The Hardware/Software Interface

Instructor:
Luis Ceze

Teaching Assistants:
Nick Hunt, Michelle Lim, Aryan Naraghi, Rachel Sobel
Who is Luis?

PhD in architecture, multiprocessors, parallelism, compilers.
Who are you?

● 85+ students (wow!)
● Who has written programs in assembly before?
● Written a threaded program before?
● What is hardware? Software?
● What is an interface?
● Why do we need a hardware/software interface?
C vs. Assembler vs. Machine Programs

if ( x != 0 ) y = (y+z) / x;

cmpl $0, -4(%ebp)

je .L2

movl -12(%ebp), %eax

movl -8(%ebp), %edx

leal (%edx,%eax), %eax

movl %eax, %edx

sarl $31, %edx

idivl -4(%ebp)

movl %eax, -8(%ebp)

.L2:

100000110111110000100000111000000000

0111010000011000

10001010100010000100100000101000

10001011010001100010010100010100

100011010000010000000010

1000100111000010

110000011111101000001111

11110110111110000100100000111110

1110101010001000010000011000001100

10001001010001000010010000011000
C vs. Assembler vs. Machine Programs

- The three program fragments are equivalent
- You'd rather write C!
- The hardware likes bit strings!
  - The machine instructions are actually much shorter than the bits required to represent the characters of the assembler code.

```c
if ( x != 0 ) y = (y+z) / x;

cmpl $0, -4(%ebp)
je .L2
movl -12(%ebp), %eax
movl -8(%ebp), %edx
leal (%edx,%eax), %eax
movl %eax, %edx
idivl -4(%ebp)
movl %eax, -8(%ebp)
.L2:
```
HW/SW Interface: The Historical Perspective

- Hardware started out quite primitive
  - Design was expensive and the instruction set was very simple
    - E.g., a single instruction can add two integers
- Software was also very primitive
Life was made a lot better by assemblers
  ○ 1 assembly instruction = 1 machine instruction, but...
  ○ different syntax: assembly instructions are character strings, not bit strings
HW/SW Interface: Higher Level Languages (HLL's)

- Higher level of abstraction:
  - 1 HLL line is compiled into many (many) assembler lines

```
C language
specification

UserProgramming C  C Compiler  Assembler  Hardware
```
Note: The compiler and assembler are just programs, developed using this same process.
Overview

● Course themes: big and little
● Four important realities
● How the course fits into the CSE curriculum
● Logistics
● HW0 released! Have fun!
● (ready? ☑️)
The Big Theme

- THE HARDWARE/SOFTWARE INTERFACE

- How does the hardware (0s and 1s, processor executing instructions) relate to the software (Java programs)?

- Computing is about abstractions (but don’t forget reality)

- What are the abstractions that we use?

- What do YOU need to know about them?
  - When do they break down and you have to peek under the hood?
  - What bugs can they cause and how do you find them?

- Become a better programmer and begin to understand the thought processes that go into building computer systems
Little Theme 1: Representation

- All digital systems represent everything as 0s and 1s
- Everything includes:
  - Numbers – integers and floating point
  - Characters – the building blocks of strings
  - Instructions – the directives to the CPU that make up a program
  - Pointers – addresses of data objects in memory
- These encodings are stored in registers, caches, memories, disks, etc.
- They all need addresses
  - A way to find them
  - Find a new place to put a new item
  - Reclaim the place in memory when data no longer needed
Little Theme 2: Translation

• There is a big gap between how we think about programs and data and the 0s and 1s of computers

• Need languages to describe what we mean

• Languages need to be translated one step at a time
  ○ Word-by-word
  ○ Phrase structures
  ○ Grammar

• We know Java as a programming language
  ○ Have to work our way down to the 0s and 1s of computers
  ○ Try not to lose anything in translation!
  ○ We’ll encounter Java byte-codes, C language, assembly language, and machine code (for the X86 family of CPU architectures)
Little Theme 3: Control Flow

- How do computers orchestrate the many things they are doing – seemingly in parallel
- What do we have to keep track of when we call a method, and then another, and then another, and so on
- How do we know what to do upon “return”

User programs and operating systems

- Multiple user programs
- Operating system has to orchestrate them all
  - Each gets a share of computing cycles
  - They may need to share system resources (memory, I/O, disks)
- Yielding and taking control of the processor
  - Voluntary or by force?
Course Outcomes

● Foundation: basics of high-level programming (Java)

● Understanding of some of the abstractions that exist between programs and the hardware they run on, why they exist, and how they build upon each other

● Knowledge of some of the details of underlying implementations

● Become more effective programmers
  ○ More efficient at finding and eliminating bugs
  ○ Understand the many factors that influence program performance
  ○ Facility with some of the many languages that we use to describe programs and data

