System Programming

Lecture 1

Yeongpil Cho

Hanyang University

About me

- 조영필 **(Yeongpil Cho)**
  - A system security researcher
  - Designing new SW/HW techniques for better security
    - Kernels & Firmware
    - Applications
    - (co-)Processors
- **Contact**
  - ypcho@hanyang.ac.kr
  - TA: 강범우
    - qjadn0630@hanyang.ac.kr

Objectives

- **Computer systems consists of various hardware and software components.**
- **Most system implementations share some fundamental design principles.**
  - e.g., Android Phone, iPhone, Windows desktop
- **We are going to generally explore those principles**
  - to understand how programs run on the system
  - to understand what affect programs
  - to, ultimately, develop better programs

Materials

- **Textbook**
  - Computer Systems: A Programmer’s Perspective 3rd Edition
    - Randal E Bryant, David R O’Hallaron
- **Reference Course**
  - Carnegie Mellon University’s Intro to Computer Systems

Evaluation

- **Exam** **60%**
  - Midterm 30%
  - Final 30%
- **Lab** **30%**
- **Attendance** **10%**
  - 3 tardies → 1 absence

Topics

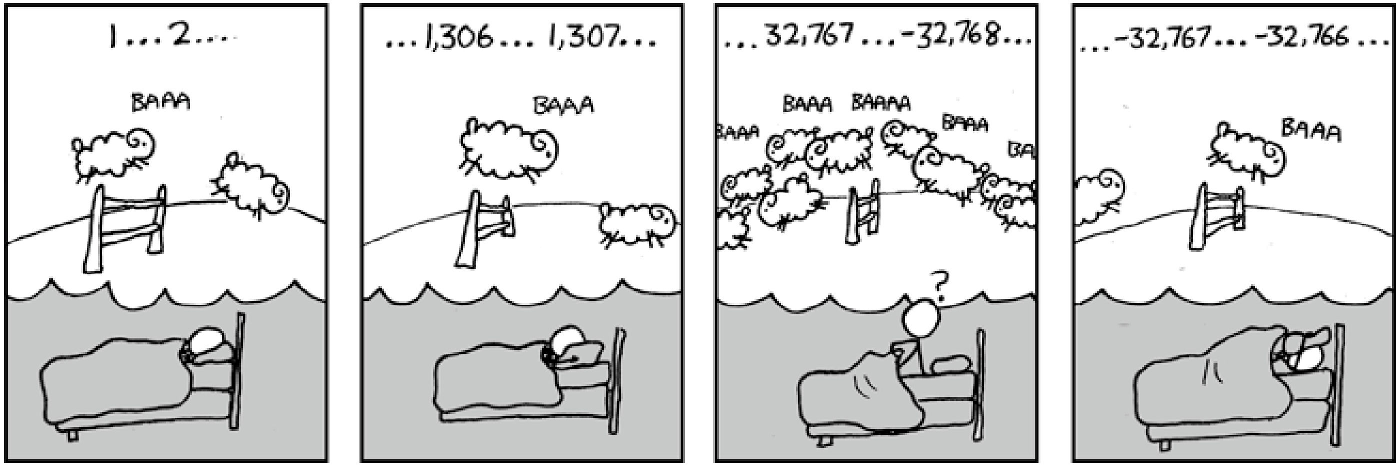
- **Overview of the system programming course**

Abstraction Is Good But Don't Forget Reality

- **Most CS and CE courses emphasize abstraction**
  - Abstract Data Type
    - List, Tree, …
  - Asymptotic analysis
    - Big-O/Θ/Ω, …
- **These abstractions are good for understanding, but we need to know reality**
  - for better performance optimization
  - for better secure implementation

Great Reality #1:
Ints are not Integers, Floats are not Reals

- **Example 1: Is $x^2 \geq 0$?**
  - Float's: Yes!
  - Int's:
    - 40,000 * 40,000 → 1,600,000,000
    - 50,000 * 50,000 → 2,500,000,000 ?? → No! → -1,794,967,296
      - Range of Integer value
        - » -2,147,483,648 ~ 2,147,483,647



Great Reality #1:
Ints are not Integers, Floats are not Reals

- **Example 2: Is (x + y) + z = x + (y + z)?**
  - Unsigned & Signed Int's: Yes!
  - Float's:
    - (1e20 + -1e20) + 3.14 → 3.14
    - 1e20 + (-1e20 + 3.14) → 3.14 ?? → No! → 0
    - Why?
      - Addition of float numbers
        - » (1) equalize exponents to the larger one
        - » (2) adjust mantissa according to the modified exponent

