Course Overview

Overview

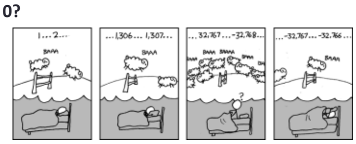
- Course theme
- Five realities
- Computer Systems

Course Theme:
Abstraction Is Good But Don't Forget Reality

- **Most CS courses emphasize abstraction**
  - Abstract data types
  - Asymptotic analysis
- **These abstractions have limits**
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- **Useful outcomes from this course**
  - Become more effective programmers
    - Able to find and eliminate bugs efficiently
    - Able to understand and tune for program performance
  - Prepare for later "systems" classes in CS
    - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems, Storage Systems, etc.

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

- **Example 1: Is $x^2 \geq 0$?**
  - Float's: Yes!



  - Int's:
    - 40000 * 40000 → 1600000000
    - 50000 * 50000 → ??
- **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) --> ??

Source: 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

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!

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
```

- Result is system specific

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:

Critical State	6
?	5
?	4
d7 ... d4	3
d3 ... d0	2
a[1]	1
a[0]	0

$\underbrace{\text{rows 0–3}}_{\texttt{struct\_t}}$

Location accessed by `fun(i)`

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

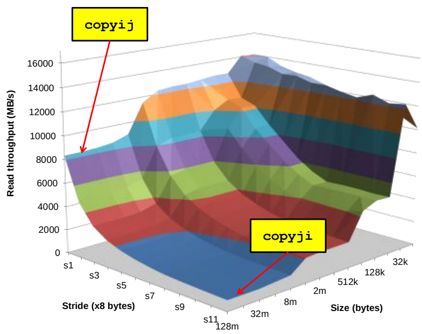
```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
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 to 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

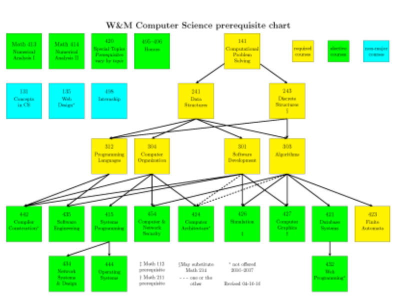
Course Perspective

- **Most Systems Courses are Builder-Centric**
  - Computer Architecture
    - Design pipelined processor in Verilog
  - Operating Systems
    - Implement sample portions of operating system
  - Compilers
    - Write compiler for simple language
  - Networking
    - Implement and simulate network protocols

Course Perspective (Cont.)

- **Our Course is Programmer-Centric**
  - Purpose is to show that by knowing more about the underlying system, one can be more effective as a programmer
  - Enable you to
    - Write programs that are more reliable and efficient
    - Incorporate features that require hooks into OS
      - E.g., concurrency, signal handlers
  - Cover material in this course that you won't see elsewhere
  - Not just a course for dedicated hackers
    - **We bring out the hidden hacker in everyone!**

Role within CS Curriculum



Topics

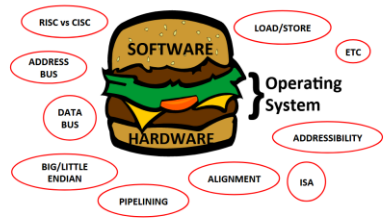
- **Programs and Data**
  - Bits operations, arithmetic, assembly language programs
  - Representation of C control and data structures
  - Includes aspects of architecture and compilers
- **The Memory Hierarchy**
  - Memory technology, memory hierarchy, caches, disks, locality
  - Includes aspects of architecture and OS
- **Exceptional Control Flow**
  - Hardware exceptions, processes, process control, Unix signals, nonlocal jumps
  - Includes aspects of compilers, OS, and architecture

Topics (cont.)

- **Virtual Memory**
  - Virtual memory, address translation, dynamic storage allocation
  - Includes aspects of architecture and OS
- **Networking, and Concurrency**
  - High level and low-level I/O, network programming
  - Internet services, Web servers
  - concurrency, concurrent server design, threads
  - I/O multiplexing with select
  - Includes aspects of networking, OS, and architecture

Computer Systems

- **system: a collection of intertwined hardware and software that must cooperate to achieve the ultimate goal of running application programs (the software) and manages the hardware**



Roles of the Operating System

- **protect the computer from misuse**
- **provide an abstraction for the hardware so that programs can be written across a variety of hardware platforms**
- **manage resources so that multiple users can share the same system**

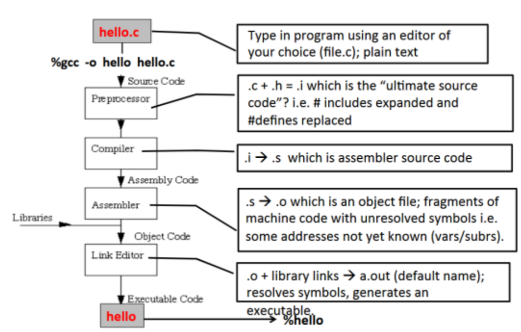
The UNIX Operating System

- **developed in the early 1970s**
- **kernel written in C**
  - C was developed to write UNIX and system programs
- **Linux is a variant of UNIX**
  - other variants: Solaris, OpenBSD, OSX

