in use, so no replacement is ever needed.