Course Overview and Introduction

CSci 2021: Machine Architecture and Organization
Lecture #1, January 20th, 2016

Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron

Overview

- Course themes
- Four realities
- Intermission: ChimeIn
- How the course fits into the CS curriculum
- Logistics

Course Theme:
Abstraction Is Good But Don’t Forget Reality

- Most CS courses emphasize abstraction
  - Abstract data types
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- Useful outcomes
  - Become more effective programmers
    - Able to find and eliminate bugs efficiently
    - Able to understand and tune for program performance
  - Prepare for later “systems” classes in CS & EE
    - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems

Great Reality #1:
Ints are not Integers, Floats are not Reals

- Example 1: Is $x^2 \geq 0$?
  - Floats: Yes!
  - Ints: $40000 \times 40000 \rightarrow 1600000000$
  - $50000 \times 50000 \rightarrow ??$

- Example 2: Is $(x + y) + z = x + (y + z)$?
  - Unsigned & Signed Ints: Yes!
  - Floats:
    - $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
    - $1e20 + (1e20 + 3.14) \rightarrow ??$

Code Security Example

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD’s implementation of getpeername
- There are legions of smart people trying to find vulnerabilities in programs

Typical Usage

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Define MSIZE 528

```c
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```
## Malicious Usage

```c
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
```

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

## Computer Arithmetic

- Does not generate random values
  - Arithmetic operations have important mathematical properties
- Cannot assume all "usual" mathematical properties
  - Due to finiteness of representations
  - Integer operations satisfy "ring" properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy "ordering" properties
    - Monotonicity, values of signs

- Observation
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

## Great Reality #2: You've Got to Know Assembly

- **Chances are, you'll never write programs in assembly**
  - Compilers are much better & more patient than you are
- **But, assembly is key to machine-level execution model**
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the lingua franca

## Great Reality #3: Memory Matters

**Random Access Memory Is an Unphysical Abstraction**

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

## Assembly Code Example

**Time Stamp Counter**

- Special 64-bit register in Intel-compatible machines
- Incremented every clock cycle
- Read with rdtsc instruction

**Application**

- Measure time (in clock cycles) required by procedure

```c
double t;
start_counter(); P(); t = get_counter();
printf("P required %f clock cycles\n", t);
```

## Code to Read Counter

- **Write small amount of assembly code using GCC's asm facility**
- **Inserts assembly code into machine code generated by compiler**

```c
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;
/* Set 'hi' and 'lo' to the high and low order bits of the cycle counter */
void access_counter(unsigned *hi, unsigned *lo) {
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1" : "=r" (*hi), "=r" (*lo) :)
        : "edx", "eax";
}
```
Memory Referencing Bug Example

```c
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
```

- Result is system specific

Memory Referencing Errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- How can I deal with this?
  - Program in Java, Ruby or ML
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

Memory System Performance Example

```c
void copyji(int src[2048][2048], int dst[2048][2048]) {
    int i, j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}

data[i][j] = src[i][j];
```

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

Why The Performance Differs

```
void copyij(int src[2048][2048], int dst[2048][2048]) {
    int i, j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

- 21 times slower (Pentium 4)

Great Reality #4: There’s more to performance than asymptotic complexity

- Constant factors matter too!
- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Example Matrix Multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)

Best code (K. Goto)

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- What is going on?

MMM Plot: Analysis

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- Effect: fewer register spills, L1/L2 cache misses, and TLB misses

Intermission: ChimeIn

- I’ll periodically break up lectures with opportunities for you to think about the material and maybe talk with the people sitting next to you
- To anonymously submit answers, we’ll use ChimeIn
- If you have a laptop with you, please go to: http://chimein.cla.umn.edu/
- And answer today’s (non-CS) question
- (Can also supposedly set up to answer with a cell-phone)

Course Perspective

- Most Systems Courses are Builder-Centric
  - Computer Architecture (CSci 4203)
    - Design pipelined processor in Verilog
  - Compilers (CSci 5161)
    - Write compiler for simple language
- 2021 is Programmer-Centric
  - Purpose is to show how by knowing more about the underlying system, one can be more effective as a programmer
  - Including, enable you to write programs that are more reliable and efficient
  - Not just a course for dedicated hackers
    - We bring out the hidden hacker in everyone
  - Cover material in this course that you won’t see elsewhere

Textbooks

- Required: Randal E. Bryant and David R. O’Hallaron,
  - http://csapp.cs.cmu.edu
  - Paper version recommended
    - Tests are open book, open notes, any paper, no electronics
    - Used quite heavily
    - How to solve labs
    - Practice problems typical of exam problems
- Optional: a book about C
  - Labs, homework, and tests require reading and writing code in C
  - Some possible suggestions listed on the course home page
Course Components

