Course Overview

15-213/18-243: Introduction to Computer Systems
1st Lecture, 11 January 2010

Instructors:
Bill Nace and Gregory Kesden

The course that gives CMU its “Zip”!
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 $x^2 \geq 0$?
- Float’s: Yes!
- Int’s:
  - $40000 \times 40000 \rightarrow 1600000000$
  - $50000 \times 50000 \rightarrow ??$

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

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

```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;
}

#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

```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");
}
```
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) ➔ 3.14
fun(1) ➔ 3.14
fun(2) ➔ 3.1399998664856
fun(3) ➔ 2.00000061035156
fun(4) ➔ 3.14, then segmentation fault

- **Result is architecture specific**
  - I execute up to fun(11) on my Core 2 Duo Mac
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>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>4</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>3</td>
</tr>
<tr>
<td>a[1]</td>
<td>2</td>
</tr>
<tr>
<td>a[0]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
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
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 (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}

void copyij(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];
}
```

21 times slower
(Pentium 4)
The Memory Mountain

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 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)

Gflop/s

- 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

Matrix size

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: less register spills, less L1/L2 cache misses, less TLB misses
Great Reality #5: Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance

- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues
Role within CS/ECE Curriculum

CS 415 Databases
CS 441 Networks
CS 410 Operating Systems
CS 411 Compilers
ECE 340 Digital Computation
ECE 447 Architecture
ECE 349 Embedded Systems
ECE 348 Embedded System Eng.
ECE 545/549 Capstone

CS 412 OS Practicum

Network Protocols
Processes Mem. Mgmt
Machine Code
Execution Model Memory System

CS 213
ECE 243

Foundation of Computer Systems
Underlying principles for hardware, software, and networking

CS 123
C Programming
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
### Teaching staff

#### Instructors
- Prof. Gregory Kesden
- Prof. Bill Nace

#### TA’s
- Dan Burrows
- Timothy Douglas
- Joel Feinstein
- Jason Franklin
- Alex Gartrell
- Ted Martin
- Machong (Mike) Mu
- Hunter Pitelka
- Josh Primero
- Tom Tuttle

#### Course Admin
- Cindy Chemsak (NSH 4303)

---

We’re glad to talk with you, but please send email or phone first

```cpp
if (you_love('C')){
    honk();
}
```
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 (6)
  - The heart of the course
  - 2 or 3 weeks each
  - Provide in-depth understanding of an aspect of systems
  - Programming and measurement

- Exams (2 + final)
  - Test your understanding of concepts & mathematical principles
Getting Help

- **Class Web Page**
  - http://www.cs.cmu.edu/~213
  - Copies of lectures, assignments, exams, solutions
  - Clarifications to assignments

- **Message Board**
  - http://autolab.cs.cmu.edu
  - Clarifications to assignments, general discussion
  - The only board your instructors will be monitoring (No Blackboard)
Getting Help

■ Staff mailing list
  ▪ 15-213-staff@cs.cmu.edu
  ▪ “The autolab server is down!”
  ▪ “Who should I talk to about ...”
  ▪ “This code {...}, which I don't want to post to the bboard, causes my computer to melt into slag.”

■ Teaching assistants
  ▪ I don't get “associativity”...
  ▪ Office hours, e-mail, by appointment
    ▪ Please send mail to 15-213-staff, *not a randomly-selected TA*

■ Professors
  ▪ Office hours or appointment
  ▪ “Should I drop the class?” “A TA said ... but ...”
Getting Help: Office Hours

- Kesden: see course website
- Nace: Wednesdays, 2:00pm - 4:30pm

- TAs:
  - Sundays – Thursdays, 6:00pm – 9:00pm
  - Wean Hall 5207 cluster
Policies: Assignments (Labs) And Exams

- Work groups
  - You must work alone unless told otherwise in writing

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

- Conflict exams, other irreducible conflicts
  - OK, but must make PRIOR arrangements with Prof. Kesden / Nace
  - Notifying us well ahead of time shows maturity and makes us like you more (and thus to work harder to help you out of your problem)

- Appealing grades
  - Within 7 days of completion of grading
    - Following procedure described in syllabus
  - Labs: Email to the staff mailing list
  - Exams: Talk to Prof. Kesden / Nace
Facilities

- **Labs will use the Intel Computer Systems Cluster (aka “the fish machines”)**
  - 15 Pentium Xeon servers donated by Intel for CS 213
  - Dual 3.2 Ghz 64-bit (EM64T) Nocona Xeon processors
  - 2 GB, 400 MHz DDR2 SDRAM memory
  - Rack mounted in the 3rd floor Wean Hall machine room
  - Your accounts are ready or nearing readiness

- **Getting help with the cluster machines:**
  - See course Web page for login directions
  - Please direct questions to your TA’s first
Timeliness

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

- **Lateness penalties**
  - Once grace days used up, get penalized 15% / day
  - Typically shut off all handins 2—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

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: permitted

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

- Presence in lectures, recitations: voluntary, recommended
Policies: Grading

- Exams: weighted $\frac{1}{4}, \frac{1}{4}, \frac{1}{2}$ (final)
- Labs: weighted according to effort (determined near the end)

The lower of lab score and exam score is weighted 60%, the higher 40%:

- Lab score: $0 \leq L \leq 100$,
  Exam score: $0 \leq E \leq 100$
  Total score: $0.6 \min(L, E) + 0.4 \max(L, E)$

Guaranteed:
- $> 90%$: A
- $> 80%$: B
- $> 70%$: C
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 (bomblab): Defusing a binary bomb
  - L3 (buflab): Hacking a buffer bomb
The Memory Hierarchy

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

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

Topics
- Hardware exceptions, processes, process control, Unix signals, nonlocal jumps
- Includes aspects of compilers, OS, and architecture

Assignments
- L4 (tshlab): Writing your own shell with job control
Virtual Memory

Topics
- Virtual memory, address translation, dynamic storage allocation
- Includes aspects of architecture and OS

Assignments
- L5 (mallocclab): Writing your own malloc package
  - Get a real feel for systems programming
Networking, and Concurrency

**Topics**
- High level and low-level I/O, network programming
- Internet services, Web servers
- Concurrency, concurrent server design, threads
- I/O multiplexing with select
- Includes aspects of networking, OS, and architecture

**Assignments**
- L6 (proxylab): Writing your own Web proxy
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!
Autolab Web Service

- Labs are provided by the Autolab system
  - Autograding handin system developed in 2003 by Dave O’Hallaron
  - Apache Web server + Perl CGI programs
  - Beta tested Fall 2003, very stable by now

- With Autolab you can use your Web browser to:
  - Review lab notes, clarifications
  - Download the lab materials
  - Stream autoresults to a class status Web page as you work
  - Handin your code for autograding by the Autolab server
  - View the complete history of your code handins, autoresult submissions, autograding reports, and instructor evaluations
  - View the class status page
Have Fun!