      - 1e20 → 0b0 11000001 01011010111100011101100
      - -1e20 → 0b1 11000001 01011010111100011101100
      - 3.14   → 0b0 10000000 10010001111010111000011

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

Great Reality #2: You've Got to Know Assembly

- **Chances are, you'll never write programs 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 models break 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**
  - a 64-bit register on x86
  - increment on every CPU clock
  - readable using `rdtsc` instruction
- **Elapsed time of a procedure.**

```
unsinged long long t1, t2;
t1 = read_counter();
P();
t2 = read_counter();

printf("execution time of P():%d clock cycles\n", t2-t1);
```

Assembly Code Example

- Implementing through inline assembly code
- Or, linking an object file of an assembly code

```
unsigned long long read_counter(){
  unsigned long long hi, lo;

  asm("rdtsc            \n
       movl %%edx, %0   \n
       movl %%eax, %1   \n")"
      : "=r" (hi), "=r" lo :: "%edx", "%eax");
                  return (lo | (hi << 32);
}
```

Great Reality #3: Memory Matters

Random Access Memory Is an Unphysical Abstraction

- **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 Example

```
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;
}
```

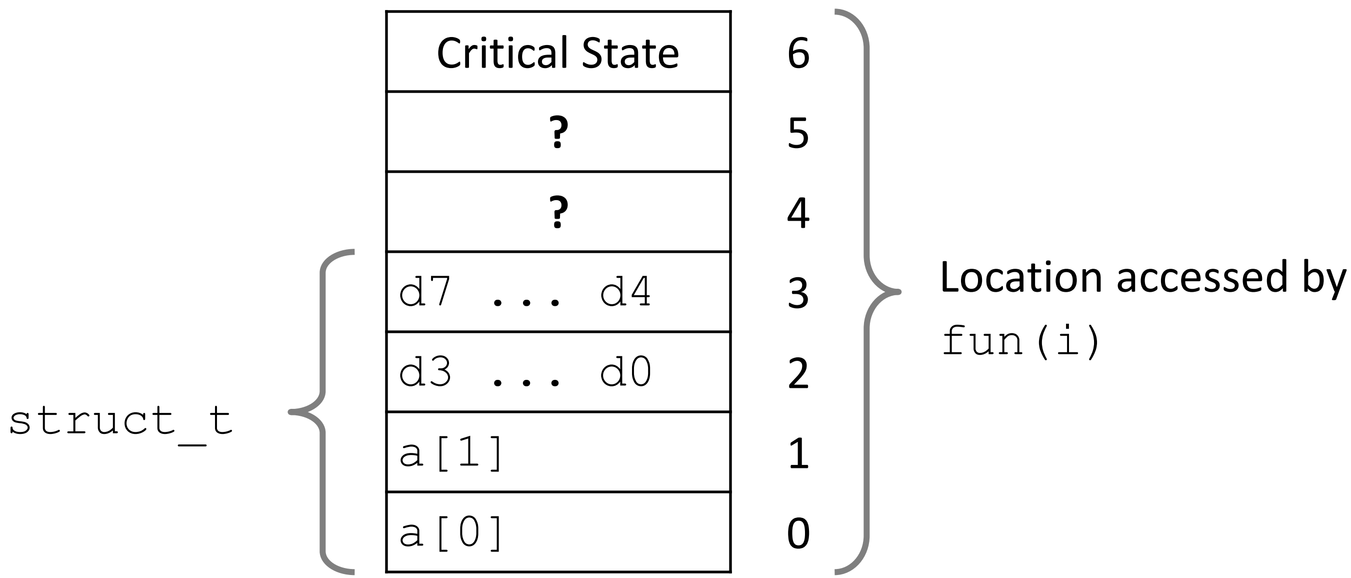
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(6) → Segmentation fault