- Lectures: Higher level concepts
- Discussion (AKA Recitation) Sections
  - Thursdays. Applied concepts, important tools and skills for labs, clarification of lectures, exam coverage
- Labs (5)
  - The heart of the course, fun but often time-consuming
  - About 2 weeks each
  - Provide in-depth understanding of an aspect of systems
  - Programming and measurement
- Homework Assignments (4)
  - Practice thinking and writing, similar to tests, partially graded
- Two Midterms and One Final Exam
  - Test your understanding of concepts & mathematical principles

Electronic Resources

- Class Web Page:
  - Complete schedule of lectures, exams, and assignments
  - Copies of lectures, assignments, exams, solutions
  - Clarifications to assignments
- Moodle Page
  - Discussion forums
  - Online turn-in of labs
- Where to send electronic questions?
  1. Moodle forum
  2. Responsible TA for a homework or lab
  3. Your discussion section TAs
  4. cs2021s16-help@cs.umn.edu (general mailing list)

Policies: Assignments, Labs, And Exams

- Groups? No.
  - You must work alone on all assignments
- Hand-in process
  - Labs due online, by 11:55pm on a weekday evening
  - Homeworks due on paper, by start of class on course days
- Conflicts
  - There will be no makeup midterms
  - One excused missed midterm will be replaced by more weight on final
- Appealing grades
  - Within 7 days of completion of grading
    - Following procedure described in syllabus
  - Note, we will regrade the whole assignment/exam

Facilities

- Do labs using CSELabs Linux machines
  - Accessible from on-campus labs or remotely (SSH)
  - Get an account if you don’t have one already, login with UMN account name and password
  - Working on your own machines may sometimes be possible, but is not supported by course staff
  - Grade based on how it runs on our machines, so be sure to test there

Timeliness

- Late labs and homeworks
  - Late period is 24 hours from due date, 85% credit
  - For assignments after class, bring to instructor’s office (4-225E Keller)
  - No credit after 24 hours
- Catastrophic events
  - Major illness, death in family, … (full list in syllabus)
  - Are an exception, and can be excused
- Advice
  - The course is fast-paced
  - Once you start running late, it’s really hard to catch up

Cheating

- What is cheating?
  - Sharing code: by copying, retyping, looking at, or supplying a file
  - Coaching: helping your friend to write a lab, line by line
  - Copying code from previous course or from elsewhere on WWW
    - Only allowed to use code we supply, or from CS:APP website
- What is NOT cheating?
  - Explaining how to use systems or tools
  - Helping others with high-level design issues
- Penalty for cheating:
  - Minimum: 0 grade on assignment or exam, report to campus OSCAI
- Detection of cheating:
  - We do check
  - Our tools for doing this are better than most cheaters think!
Policies: Grading

- Exams (60%): weighted 10%, 10%, 40% (final)
- Labs (30%)
- Homework Assignments (10%)

Guaranteed:
- ≥ 90%: A-
- ≥ 80%: B-
- ≥ 70%: C-

Curve:
- Will likely apply, in your favor only, so that grade distribution is similar to historical averages.

Lab 0: Logistics Practice

- Practice how to log into a Unix machine, edit and compile a program
- “Hello, world”-style program that just prints a message
- Directions on Moodle, can submit, but won’t be graded
- More details covered in tomorrow’s discussion sections

Data Representation

- Topics
  - Bit-level operations
  - Machine-level integers and floating-point
  - C operators and things that can go wrong

- Assignments
  - L1 (Data lab): Manipulating bits

Machine-level Program Representation

- Topics
  - Assembly language programs
  - Representation of C control and data structures
  - E.g., what does a compiler do?

- Assignments
  - L2 (Bomb lab): Defusing a binary bomb
  - L3 (Buffer attack lab): Hacking a program that has a buffer overflow bug

CPU Architecture

- Topics
  - The parts of a CPU and how they work together
  - Related topic: optimizing to use CPU resources more efficiently

- Assignments
  - L4 (Architecture lab): Modify a simplified CPU and some code that runs on top of it

The Memory Hierarchy

- Topics
  - Memory technology, memory hierarchy, caches, disks, locality
  - How virtual memory works

- Assignments
  - L5 (Cache lab): Building a cache simulator and optimizing for locality.
    - Learn how to exploit locality in your programs.
Lab Rationale

- Each lab has a well-defined goal such as solving a puzzle or winning a contest
- Doing the lab should result in new skills and concepts
- We try to use competition in a fun and healthy way
  - Set a reasonable threshold for full credit
  - Post intermediate results (anonymized) on Web page for glory!

Welcome and Enjoy!