Systems Programming and Computer Architecture
(252-0061-00)

Timothy Roscoe
Herbstsemester 2016
1: Introduction

Systems Programming and Computer Architecture
252-0061-00, Herbstsemester 2016
Timothy Roscoe
This course covers in depth...

• What makes programs go fast (and slow)
  – How to write fast and correct code
  – How to write good systems code
• Programming in C
  – Still the systems programming language of choice
• Programming in Assembly Language
  – What the machine understands
• Programs as more than mathematical objects
  – E.g. how does Facebook work?
Logistics

• Lectures 10:00-12:00
  – Tuesdays: HG G 5
  – Wednesdays: NO C 60
• Tutorial sessions – very important!
  – Thursday 13:00 – 15:00, various rooms
    • CHN D 42, CHN G 46, LEE C 104, LEE D 105, LFW C 1, ML J 34.1, ML J 37.1
  – Tools and skills for lab exercises
  – Clarification of lectures, etc.
  – Knowledge needed for exams, but not in the lectures!
• There will be a session this Thursday
  – Sign up at: http://echo.ethz.ch/
  – Check the course web page on Wednesday
Language

• I’ll teach in English (and C...)
  – If I speak too fast, or say something unclear, raise your hand!
  – Please ask questions!

• Assistants all speak German and English

• Examination (100% of grade):
  – Paper will be in English
  – Answers may be in German or English
Tomorrow: NO C 60
Where is that?
More logistics

• Web site:
  
  http://www.systems.ethz.ch/courses/fall2016/SPCA
  
  – Lecture notes should appear in advance on web site
  – The notes are not intended to be understood without lectures...

• Procedure for answering additional questions:
  1. Ask your friends
  2. Check the web
  3. Ask your teaching assistant
  4. Ask another teaching assistant
  5. Email me (troscoe@inf.ethz.ch)
Acknowledgements

• Lots of material from the famous **CS 15-213** at Carnegie Mellon University
  – Basis for the **book**

• Some C programming sides adapted from **CSE333** at University of Washington
  – Many thanks to (ex-)Prof. Steve Gribble

• New material:
  – Multicore, devices, etc.
  – All my fault 😊
Questions?
1.1: What is “Systems”? 

Systems Programming and Computer Architecture
252-0061-00, Herbstsemester 2016
Timothy Roscoe
“Systems” as a field

• Encompasses:
  – Operating systems
  – Database systems
  – Networking protocols and routing
  – Compiler design and implementation
  – Distributed systems
  – Cloud computing & online services

• On and above *hardware/software* boundary
You are here:

Computational Science
Visual Computing
Computer Security
Etc.

*Systems topics*

Software

Hardware

Systems Programming and Computer Architecture

Processor design
Digital design, electrical engineering
You are here:

Application areas: Visual Computing, Big Data, Numerical Analysis, Machine Learning, etc.

- Compiler Design
- ... (Dashed line indicating a placeholder)
- Information Systems
- Embedded Systems

- Networks and Operating Systems
- Data modelling and Databases

- Systems Programming and Computer Architecture
- Parallel Programming
- Digital Circuits
Motivation

• Most CS courses emphasize **abstraction**
  – Abstract data types (objects, contracts, etc.)
  – Asymptotic analysis (worst-case, complexity)
• These abstractions have **limitations**
  – Often don’t survive contact with reality
  – Especially in the presence of bugs
  – Need to understand details of underlying implementations
Summary: Course Goals

• Become more effective programmers
  – Find and eliminate bugs efficiently
  – Understand and tune for program performance

• Prepare for later systems classes at ETHZ
  – Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems
1.2: Motivation - Five realities

Systems Programming and Computer Architecture
252-0061-00, Herbstsemester 2016
Timothy Roscoe
Reality #1:

int’s are not integers.
float’s are not real numbers.
ints are not integers, floats are not reals

• Is $x^2 \geq 0$?
  – floats: Yes!
  – ints:
    • $40000 \times 40000 \rightarrow 1600000000$
    • $50000 \times 50000 \rightarrow ??$

• Is $(x + y) + z = x + (y + z)$?
  – unsigned & signed ints: Yes!
  – floats:
    • $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
    • $1e20 + (-1e20 + 3.14) \rightarrow ??$

http://xkcd.com/571
Computer arithmetic

• Does not generate random values
  – Arithmetic operations have important mathematical properties

