Computer Systems Organization

Lecture 1: Overview

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Who Am I?

• Mohamed Zahran (aka Z)
• Computer architecture/OS/Compilers Interaction
• http://www.mzahran.com
• Office hours: Mon 3:00-5:00 pm
• Room: WWH 320
Formal Goals of This Course

• What happens under the hood when you boot your computer and start running applications?
• How do software and hardware interact?
• How to write a program and execute it on a computer?
• This course is programmer-centric
  – Understanding of underlying system makes a more effective programmer
  – Bring out the hidden hacker in everyone
Informal Goals of This Course

• To get more than an A
• To learn computer systems and enjoy it
• To use what you have learned in MANY different contexts
• To start your research project if you want
• To enjoy the course!
The Course Web Page

http://cs.nyu.edu/courses/spring13/CSCI-UA.0201-003/index.html

• Lecture slides
• Info about mailing list, labs, ...
• Useful links (manuals, tools, ...)
Textbooks

KR

BO

THE C PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

SECOND EDITION

COMPUTER SYSTEMS
A Programmer's Perspective

BRYANT • O’HALLARON

SECOND EDITION
Course Components

• Lectures
  – Higher level concepts

• Programming labs (5)
  – The heart of the course
  – 1-3 weeks each
  – Provide in-depth understanding of some aspect of systems

• One final exam
Course Syllabus

• Basic C
  – L1 (Rabin-Karp Lab)

• Assembly: Representation of program and data
  – L2 (Bomblab) and L3 (Buflab)

• Memory hierarchy

• Exceptional control flow: exceptions, process control, signals
  – L4 (Proclab)

• Virtual Memory: address translation, allocation
  – L5 (Malloclab)

• Networking and concurrency
Policies: Assignments (Labs)

• You must work alone on all assignments
  – Post all questions on the mailing list,
  – You are encouraged to answer others’ questions, but refrain from explicitly giving away solutions.

• Hand-ins
  – Assignments due at 11:59pm on the due date
  – Everybody has 5 grace days for the whole semester
  – Zero score if a lab is handed in more than 3 days late
Facilities

• Lab environments:
  – Use official class VM image
    • Download (free) virtualbox for Windows/Linux
    • Download VM appliance from course web page
Cheating

• **What is cheating?**
  – Sharing code: by copying, looking at others’ files
  – Coaching: helping your friend to write a lab
  – Copying code from previous course or from elsewhere on WWW
    • You can only use code we supply

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

• Lab : 60%
• Final : 40%
Course Theme:
Abstraction Is Good But Don’t Forget Reality

• Most CS courses emphasize abstraction
• This class:
  – Helps you peek ``under-the-hood”
• Useful outcomes
  – Become more effective programmers
    • Debug problems
    • Tune performance
  – Prepare for later “systems” courses in CS
    • Compilers, Operating Systems, Networks, Computer Architecture, Distributed Systems
Reality #1:
Ints are not Integers, Floats are not Reals

• $x^2 \geq 0$?

• $(x + y) + z = x + (y + z)$?

Overflow!!

$1e20 + (-1e20 + 3.14)! = 3.14$
Reality #2: You’ve Got to Know Assembly

• No need to program in assembly
• Knowledge of assembly helps one understand machine-level execution
  – Debugging
  – Performance tuning
  – Writing system software (e.g. compilers, OS)
  – Creating / fighting malware
    • x86 assembly is the language of choice!
Reality #3: Memory Matters

• Memory is not unbounded
  – It must be allocated and managed
• Memory referencing bugs especially wicked
• Memory performance is not uniform
  – Cache and virtual memory effects can greatly affect performance
Memory Referencing Errors

- C/C++ let programmers make memory errors
  - Out of bounds array references
  - Invalid pointer values
  - Double free, use after free
- Errors can lead to nasty bugs
  - Corrupt program objects
  - Effect of bug observed long after the corruption
Memory Referencing Bug Example

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

fun(0) = 3.14
fun(1) = 3.14
fun(2) = ?
fun(3) = ?
fun(4) = ?
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 getpeereame
- 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
”, 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);
    ...
}
Reality #4: Asymptotic performance is not always sufficient

- Constant factors matter
- Even operation count might not predict performance
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Memory System Performance Example

- Performance depends on access patterns

```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)
Example Matrix Multiplication

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

- Standard desktop computer and compiler
- Both implementations have \textbf{exactly} the same operations count \((2n^3)\)
Carnegie Mellon

MMM Plot: Analysis

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

Gflop/s

Reason for 20x: Blocking or tiling, loop unrolling, array scalarization

Effect: fewer register spills, L1/L2 cache misses, and TLB misses
Reality #5: Computer is more than the CPU

• They need to do I/O (get data in and out)

• They communicate with each other over networks
  • Concurrent operations by autonomous processes
  • Coping with unreliable media
  • Cross platform compatibility
  • Complex performance issues
A Little Bit of History

Eckert and Mauchly

- 1st working electronic computer (1946)
- 18,000 Vacuum tubes
- 1,800 instructions/sec
- 3,000 ft³
A Little Bit of History

- Maurice Wilkes

EDSAC 1 (1949)

http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99/

1st stored program computer
650 instructions/sec
1,400 ft³
A Little Bit of History

- 1954 IBM developed 704
- All programming done in assembly
- Software costs exceed hardware costs!
A Little Bit of History

- **Fortran I (project 1954-57)**
- The main idea is to translate high level language to assembly
- Many thought this was impossible!
- In 1958 more than 50% of software in assembly!
- Development time halved!

John Backus
(December 3, 1924 – March 17, 2007)
Problem $\rightarrow$ Algorithm Development $\rightarrow$ Programmer

1. High Level Language
2. Assembly Language
3. Machine Language
4. Microarchitecture
5. Logic Level

Device Level $\rightarrow$ Semiconductors $\rightarrow$ Quantum
Source Code to Execution

C source → Compiler → Assembly → Assembler → Object File → Linker → Library → Executable

DLL
Conclusions

• This first lecture was just an overview. More fun is yet to come!

• Computer system can be viewed as layers of abstractions $\rightarrow$ knowing these layers helps us see the big picture