Introduction to Computer Systems

Seth Goldstein & Bruce Maggs
January 14, 2003

Topics:
- Theme
- Five great realities of computer systems
- How this fits within CS curriculum
- Staff, text, and policies
- Lecture topics and assignments
- Lab rationale

Course Theme

- Abstraction is good, but don’t forget reality!

Courses to date emphasize abstraction
- Abstract data types
- Asymptotic analysis

These abstractions have limits
- Especially in the presence of bugs
- Need to understand underlying implementations

Useful outcomes
- Become more effective programmers
- Able to find and eliminate bugs efficiently
- Able to tune program performance
- Prepare for later “systems” classes in CS & ECE
  - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems

Great Reality #1

Int’s are not Integers, Float’s are not Reals

Examples
- Is x^2 ≥ 0?
  - Float’s: Yes!
  - Int’s:
    - 40000 * 40000 --> 1600000000
    - 50000 * 50000 --> ??
- Is (x + y) + z = x + (y + z)?
  - Unsigned & Signed Int’s: Yes!
  - Float’s:
    - (1e20 + -1e20) + 3.14 --> 3.14
    - 1e20 + (-1e20 + 3.14) --> ??

Computer Arithmetic

Does not generate random values
- Arithmetic operations have important mathematical properties

Cannot assume “usual” 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
- Understanding assembly key to machine-level execution model
  - Behavior of programs in presence of bugs
  - High-level language model breaks down
  - Tuning program performance
  - Understanding sources of program inefficiency
  - Implementing system software
  - Compiler has machine code as target
  - Operating systems must manage process state

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 required by procedure
  - In units of clock cycles

```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;

/* Get cycle counter */
access_counter(&ncyc_hi, &ncyc_lo);
```

```c
void access_counter(unsigned *hi, unsigned *lo){
    asm("rdtsc; movl %edx,%0; movl %eax,%1"
        : "=r" (*hi), "=r" (*lo) :: "%edx", "%eax");
}
```

`double get_counter(){
unsigned ncyc_hi, ncyc_lo;
unsigned hi, lo, borrow;
/* Do double precision subtraction */
lo = ncyc_lo - cyc_lo;
borrow = lo > ncyc_lo;
hi = ncyc_hi - cyc_hi - borrow;
return (double) hi * (1 << 30) * 4 + lo;
}````
Measuring Time

Trickier than it Might Look

- Many sources of variation

Example

- Sum integers from 1 to n

<table>
<thead>
<tr>
<th>n</th>
<th>Cycles</th>
<th>Cycles/n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>961</td>
<td>9.61</td>
</tr>
<tr>
<td>1,000</td>
<td>8,407</td>
<td>8.41</td>
</tr>
<tr>
<td>1,000</td>
<td>8,426</td>
<td>8.43</td>
</tr>
<tr>
<td>10,000</td>
<td>82,861</td>
<td>8.29</td>
</tr>
<tr>
<td>10,000</td>
<td>82,876</td>
<td>8.29</td>
</tr>
<tr>
<td>100,000</td>
<td>8,419,907</td>
<td>8.42</td>
</tr>
<tr>
<td>1,000,000</td>
<td>8,425,181</td>
<td>8.43</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>8,371,230,591</td>
<td>8.37</td>
</tr>
</tbody>
</table>

Timing System Performance

```c
main(int argc, char** argv) {
  ...
  for (i=0; i<t; i++) {
    start_counter();
    count(n);
    times[i] = get_counter();
  }
  ...
}
```

```c
int count(int n) {
  int i;
  int sum = 0;
  for (i=0; i<n; i++) {
    sum += i;
    times[i] = get_counter();
  }
  return sum;
}
```

Great Reality #3

Memory Matters

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated

- 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 bugs especially pernicious
  - Effects are distant in both time and space
Hardware Organization (Naïve)