● Prepare for later classes in CSE
Reality 1: Ints ≠ Integers & Floats ≠ Reals

- Representations are finite

- 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

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE]; int len = 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 */
    if (KSIZE > maxlen) len = 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]; int len = 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 */
if (KSIZE > maxlen) len = maxlen;
memcpy(user_dest, kbuf, len);
return len;
}

#define MSIZE 528
void getstuff() {
char mybuf[MSIZE];
copy_from_kernel(mybuf, MSIZE);
printf("%s
", mybuf);
}
Malicious Usage

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE]; int len = 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 */
    if (KSIZE > maxlen) len = maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
Reality #2: You’ve Got to Know Assembly

● Chances are, you’ll never write a program in assembly code
  ○ Compilers are much better and more patient than you are

● But: Understanding assembly is the key to the machine-level execution model
  ○ Behavior of programs in presence of bugs
    ■ High-level language model breaks down
  ○ Tuning program performance
    ■ Understand optimizations done/not done by the compiler
    ■ Understanding sources of program inefficiency
  ○ Implementing system software
    ■ 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
/* 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) /* output */
         : /* input */
         : "%edx", "%eax"); /* clobbered */
}
```
Reality #3: Memory Matters

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

● Memory referencing bugs are 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
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:

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 (or C#, or ML, or …)
  - 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 program steps 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];
}
```

```c
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)
Reality #4: Performance isn’t counting ops

- 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 memory system is organized
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Example Matrix Multiplication

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count \((2n^3)\)
MMM Plot: Analysis

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
CSE351’s role in new CSE Curriculum

● Pre-requisites
  ○ 142 and 143: Intro Programming I and II

● One of 6 core courses
  ○ 311: Foundations I
  ○ 312: Foundations II
  ○ 331: SW Design and Implementation
  ○ 332: Data Abstractions
  ○ 351: HW/SW Interface
  ○ 352: HW Design and Implementation

● 351 sets the context for many follow-on courses
CSE351’s place in new CSE Curriculum

The HW/SW Interface

Underlying principles linking hardware and software
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.)

- This course is Programmer-Centric
  - Purpose is to show how software really works
  - By understanding the underlying system, one can be more effective as a programmer
    - Better debugging
    - Better basis for evaluating performance
    - How multiple activities work in concert (e.g., OS and user programs)
  - Not just a course for dedicated hackers
    - What every CSE major needs to know
  - Provide a context in which to place the other CSE courses you’ll take
Textbooks

- **Computer Systems: A Programmer’s Perspective, 2nd Edition**
  - Randal E. Bryant and David R. O’Hallaron
  - Prentice-Hall, 2010
  - [http://csapp.cs.cmu.edu](http://csapp.cs.cmu.edu)
  - This book really matters for the course!
    - How to solve labs
    - Practice problems typical of exam problems

- **A good C book.**
  - C: A Reference Manual (Harbison and Steele)
  - The C Programming Language (Kernighan and Ritchie)
Course Components

- **Lectures (~30)**
  - Higher-level concepts – I’ll assume you’ve done the reading in the text
- **Sections (~10)**
  - Applied concepts, important tools and skills for labs, clarification of lectures, exam review and preparation
- **Written assignments (4)**
  - Problems from text to solidify understanding
- **Labs (4)**
  - Provide in-depth understanding (via practice) of an aspect of systems
- **Exams (midterm + final)**
  - Test your understanding of concepts and principles
Resources

- **Course Web Page**
  - [http://www.cse.washington.edu/351](http://www.cse.washington.edu/351)
  - Copies of lectures, assignments, exams

- **Course Discussion Board**
  - Keep in touch outside of class – help each other
  - Staff will monitor and contribute

- **Course Mailing List**
  - Low traffic – mostly announcements; you are already subscribed

- **Staff email**
  - Things that are not appropriate for discussion board or better offline

- **Anonymous Feedback (will be linked from homepage)**
  - Any comments about anything related to the course where you would feel better not attaching your name
Policies: Grading

● Exams: weighted 1/3 (midterm), 2/3 (final)

● Written assignments: weighted according to effort
  ○ We’ll try to make these about the same

● Labs assignments: weighted according to effort
  ○ These will likely increase in weight as the quarter progresses

● Grading:
  ○ 25% written assignments
  ○ 35% lab assignments
  ○ 40% exams
Welcome to CSE351!

- Let’s have fun
- Let’s learn – together
- Let’s communicate
- Let’s set the bar for a useful and interesting class

Many thanks to the many instructors who have shared their lecture notes – I will be borrowing liberally through the qtr – they deserve all the credit, the errors are all mine

- UW: Gaetano Borriello (Inaugural edition of CSE 351, Spring 2010)
- CMU: Randy Bryant, David O’Halloran, Gregory Kesden, Markus Püschel
- Harvard: Matt Welsh
- UW: Tom Anderson, Luis Ceze, John Zahorjan