CS61
Lecture 12
Thursday, October 10, 2013

Problem set due tomorrow

Midterm next Thursday (Oct 17) in class during class, practice out today - open book/note/computer, no internet (no stackoverflow)

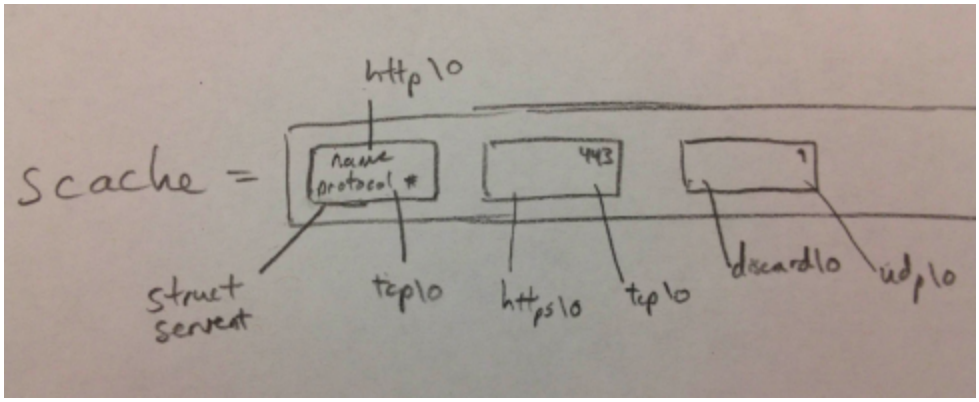
- nobody should access resources outside the cs61.seas.harvard.edu site
- go over lecture code, understand better to help review
- cannot run “experiments” during the test

Review here next Tuesday night (Oct 15) 7:30-9pm same room

Today: finish **Caches**, begin next unit **Instructions**

---

service translate ...



scache is an array of servent structs

Sequential arrangement good for processor cache because of locality - one item probably means we want the next one

Cache line = 64, 4 items x 16 byte fit (4 of them fit into 1 cacheline)

Separate pointers inside each struct will take up 1 cacheline each (even though they don't fill the entire 64 bytes)

Demo: l12/servicetranslate-03.c

Function my_getservbyname creates cache and looks through for what we want (number)

Use 75-80% of cache line

Strings allocated but not next to each other, handled by memory allocator, so low utilization (random accesses have bad utilization)

These cachelines are probably not used more than once, because we only need to use that 4-byte string

Every time we look up a new name/protocol, a new cacheline will have to be created because it was not close to any of the previous cachelines in memory

The chance that there is good locality between these randomly accesses in memory is not clear

Improvements:

Protocols usually udp or tcp, so optimization for common case

First goal: use less memory, so access pattern on protocol names will have more locality certain protocol names share same spot in memory, and thus speed because that address is probably already in the cache

Demo: servicetranslate-04.c, canonicalize_proto is this idea, stores every distinct protocol name once in a cache. One array for strings, growable buffer that doubles every time we run out of space

Canonicalization gets same benefit without changing format of this object so older code can work

Demo time ./servicetranslate-03 < input.txt > /dev/null
vs 04

.6 vs .5 seconds: 04 is slightly faster since we access less memory

Furthermore, once you canonicalize, you can compare pointers. If the strings are the same then the pointers will be the same because they point to the same string (treating pointer comparison as data comparison)

In servicetranslate-04, canonicalize our input proto to get a pointer that you can just

compare to the one in the cache entry - doing so leads to a slight increase in speed

```
my_getservbyname(const char* name, const char* proto) {
      if (!scache)
            populate_services_cache();
      proto = canonicalize_proto(proto);
      for (size_t i = 0; i != scache_size; ++i)
            if (proto == scache[i].entry.s_proto
                && strcmp(name, scache[i].entry.s_name) == 0)
                  return &scache[i].entry;
      return NULL;
}
```

The proto compare should be moved to the front so we only have to do that fast comparison instead of evaluating strcmp first (slower) - much faster since we're only accessing half the memory as before (< 0.4 s)

Can further optimize service names (arranged poorly in cache) - let's load one array instead of two

```
typedef struct scache_slot {
      struct servent entry;
            uint32_t namehint;
} scache_slot;

uint32_t getnamehint(const char* name) {
    // initialize to avoid random data from before
    uint32_t h = 0; // one idea to use a hash function, but we can just use
first chars
    strncpy((char *) &h, name, sizeof(h));
    return h;
}
```

