COURSE OVERVIEW

SYSTEMS I

Instructor:
Professor Emmett Witchel
Overview

• Course theme
• Five realities
• How the course fits into the CS/ECE curriculum
• Logistics
Course Theme:
Abstraction Is Good But Don’t Forget Reality

• Most CS and CE 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 & ECE
    • 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:
  - $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
  - $1e20 + (-1e20 + 3.14) \rightarrow ??$

Source: xkcd.com/571
**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) {
    /* Byte count len is minimum of buffer size and 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

/* 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;
}

#define MSIZE 528

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

/* 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;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
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: Understanding 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 language of choice!
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

```
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");
}
```
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
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) \rightarrow 3.14
fun(1) \rightarrow 3.14
fun(2) \rightarrow 3.1399998664856
fun(3) \rightarrow 2.00000061035156
fun(4) \rightarrow 3.14, then segmentation fault

• Result is architecture specific
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, then segmentation fault

Explanation:

<table>
<thead>
<tr>
<th>Saved State</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d3 ... d0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)
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

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

```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];
}

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)
The Memory Mountain

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)

Intel Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
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)

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have **exactly** the same operations count \((2n^3)\)
- What is going on?
Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

- **Gflop/s**
  - Multiple threads: 4x
  - Vector instructions: 4x
  - Memory hierarchy and other optimizations: 20x

- 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
Course Perspective

• Most Systems Courses are Builder-Centric
  • Computer Architecture
    • Design pipelined processor in Verilog
  • Operating Systems
    • Implement large portions of operating system
  • Compilers
    • Write compiler for simple language
  • Networking
    • Implement and simulate network protocols
Course Perspective (Cont.)

• Our Course is Programmer-Centric
  • Purpose is to show how by knowing more about the underlying system, one can be more effective as a programmer
  • Enable you to
    • Write programs that are more reliable and efficient
    • Incorporate features that require hooks into OS
      • E.g., concurrency, signal handlers
  • 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

- Randal E. Bryant and David R. O’Hallaron,
  - http://csapp.cs.cmu.edu
  - This book really matters for the course!
    - How to solve labs
    - Practice problems typical of exam problems

- Brian Kernighan and Dennis Ritchie,
Course Components

• Lectures
  • Higher level concepts

• Recitations
  • Applied concepts, important tools and skills for labs, clarification of lectures, exam coverage

• Labs (7)
  • The heart of the course
  • 1-3 weeks each
  • Provide in-depth understanding of an aspect of systems
  • Programming and measurement

• Exams (3)
  • Test your understanding of concepts & mathematical principles
Course Learning

- Lectures
  - Good for overview, resolving questions, flagging topics for further review
- Reading
  - Good for specifics, good preparation for lecture
- Homeworks
  - Cement your understanding, give each other questions
- Exams will require you to understand the material. Such understanding likely requires attending lecture and reading.
Getting Help

• Class Web Page
  • Complete schedule of lectures, exams, and assignments
  • Copies of lectures, assignments, exams, solutions
  • Clarifications to assignments

• Message Board
  • We will use piazza

• 1:1 Appointments
  • Office hours on web page
  • You can schedule 1:1 appointments with any of the teaching staff
Policies: Assignments (Labs) And Exams

- **Work groups**
  - You must work alone on all assignments

- **Handins**
  - Assignments due at 11:59pm on Thurs evening
  - Electronic handins using turnin (no exceptions!)

- **Conflicts for exams, other irreducible conflicts**
  - OK, but must make PRIOR arrangements at start of semester
  - Notifying us well ahead of time shows maturity and makes things easier for us (and thus we work harder to help you with your problem)

- **Testing accommodation**
  - Please submit requests within 1 week of course start

- **Appealing grades**
  - Within 7 days of completion of grading, in writing
Facilities

- See course information for lab location
- Need a cs account (mandatory!)
  - Request one here
  - [https://apps.cs.utexas.edu/udb/newaccount/](https://apps.cs.utexas.edu/udb/newaccount/)
- cs.utexas.edu machines
- Public labs
Timeliness

- Grace days
  - 4 slip days for the course
  - Covers scheduling crunch, out-of-town trips, illnesses, minor setbacks
  - Save them until late in the term!

- Lateness penalties
  - Once slip day(s) used up, get penalized 20% per day
  - No handins later than 3 days after due date

- Catastrophic events
  - Major illness, death in family,
  - Formulate a plan (with your academic advisor) to get back on track

- Advice
  - 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
  • Please identify your collaborators explicitly on HW and labs

• Penalty for cheating:
  • Removal from course with failing grade
  • Permanent mark on your record

• Detection of cheating:
  • We do check
  • Our tools for doing this are much better than most cheaters think!
Other Rules of the Lecture Hall

• Laptops: not permitted (danger, youtube)
  • See me for exceptions

• Electronic communications: *forbidden*
  • No email, instant messaging, cell phone calls, etc

• No audio or video recording

• Presence in lectures, recitations: mandatory
Policies: Grading (approximate)

- Exams (50-60%)
- Labs (30-40%)
- Homworks (5%)
- Class participation (5%)
- Graded on a curve
Programs and Data

• Topics
  • Bits operations, arithmetic, assembly language programs
  • Representation of C control and data structures
  • Includes aspects of architecture and compilers

• Assignments
  • L1 (datalab): Manipulating bits
  • L2 (archlab): Y86 (assembly) Programming
  • L3 (bomblab): Defusing a binary bomb
Architecture: Datapath & Pipelining

• Topics
  • How does a processor fetch, decode & execute code?
  • Pipelined processors, latency, and throughput

• Assignments
  • L4 (archlab): Extending a basic processor implementation
  • L5 (archlab): Modifying a pipelined processor
The Memory Hierarchy

• Topics
  • Memory technology, memory hierarchy, caches, disks, locality
  • Includes aspects of architecture and OS

• Assignments
  • L6 (memlab): Mapping the performance of the memory hierarchy
Performance Analysis

- **Topics**
  - Co-optimization (control and data), measuring time on a computer
  - Includes aspects of architecture, compilers, and OS

- **Assignments**
  - L7(perflab): Manually optimizing an algorithm
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
Welcome

and Enjoy!