• Cannot assume all “usual” mathematical 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
Reality #2:

You’ve got to know assembly.
You’ve got to know assembly

- Chances are, you’ll never write program in assembly
  - Compilers are much better & more patient than you are
- But: understanding assembly is **key** to 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**
    - Compiler has machine code as target
    - 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
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* 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)
         : "%edx", "%eax");
}
```
Reality #3:

Memory matters.
RAM is an unrealistic abstraction.
Memory matters

• Memory is **not unbounded**
  – It must be allocated and managed
  – Many applications are memory-dominated
• Memory **referencing bugs** 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

typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}

fun(0) -> 3.14
fun(1) -> 3.14
fun(2) -> 3.1399998664856
fun(3) -> 2.00000061035156
fun(4) -> 3.14
fun(6) -> Segmentation fault

Actual results are system-specific...
Memory Referencing Bug

typedef struct {
    int a[2];
    double d;
} struct_t;

fun(0) -> 3.14
fun(1) -> 3.14
fun(2) -> 3.1399998664856
fun(3) -> 2.00000061035156
fun(4) -> 3.14
fun(6) -> Segmentation fault

Explanation:

<table>
<thead>
<tr>
<th>Critical State</th>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>a[1]</td>
</tr>
<tr>
<td>5</td>
<td>a[0]</td>
</tr>
<tr>
<td>4</td>
<td>d3 ... d0</td>
</tr>
<tr>
<td>3</td>
<td>d7 ... d4</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>
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 C#, or ML, or Scala, or Haskell, or... (Rust?)
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors
Reality #4:

There’s much more to performance than asymptotic complexity
There’s much more to performance than asymptotic complexity

- **Constant factors** matter too!
- Even 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 to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Memory system performance

Hierarchical memory organization

Performance depends on access patterns
  – Including how step through multi-dimensional array