Software: Text and ASCII Files

- **a file is a sequence of bytes**
  - not a magical container of bytes, but the bytes themselves
- **information in files is interpreted in context**
  - the same sequence of bytes can represent a character, an integer, a float, or an instruction, etc.
- **Chapter 2**

Software: Compilation System



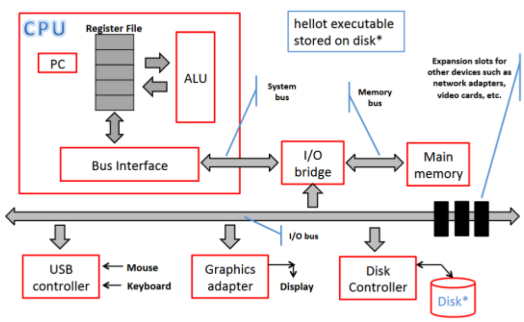
Assembly Language

- **instruction-based execution (Chapters 3, 7)**
  - each program is a sequence of instructions written in machine language
  - processor executes each instruction, one at a time, sequentially
  - convenient to use assembly language rather than machine language
- **you will probably never have to write assembly code**
  - compilers translate high-level code to assembly code for you
  - more patient and (mostly) better than users
- **understanding assembly code is key to machine-level execution model**
  - behavior of program with bugs
  - tuning program performance (with or without help from compiler)
  - implementing system software
  - fighting malware

Assembly Code

- **can use disassembler**
  - tool that shows instruction sequence for executable program
  - UNIX command
    - gcc –o hello hello.c
    - objdump –D –t –s hello
      - -d -- disassemble: display assembler for executable sections
      - -D -- disassemble all
      - -S -- source: intermix source code with assembly
      - -s -- full contents: display full contents of all sections
      - -t -- syms: display contents of symbol table
      - -T -- dynamic syms: display contents of dynamic symbol table

Hardware



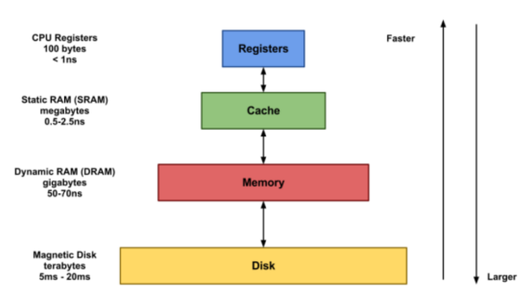
Hardware Organization

- **Processor (CPU)**
  - interprets/executes instructions stored in memory
  - updates the PC to point to the next instruction
  - PC (Program Counter)
    - points at (contains the address of) some machine-level instruction in main memory
  - ALU (Arithmetic Logic Unit)
    - computes new data and address value
  - Register file
    - small storage device containing word-sized registers with their own names
- **Chapter 5**

Hardware Organization

- **I/O Devices**
  - system's connection to the outside world
  - transfers information between I/O bus and I/O devices
- **Main Memory**
  - temporary storage
  - holds both program and data
    - von Neumann architecture
  - linear array of bytes, each with a unique address starting at 0
- **Bus**
  - transfers one "word" at a time
    - fundamental system parameter
    - amount can fetch from memory at one time
    - tends to be the size of the data bus

Memory Hierarchy



Purpose of Memory Hierarchy

- **reduce memory latency**
  - latency is the time (often measured in cycles) between a memory request and its completion
- **maximize memory bandwidth**
  - bandwidth is the amount of useful data that can be retrieved over a certain time interval
- **manage overhead**
  - cost of performing optimization (e.g., copying) should be less than anticipated gain
- **Chapter 6**

Abstraction Provided by the OS

- **Process (Chapter 8)**
  - a running program
  - threads: multiple execution units
  - includes memory and I/O devices (i.e., file abstraction)
- **Virtual Memory (Chapter 9)**
  - provides each process with the illusion that it has exclusive use of the main memory
  - program code and data
    - includes files
    - begins at same fixed address for all processes
    - address space
- **Files (Chapter 10)**
  - sequence of bytes

Address Space

- **An array of 8-bit bytes**
- **A pointer is just an index into this array**

ADDRESS SPACE	Decription/info
Kernel virtual memory	Memory invisible to user code
User stack (created at run time)	Implements function calls
↑ ↓	
Memory mapped region for shared libraries	Ex. printf function
↑	
Run-time heap (created at run time by malloc/ calloc)	Dynamic in size
Read/write data	Program (executable file)
Read-only code and data	Fixed size

32/64 bit starting address

Address 0 →

Notice symbolically drawn with memory "starting" at the bottom