Memory Referencing Bug Example

```
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:



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, Ruby, Python, ML, …
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

Great Reality #4: There's more to performance than asymptotic complexity

- **Constant factors matter too!**
- **And 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 Example

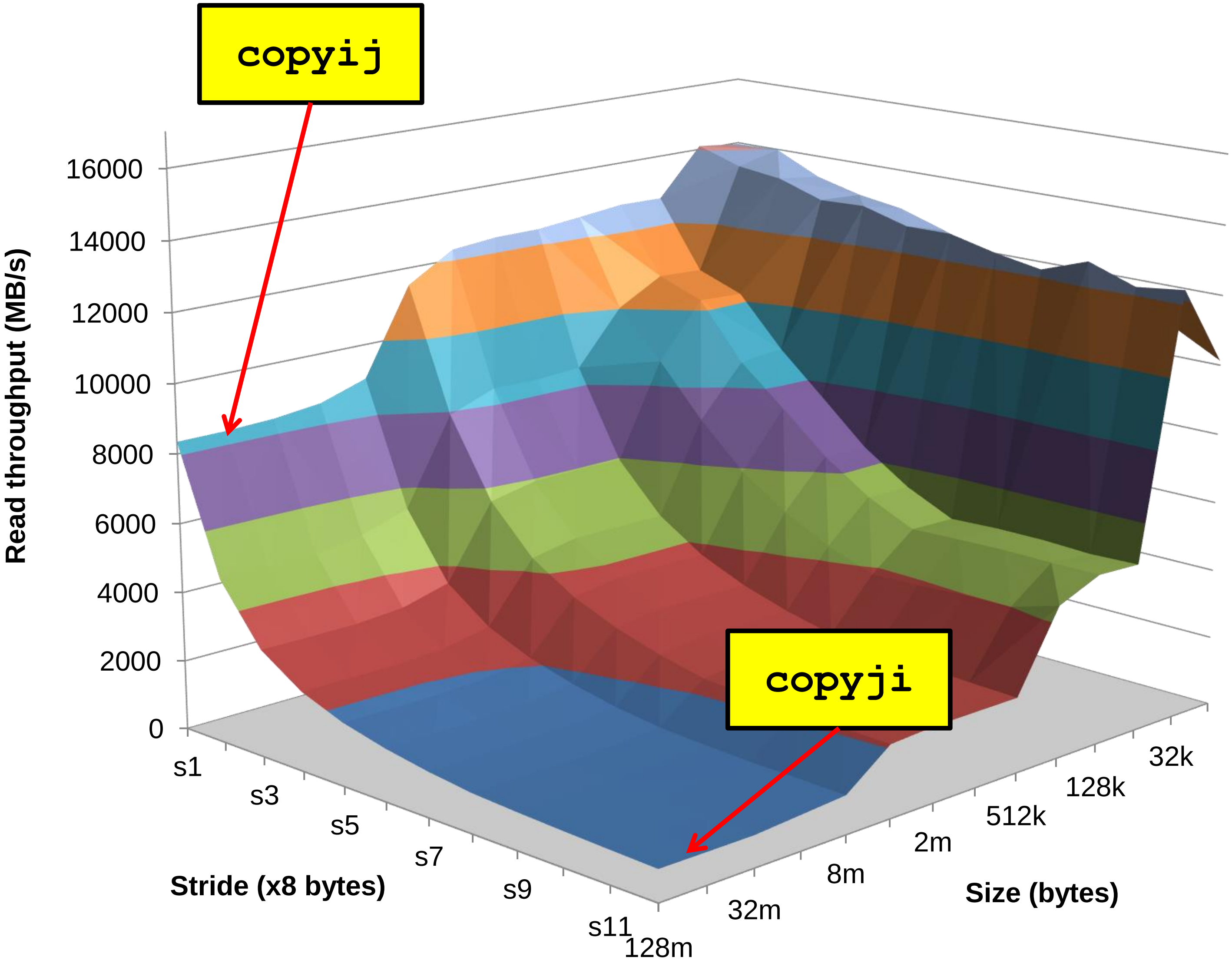
```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
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];
}
```

4.3ms **2.0 GHz Intel Core i7 Haswell** **81.8ms**

- **Hierarchical memory organization**
- **Performance depends on access patterns**
  - Including how step through multi-dimensional array

Why The Performance Differs



Great Reality #5:
Computers do more than execute programs

- **They need to get data in and out**
  - I/O system critical to program reliability and performance

- **They communicate with each other over networks**
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

Amdahl's Law

- **When we improve a part of the system, how does it affect the entire performance?**
  - T_{old} = Original execution time
  - T_{new} = New execution time
  - α = Ratio of the improved part
  - k = Degree of improvement
  - $T_{new} = (1-\alpha)T_{old} + (\alpha T_{old})/k$
  - (Speed up) S = T_{old}/T_{new} = $\frac{1}{(1-\alpha)+\alpha/k}$
  - ex1) improve 60% of the system (α = 0.6) by three times (k = 3)
    - S = 1/[(1-0.6) + 0.6/3] = 1.67
  - ex2) improve 30% of the system (α = 0.3) by twenty times (k = 20)
    - S = 1/[(1-0.3) + 0.3/3] = 1.4
- **Conclusion: Try to improve a major part of the system as possible.**