Systems Optimization

- Understanding Compiler optimization blockers
- Code Motion
- Profiling Tools
- Library Performance Debugging

Feedback tool

-Logging in 1000 times to feedback tool will break it

Grading for Pset4

1. Does it produce correct answers
2. Does it beat stdio on sequential
3. Does it beat stdio on stride (sometimes)
4. Style

Code will also be tested on a huge file that is too large to fit in memory

To generate a huge file for testing purposes:

yes y | head -c $((1 << 30))

1 GB worth of characters, filled with y on every line

To enlarge this file: yes y | head -c $((1 << 30)) > f

cat f f > g

cat g g > h

Goal: Make programs faster

Tools to speed up programs:

- Improve locality: is friendly to rest of the system
- Reduce the number of system calls and other expensive operations
- Algorithm with better computational complexity
- Better hardware (may seem weird but in real life $$= faster)
- *Introduced later: Avoided optimization blocker - see Aliasing

Warning: implementing all of these will probably make your program break because you make a stupid mistake.

Quotes about optimization: “The best speedup is a working program.”

“Premature optimization is the root of all evil.”

When is optimization premature? When you don’t know how fast your code is, when you don’t know where your speed problems are, when you’re trying to speed up things that haven’t been tested

Optimization:

- Find your slow points, then optimize the bottlenecks
- Use program "time" to find how long a program takes to run
  - time ./strtolower: eturns the amount of time the program took to run
- After speed is calculated, you want to compare it to something, a "performance baseline" to measure your speed by.
- In this example, use a "strtolower" function that doesn't do anything (literally the body is commented out) to isolate program, also measure with different number of arguments (first 100mil characters, etc)

Problem with our strtolower: strlen is being calculated every iteration of a loop, so while strtolower can run in O(n), ours is calculating strlen, which is O(n), every time, so our code is running in O(n^2)
Solution: to create a variable len = strlen(s), use "len" in loop rather than strlen

This is **"code motion"** (moving statements or expressions within the program to optimize)
We made an algorithmic improvement: going from O(n^2) to O(n) - these programs do the same thing but have different complexities. Sometimes, compilers will do this optimization for us, but ours did not because we were working with something that could be changing in our loop, so our compiler couldn't predict if it could optimize.

Aliasing:

- Before, we were working with code that might have modified s (indeed we were changing the case), so our compiler couldn't optimize or else it might have messed up our code. (even though we know that strlen is going to be different for a string and a lowercase string)
- However, when we work with code that doesn't modify s, like "++nchars" that just calculates the length of the string, the compiler can be absolutely certain it won't be messing up the result

"As-If Rule": The optimizer is allowed to transform code as long as the code still behaves as if the original code did.
Since the body of our loop modified s by making the characters in s lower-case, our optimizer could not prove that moving code out of the loop would run the program the same, which is why it could not transform the call to strlen to be outside of the loop.

"Aliasing" because the value in s[i] might point to the same place in memory (be an alias) for s, therefore modifying s inside the loop

When program is recompiled (after we modify the inside of the loop to do no useful work), the compiler is able to call strlen 0 times
If we call strlen something like "strlen2," our compiler still optimizes (?)

Servicetranslate

Service translate: supposed to take in a bunch of strings that correspond to ports, outputs the specific port number corresponding to input port name. Ex. www -> 80, https -> 443, foo -> 0, blah -> 0, ftp -> 21

Port Numbers (used to differentiate different network connections): every machine has a database that maps port names to port numbers

Now we are looking at the speed of this program (first try 1.02 seconds)

Looking for code motion opportunities to speed up the program (we are counting the number of lines in the file very inefficiently)
Just making a new variable to hold the line count variable drops the time to .4 seconds

However, .4 is still slow. We need new tools to figure out how to measure performance and find ways of making this better.

Performance Measurement Tools

gprof: Sampling profiler
    Takes timed, random samples of program state and then reports them
perf: Whole system profiler
    Samples everything running on the machine, including the kernel
gdb: Sampling profiler (human-scale)
    Pressing ctrl-c a couple of times

gprf

1. Type "-prof" at the end of the program run command (servicetranslate-prof)
2. Run gprof on the gmon file that is created
3. This output shows the percentage of time used in each functions (by sampling). In our very inefficient program which calls "count_lines" every iteration of the loop, 100% of our time is being used with count_lines, and it's called 4200450 times.
4. gprof does not include library functions and system calls in its results.

perf is better.

perf

1. Record phase
    sudo perf record [command line]
2. Report phase
    sudo perf report
3. Reports most expensive instructions, whether in the kernel or somewhere else, which can catch bad cache designs as well

4. Use strace to find out what system calls a program makes
5. Can see that we're opening the etc/services file 2049 times by using the basic system call (getservbyname). This is extremely inefficient- every time we check a line to see if it is a port, we must reopen and read the file

Solution: **caching**
Read in the file once and save it somewhere

scache
Array in memory with each entry containing s.name, s.port, s.proto
Look at every entry in the cache, if the names match and the protocols match, then return that, otherwise return null pointer

This now gives a factor of 20 increase in speed (.02s)
We can now run the program on the big file which it does in 2.84s

In our etc/services list, we see only 2 protocols for every service: tcp and udp
Looking up tcp and udp protocols into the cache is slow because every single one requires a separate lookup
Solution: keep exactly 1 version of "tcp" in memory, so instead of using strcmp to compare strings which requires reading in memory, can just compare pointers , saves roughly .4 seconds