CSCI-UA.0201

Computer Systems Organization

Introduction

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
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.
Course Information and Resources

• Course web page (general info, syllabus, etc.)
  http://cs.nyu.edu/wies/teaching/cso-fa19/

• Piazza (announcements, course related discussions)
  https://piazza.com/class/jznb6uqqbcd4e6

  You should already be enrolled. Please complete the questionnaire!

• Github (sample code, assignment submission)
  https://github.com/nyu-cso-005-fa19

• NYU classes (grade distribution only)
Textbook

Randy Bryant

Dave O’Hallaron
Important Dates

• Class meetings
  – Monday and Wednesdays, 3:30-4:45pm

• Recitations (Goktug Saatcioglu)
  – Thursdays, 12:30-1:45pm

• Office hours
  – Thomas Wies: Tuesdays, 4-5pm in 60FA 403
  – Goktug Saatcioglu: Fridays, 3-4pm in WWH 905
  – or by appointment

• Midterm exam
  – Wednesday, Oct 23, 3:30-4:45pm

• Final exam
  – Wednesday, Dec 18, 4:00-5:50pm
Course Components

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

• Assignments and Programming labs (30%)
  – roughly bi-weekly
  – provide in-depth understanding of some aspect of systems
  – also serve as exam preparation

• Midterm Exam (30%)

• Final Exam (40%)
Submission Policy

• You must work alone on all assignments

• Pay attention to due dates/times (announced on Piazza).

• Submission is via Github (more on that later).

• Late submissions will be graded with a 10% penalty per (started) day of late submission.

• No solutions will be accepted one week after the submission deadline.
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*
Main Topics

• Basic C Programming
• Representation of programs and data
• Memory hierarchy and systems hardware
• Basic Assembly Programming
• Dynamic memory allocation
• Virtual Memory
• Concurrency & Processes
Abstraction in Computer Science

The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer. Edsger Dijkstra

- Computer system can be viewed as layers of abstractions
- Most CS courses emphasize abstraction
  - e.g. data types, high-level programming languages
Abstraction in Computer Science

[Computer scientists] are individuals who can rapidly change levels of abstraction, simultaneously seeing things 'in the large' and 'in the small'. Donald Knuth
Course Goals

• Computer system can be viewed as layers of abstractions
• Sometimes you must break through these abstractions
• This class helps you:
  – peek *under-the-hood*
  – understand these layers to see the big and the small picture
  – 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

• $x^2 \geq 0$? Overflow!!
• $(x + y) + z = x + (y + z)$?

10^{20} + (-10^{20} + 3.14) \neq 3.14

Source: xkcd.com/571
Arithmetic Overflow

Ariane 5 maiden flight

**Cause:** software error in inertial reference system
64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer.
Reality #2:
You've Got to Know Assembly

• Usually 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 influence 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
  – Security vulnerabilities
  – Effect of bug observed long after the corruption
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) = ?
Heartbleed Bug - I ♥ love OpenSSL

/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);

value of payload controlled by attacker
memcpy may copy memory beyond payload buffer
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

- 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
Example Matrix Multiplication

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

Gflop/s

- Standard desktop computer and compiler
- Both implementations have *exactly* the same operations count \(2n^3\)
Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Gflop/s

- 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 ft$^3$
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)
A Little Bit of History

• C (1973)
• General purpose language that efficiently translates to assembly.
• Still de facto the language of choice for systems programming
• Current standard: C18

Dennis Ritchie
(September 9, 1941 – October 12, 2011)
Problem → Algorithm Development → Programmer

High Level Language → Compiler (translator)

Assembly Language → Assembler (translator)

Machine Language → Control Unit (Interpreter)

Microarchitecture → Microsequencer (Interpreter)

Logic Level

Device Level → Semiconductors → Quantum
Source Code to Execution

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

Loader → DLL