# Course Introduction

Tom Nurkkala

COS 284

# COS284—Introduction to Computer Systems

# Course Introduction

## Tom Nurkkala

# Motivation

- Abstraction is Good, but Don't Forget Reality
- Most CSE courses emphasize abstraction **Abstract data types, asymptotic analysis
- Abstractions have limits
  - Especially in the presence of bugs
  - Understand details of underlying implementations
- Useful outcomes
  - Become more effective programmers
    - Find and eliminate bugs efficiently
    - Understand and tune for program performance
  - Prepare for later classes

# Programmer-Centric Course

- Purpose
  - Show how …
    by knowing more about the underlying system …
    you can be a more effective programmer
- Enable you to
  - Write programs that are more reliable and efficient
  - Incorporate features that require hooks into OS
    - E.g., concurrency, signal handlers
  - Not just for dedicated hackers
    - Bring out the hidden hacker in everyone!
- Cover material you won't see elsewhere

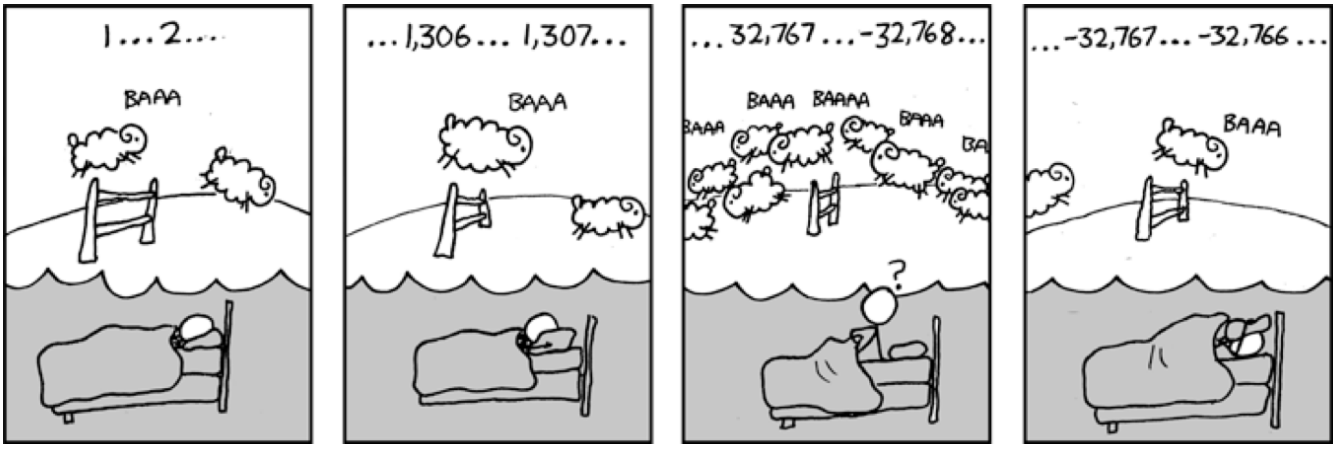
# Five Great Realities

# Reality #1

An `int` isn't an integer, a `float` isn't a real

Example 1: Is $x^2 \ge 0$?

- For a `float`: *yes*
- For an `int`:
    - $40000 \times 40000 = 1600000000$
    - $50000 \times 50000 =$ ??




Example 2: Is $(x + y) + z = x + (y + z)$?

- For `int`: *yes*
- For `float`
    - $(10^{20} + -10^{20}) + 3.14 = 3.14$
    - $10^{20} + (-10^{20} + 3.14) =$ ??

# Code Security Example

*Code Security Example*

```
/* 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 `getpeername`

There are legions of smart people trying to find vulnerabilities in programs

*Friendly Usage*

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

*Code Security Example*

```
/* 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;
}
```

*Malicious Usage*

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    /* Boom */
}
```

*Code Security Example*

```
/* 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;
}
```

# Reality #2

You've got to know assembly

- Probably never write assembly programs
  - Compilers are much better and more patient
  - But key to machine-level execution model
- Behavior of programs in presence of bugs
  - High-level language models break down
- Tuning program performance
  - Understand compiler optimizations
  - Understand 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!