Implementations of Matrix Multiplication

- Multiple ways to nest loops

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

```c
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        sum = 0.0;
        for (j=0; j<n; j++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Memory Performance Example

Matmult Performance (Alpha 21164)

Memory System
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, Lisp, or ML
- Understand what possible interactions may occur
- Use or develop tools to detect referencing errors

Great Reality #4

There’s more to performance than asymptotic complexity

Constant factors matter too!
- 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
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 Curriculum

Transition from Abstract to Concrete!
- From: high-level language model
- To: underlying implementation

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 knowing more about the underlying system, leads one to be a more effective 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. Seth Goldstein (Wed 11:00-12:00, WeH 7122)
  - Prof. Bruce Maggs (Fri 2:00-3:00, WeH 4123)

- TA's
  - Dave Koes (Tue 5-6pm, WeH 3723)
  - Jiin Joo Ong (Tue 8-9pm, WeH 3108)
  - Shaheen Gandhi (Fri 12:30-1:30pm, WeH 3108)
  - Mike Nollen (Mon 3-4pm, WeH 3108)
  - Greg Reshko (Wed 2-3pm, WeH 3108)

- Course Admin
  - Dorothy Zaborowski (WeH 4116)

These are the nominal office hours. Come talk to us anytime!
(Or phone or send email)

Textbooks

- Randal E. Bryant and David R. O'Hallaron,
  - http://csapp.cs.cmu.edu/

- Samuel P. Harbison III and Guy L. Steele Jr.,
    Prentice Hall, 2002
  - http://careferencemanual.com/

Course Components

Lectures
- Higher level concepts

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

Labs
- The heart of the course
- 1, 2, or 3 weeks
- Provide in-depth understanding of an aspect of systems
- Programming and measurement

Getting Help

Web
- www.cs.cmu.edu/~213
- Copies of lectures, assignments, exams, solutions
- Clarifications to assignments

Newsgroup
- cmu.cs.class.cs213
- Clarifications to assignments, general discussion

Personal help
- Professors: door open means come on in (no appt necessary)
- TAs: please mail or zephyr first.
Policies: Assignments

Work groups
- Labs 1 – 3: You must work alone
- Labs 4 – 7: You may work in groups of two

Handins
- Assignments due at 11:59pm on specified due date
- Typically 11:59pm Wednesday evening
- Electronic handins only
- Allowed a total of up to 5 late days for the semester

Makeup exams and assignments
- OK, but must make PRIOR arrangements with either Prof. Goldstein or Maggs

Appealing grades
- Within 7 days of due date or exam date
- Assignments: Talk to the lead person on the assignment
- Exams: Talk to either Prof. Goldstein or Maggs

Cheating

What is cheating?
- Sharing code: either by copying, retyping, looking at, or supplying a copy of a file.

What is NOT cheating?
- Helping others use systems or tools.
- Helping others with high-level design issues.
- Helping others debug their code.

Usual penalty for cheating:
- Removal from course with failing grade.
- Note in student’s permanent record

Policies: Grading

Exams (40%)
- Two in class exams (10% each)
- Final (20%)
- All exams are open book/open notes.

Labs (60%)
- 7 labs (8-12% each)

Grading Characteristics
- Lab scores tend to be high
  - Serious handicap if you don’t hand a lab in
- Tests typically have a wider range of scores

Facilities

Assignments will use Intel Computer Systems Cluster (aka “the fish machines”)
- 25 Pentium III Xeon servers donated by Intel for CS 213
  - 550 MHz with 256 MB memory.
  - Rack mounted in the 3rd floor Wean machine room.
  - We’ll be setting up your accounts this week.

Getting help with the cluster machines:
- See course Web page for info
- Please direct questions to your TAs
Programs and Data

Topics
- Bits operations, arithmetic, assembly language programs, representation of C control and data structures
- Includes aspects of architecture and compilers
- Learning the tools

Assignments
- L1 Available NOW! (Due 1/24 11:59pm)
  - L1: Manipulating bits
  - L2: Defusing a binary bomb
  - L3: Hacking a buffer bomb

Performance

Topics
- High level processor models, code optimization (control and data), measuring time on a computer
- Includes aspects of architecture, compilers, and OS

Assignments
- L4: Optimizing Code Performance

The Memory Hierarchy

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

Assignments
- L4: Optimizing Code Performance

Linking and Exceptional Control Flow

Topics
- Object files, static and dynamic linking, libraries, loading
- Hardware exceptions, processes, process control, Unix signals, nonlocal jumps
- Includes aspects of compilers, OS, and architecture

Assignments
- L5: Writing your own shell with job control
Virtual memory

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

Assignments
- L6: Writing your own malloc package

I/O, 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
- L7: Writing your own Web proxy

Lab Rationale

Each lab should have a well-defined goal such as solving a puzzle or winning a contest.
- Defusing a binary bomb.
- Winning a performance contest.

Doing a lab should result in new skills and concepts
- Data Lab: computer arithmetic, digital logic.
- Bomb Labs: assembly language, using a debugger, understanding the stack
- Perf Lab: profiling, measurement, performance debugging.
- Shell Lab: understanding Unix process control and signals
-Malloc Lab: understanding pointers and nasty memory bugs.
- Proxy Lab: network programming, server design

We try to use competition in a fun and healthy way.
- Set a threshold for full credit.
- Post intermediate results (anonymized) on Web page for glory!

Have a Great Semester!