CSCI-UA.0201

Computer Systems Organization

Overview

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

• Mohamed Zahran (aka Z)
• Research interest: Computer architecture/OS/Compilers Interaction
• http://www.mzahran.com
• Office hours: Tu/Th 2:00-3:00 pm
• My office: WWH 320
• If you want to meet outside these office hours, please email me first.
What we will learn in this course

• What happens under the hood when you boot your computer and start running applications?
• How do software and hardware interact?
• This course is programmer-centric
  – Understanding of underlying system makes you a more effective programmer and helps you find hidden bugs!
  – Bring out the hidden hacker in everyone
  – Be way more efficient debugger
  – Tune your programs for performance
But also we want

• To use what you have learned in MANY different contexts
• To start your research project if you want
• To know the big picture of the whole computing stack.
• To enjoy the course!
Course Components

• Lectures
  – Higher level concepts
  – slides + reading material from the textbook

• Programming labs (~3 of them → 15%)
  – 1-2 weeks each
  – Provide in-depth understanding of some aspect of systems

• Homework assignments (~2-3 of them → 5%)
  – Labs do not cover all the material we will study!
  – For theoretical knowledge

• Exam 1 (40%)
• Exam 2 (40%)
Exams

• Closed books but cheat sheets allowed
  – Exam 1: one cheat sheets (i.e. 2 A4 pages)
  – Exam 2: two cheat sheets (i.e. 4 A4 pages)
• No electronic devices allowed
• Cheat sheet:
  – Feel free to write anything on it (front and back), before the exam.
  – You are allowed to take it to the exam.
Main Topics

• Basic C
• Representation of program and data
• Memory hierarchy and systems hardware
• Dynamic memory allocation
• Virtual Memory
• Concurrency & Processes
Policies: Assignments (Labs)

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

• Hand-ins
  – Labs due at **11:59pm on the due date**
  – Submitted through NYU classes
Integrity and Collaboration

• What is cheating?
  – Sharing code: by copying, retyping, looking at, or supplying a file
  – Describing code: verbal description of code from one person to another.
  – Coaching: helping your friend to write a lab, line by line
  – Searching the Web for solutions
  – Copying code from a previous course or online solution
    • You are only allowed to use code we supply

• What is NOT cheating?
  – Explaining how to use systems or tools
  – Helping others with high-level design issues

• Ignorance is not an excuse

We have sophisticated tools for detecting code plagiarism
What can positively affect your grade (i.e. help be boost your final grade)?

• Participate in the forums at NYU classes, by asking questions or answering questions of other.

• Submit your assignments on time and in the correct format.
What can negatively affect your grade?

• Coming up with a lot of excuses to get extensions (except documented health problems), or higher grades, examples:
  – My machine crashed just before the deadline. You better submit a version each time you complete part of the lab. You are allowed to submit several times. We will grade the last submission.
  – I have many assignments on other courses so please give me extension.
  – I submitted one minute after the deadline but the server did not accept it.

• Asking questions on the forums that have been asked before.
  – Yes, you better read the previous questions and answers on the forums. It is a good way of studying because people may ask good questions that did not come to your mind.

• Asking about a rule that has already been mentioned in an announcement on NYU classes, slides, or assignment description.
  – You need to pay attention.
Web Presence

• **Course webpage:**
  - sec 1: http://cs.nyu.edu/courses/fall18/CSCI-UA.0201-001/
  - sec 3: http://cs.nyu.edu/courses/fall18/CSCI-UA.0201-003/

• **NYU Classes**
  - Announcements
  - Forums (arranged by topics)
  - Submitting labs and assignments
Course Theme:
Abstraction Is Good But Don’t Forget Reality

• Most CS courses emphasize abstraction
  – e.g. data types
• Abstracts are good but have limitations
  – Especially in the presence of bugs!
• This class helps you:
  – peek under-the-hood
  – become more effective programmers
    • Debug problems
    • Tune performance
  – prepare for later courses in CS
    • Compilers, Operating Systems, Computer Architecture, Distributed Systems, parallel computing, ...
Reality #1:  
Ints are not Integers  
Floats are not Reals

- $x^2 \geq 0$?  
- $(x + y) + z = x + (y + z)$?  

Overflow!!

$10^{20} + (-10^{20} + 3.14)! = 3.14$  

Source: xkcd.com/571
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)
{
  int a[2];
  double d[1] = {3.14};
  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
- There are legions of smart people trying to find vulnerabilities in programs
/* 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);
    . . .
Reality #4: Asymptotic performance is not always sufficient

- Factors like memory access, communication, etc. matter
- Even operation count might not predict performance
- Must understand system to optimize performance
  - How are 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

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

- Performance depends on access patterns
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 exactly the same operations count \((2n^3)\)

Best code (K. Goto)

Triple loop

160x
Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

- Memory hierarchy and other optimizations: 20x
- Vector instructions: 4x
- Multiple threads: 4x

- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization
- **Effect:** fewer register spills, L1/L2 cache misses, and TLB misses
Reality #5: Computers are 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

1st stored program computer
650 instructions/sec
1,400 ft³

EDSAC 1 (1949)

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

High Level Language

$\downarrow$

Compiler (translator)

Assembly Language

$\downarrow$

Assembler (translator)

Machine Language

$\downarrow$

Control Unit (Interpreter)

Microarchitecture

$\downarrow$

Microsequencer (Interpreter)

Logic Level

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

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

Diagram shows the steps from C source code to an executable program, including compiler, assembler, linker, and loader stages.
Conclusions

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

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

Welcome ... And Enjoy ☺