Servent structure limited, but scache has slots that we can add stuff to - one idea: space for name (maybe 8 bytes), small ones could fit, then only allocate more memory for larger ones. Prof Kohler wants to use 4 bytes as a "namehint", so early characters that conflict will stop checking faster (do not need to do the strcmp), and if they are the same, then we go into memory and check. Conservative change helps performance, but not certain (unlike canonicalization)

We copy the first 4 bytes of name into a 4 byte integer "namehint" in our cache struct

```
my_getservbyname(const char* name, const char* proto) {
```

```
        if (!scache)
                populate_services_cache();
        proto = canonicalize_proto(proto);
        uint32_t h = getnamehint(name);
        for (size_t i = 0; i != scache_size; ++i)
                if (proto == scache[i].entry.s_proto
                    && h == scache[i].namehint
                    && strcmp(name, scache[i].entry.s_name) == 0)
                  return &scache[i].entry;
        return NULL;
}
```

In my_getservbyname getnamehint(name) and then compare before you strcmp, protocol and namehint comparisons both 4 bytes vs 4 bytes, in the same cache, so order there doesn't matter as much - speed still around 0.4s … Prof Kohler wants to use bigger file to see time difference

Faster: don't store cache in random order, but rather sort and binary search - advantage of finding a specific protocol faster in O(log n)

st_10 uses dumb cache implementation but faster algorithm (binary search), st_11 has namehint as a cache-aware data structure to improve performance instead, but st_11 still faster

DONE WITH **CACHE**, `break;` then **INSTRUCTIONS**

IMPORTANT: code for problem set 2 runs faster on your computer than server - believe grading server, your computer has virtual machine, and server is more real and listens to all their instructions. But ratios should be the same

GOAL: reverse engineer a program from x86 assembly

Prof Kohler - shows random code (f00.s) and stares at it in silence until we tell him what it does … is x86 assembly instructions - output of a compiler from a c file - what does this c file look like?

Declares function f, lines with a . and start in the middle are metadata about instructions, # are comments, lines that don't have . are actual instructions, something with a colon defines addresses

To test, compile a file with what you think it had, use clang -S or gcc -S and see if it matches
    -S gets assembly output instead of binary output

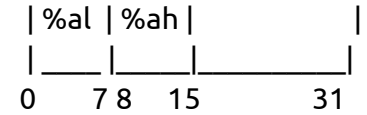
C abstract machine doesn't know about registers, but f01.s is a program that puts 0 into a register and then returns

All registers start with % signs in x86

%eax : accumulator

Historical aside about first x86 machines … had %ax (16 bits) with parts %al (low 8 bits) and %ah (high 8 bits) … re-extended and used e as prefix

%eax box:




```
movl $0, %eax
ret
```

$ means immediate (integer 0), % means it's an address or register
source is on the left, destination is on the right (putting value 0 into %eax)

Tried int a = 0; (void) a; but compiler threw it away since we didn't use it, and not what we wanted

Can't directly assign register … but return 0; does what we want since we have to move 0 into %eax to return it

Processors don't understand functions, but only operations on numbers. Compiler enforces conventions to translate from abstract machine into processor (what happens to return values? leave them in %eax)

Return address of location of instructions to go is still in stack, but actual value in register

```
movl b, %eax
addl a, %eax
ret
```

f02.s : returns a + b (addition is commutative … for computer integers, so order doesn't matter). We know a and b are globals and not arguments because there are no percent signs, so they are globals

f03.s - f05.s : same assembly but different source code (back to first lecture w/ different

types) adding unsigned or elements of arrays or pointers

comments have to do with compilers breaking control flow into units that are sequential

f06.s : only uses lower 32 bits of a long long because it knows we never touch the other 32 bits

“How? No one line program will bother a compiler”

```
movl a, %eax
addl $-3, %eax
ret
```

f07.s : returns a - 3
f08.s : same assembly but adds large number that overflows until it gets to a - 3 (adding 2^32 - 3) == (a + 2^32 - 3) mod 2^32 == (a - 3) since arithmetic is done mod 2^32

Negative numbers? Tuesday