```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];
}
```

```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];
}
```

Intel Core i7 2.7 GHz

5.2 ms

162 ms !
The Memory Mountain

Intel Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)
Example: matrix multiplication

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

Gflop/s

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have **exactly** the same operations count \((2n^3)\)
- What is going on?
Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Gflop/s

- Why? Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, ...
- **Effect:** less register spills, less L1/L2 cache misses, less TLB misses
Reality #5:

Computers don’t just execute programs
Computers don’t just run programs

• They need to get data in and out
  – I/O critical to program reliability and performance

• They communicate over networks
  – Many system-level issues arise with a network
    • Concurrent operations by autonomous processes
    • Coping with unreliable media
    • Cross-platform interoperability
    • Complex performance issues
Lies our teachers tell us

Figure 2.1 A modern computer system.
Lies our teachers tell us

Figure 1.4
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

systems, but all systems have a similar look and feel. Don’t worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.
A modern SoC
Summary

1. ints are not integers. 
   floats are not real numbers.
2. You’ve got to know assembly.
3. Memory matters. 
   RAM is an unrealistic abstraction.
4. There’s much more to performance than asymptotic complexity.
5. Computers don’t just evaluate programs.
1.3: What I’ll assume you know

Systems Programming and Computer Architecture
252-0061-00, Herbstsemester 2016
Timothy Roscoe
Courses already

• Programming & software engineering
• Parallel programming
• Data structures and algorithms
• Digital Circuits
• Discrete Mathematics
What I’ll assume you know #1:

- Binary, and Hexadecimal notation
- Memory, addresses, bytes, and words
- Byte-ordering  
  (Big-Endian, Little-Endian)
- Boolean algebra  
  (and, or, not, xor)
- Generalized Boolean algebra  
  (bitwise operations on words as bit vectors)
What I’ll assume you know #2:

- Processor architecture, pipelines
- MIPS assembly (we’ll work in 64-bit x86)
  - Registers
  - Addressing modes
  - Instruction formats
- Basic memory systems
  - cache architectures
  - virtual memory
  - I/O devices

Is this a problem? Say now!
What I’ll assume you know #3:

- Software engineering
  - Object-orientation
  - Design-by-contract
  - Strong typing

- Concurrency and parallelism
  - Threads
  - Locks, mutexes, condition variables

Is this a problem? Say now!
What I’ll assume you know #4:

- How to write programs
  - Eiffel
  - C# (or Java)
  - A bit of C
  - MIPS assembly
Textbooks

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

- Brian Kernighan and Dennis Ritchie,

Key book for the course!
You might also enjoy

• Peter van der Linden

• Samuel Harbison and Guy Steele
  – C: A Reference Manual
  – 5th edition 2002
OK!
Just in case...
Binary representations

• Base 2 number representation
  – Represent $15213_{10}$ as $11101101101101_2$
  – Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
  – Represent $1.5213 \times 10^4$ as $1.1101101101101_2 \times 2^{13}$

• Electronic implementation
  – Easy to store with bistable elements
  – Reliably transmitted on noisy and inaccurate wires
Encoding byte values

- **Byte = 8 bits**
  - Binary: 00000000\(_2\) to 11111111\(_2\)
  - Decimal: 0\(_{10}\) to 255\(_{10}\)
    - First digit must **not be 0** in C
  - Hexadecimal: 00\(_{16}\) to FF\(_{16}\)
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B\(_{16}\) in C as 0xFA1D37B
      - Or 0xFA1D37B

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
Byte-oriented memory organization

- Programs refer to *virtual addresses*
  - Conceptually, a very large array of bytes
  - Secretly, a hierarchy of different memory types
  - System provides *address space* private to particular “process”
    - Program being executed
    - Program can clobber its own data, but not that of others

- Compiler + runtime system control allocation
  - Where different program objects should be stored
  - All allocation within single virtual address space
Machine words

• Machines have a “word size”
  – Nominal size of integer-valued data
    • Including addresses
  – Many current machines use 32-bit (4 byte) words
    • Limits addresses to 4GB
    • Becoming too small for memory-intensive applications
  – Modern machines often use 64-bit (8 byte) words
    • Potential address space \( \sim 1.8 \times 10^{19} \) bytes
    • x86-64 machines support 48-bit addresses: 256 Terabytes
  – Machines support multiple data formats
    • Fractions or multiples of word size
    • Always integral number of bytes
Word-oriented memory

- Addresses specify byte locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
Byte ordering

• How should bytes within multi-byte word be ordered in memory?
  – Big Endian: Sun, PPC, Internet
    • Least significant byte has highest address
  – Little Endian: x86
    • Least significant byte has lowest address

• Origin: “Gullivers Reisen” (Gulliver’s Travels)

• Which end to crack a soft-boiled egg?

Origin: “Gullivers Reisen” (Gulliver’s Travels)

Egg in “little endian” configuration (Wikipedia)
Byte ordering example

• Big Endian
  – Least significant byte has highest address

• Little Endian
  – Least significant byte has lowest address

• Example
  – Variable x has 4-byte representation 0x01234567
  – Address given by &x is 0x100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Reading byte-reversed listings

- Disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- Example fragment:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00</td>
<td>cmp %ebx $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- Deciphering numbers:
  - Value: 0x12ab
  - Pad to 4 bytes: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse (endian): ab 12 00 00
Examining data representations

• Code to print byte representation of data
  – Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++) {
        printf("0x%p\t0x%.2x\n", start+i, start[i]);
    }
}
```

printf directives:
  %p: print pointer
  %x: print hexadecimal
show_bytes example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result using Linux on 32-bit Intel x86:

```c
int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

AND: A&B = 1 when A=1 and B=1

(Exclusive) OR: A|B = 1 when A=1 or B=1

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Exclusive-OR (XOR): A^B = 1 when A=1 or B=1 but not both

NOT: ~A = 1 when A=0
Application of boolean algebra

- Applied to digital systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0

\[
A \& \neg B \lor \neg A \& B = A^\lor B
\]
General boolean algebras

• Operate on bit vectors
  – Operations applied bitwise

\[
\begin{array}{c}
01101001 \\ & 01010101 \\
\hline
01000001
\end{array}
\quad \begin{array}{c}
01101001 \\ | 01010101 \\
\hline
01111101
\end{array}
\quad \begin{array}{c}
01101001 \\ ^ 01010101 \\
\hline
00111100
\end{array}
\quad \begin{array}{c}
\sim 01010101 \\
\hline
10101010
\end{array}
\]

• All of the properties of boolean algebra apply