# Reality #3

Memory Matters

- Random Access Memory is an *abstraction*
- Memory is not unbounded
    - It must be allocated and managed
    - Many applications are memory dominated
- Memory referencing bugs 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
- C and C++ do not provide any memory protection
    - Out of bounds array references
    - Invalid pointers
    - Abuses of `malloc` and `free`
- Can lead to nasty bugs
    - 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 or ML
    - Understand what possible interactions may occur
    - Tools to detect referencing errors (e.g. `valgrind`)

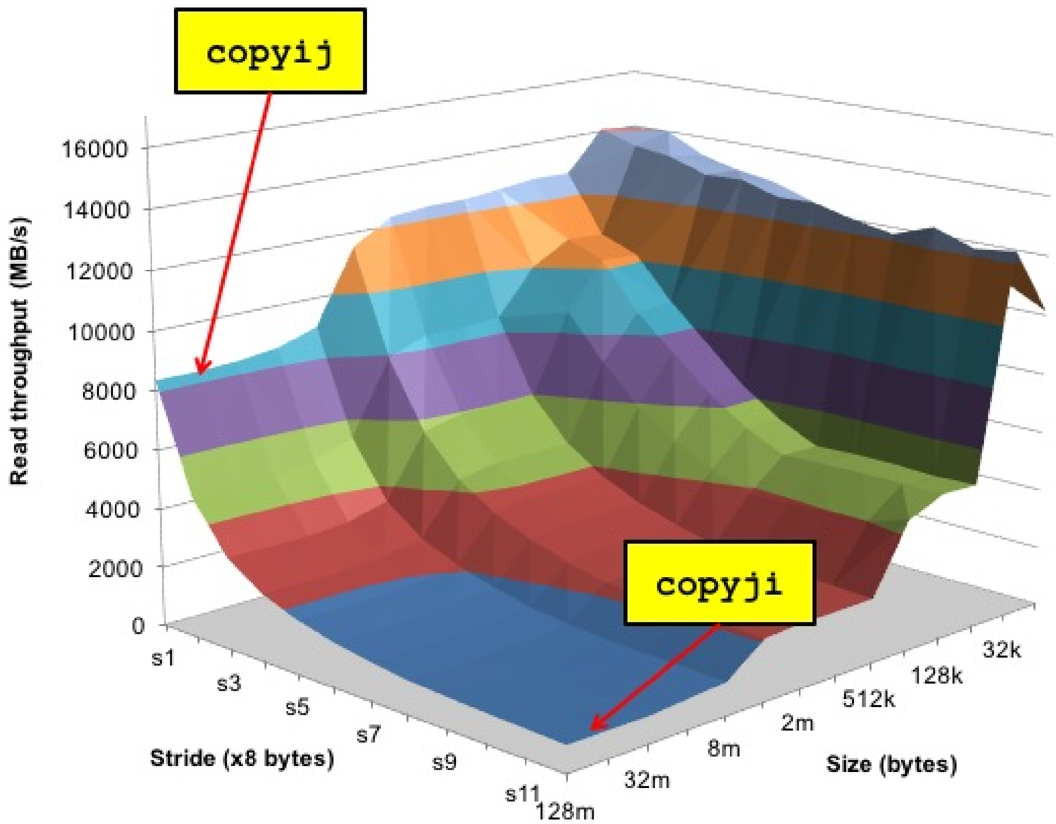
*Runs quickly*

```
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];
}
```

*Runs 21 times slower*

```
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];
}
```

- 21 times slower (Pentium 4)
- Memory is organized hierarchically
- Performance depends on access patterns
  - Including stepping through multi-dimensional array



# Reality #4

Performance is about more than asymptotic complexity

- Constant factors matter too!
- Even exact op count doesn’t predict performance
  - Easily 10:1 performance range based on code written
  - Optimize at multiple levels: algorithm, data representations, procedures, loops
- Must understand system to optimize
  - How programs compiled and executed
  - How to measure performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

# Reality #5

Computers do more than run programs

- Need to get data in and out
  - I/O system critical to reliability and performance
- Communicate with over networks
  - Many system-level issues arise
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues