BIL 220 – Introduction to Systems Programming

Spring 2012

Instructors:

Aykut & Erkut Erdem

WE'VE GOT
A NEW
COURSE!
TOTALLY REDESIGNED
INSIDE AND OUT

Acknowledgement: The course slides are adapted from t[illegible]y
R.E. Bryant, D.R. O'Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.

Overview

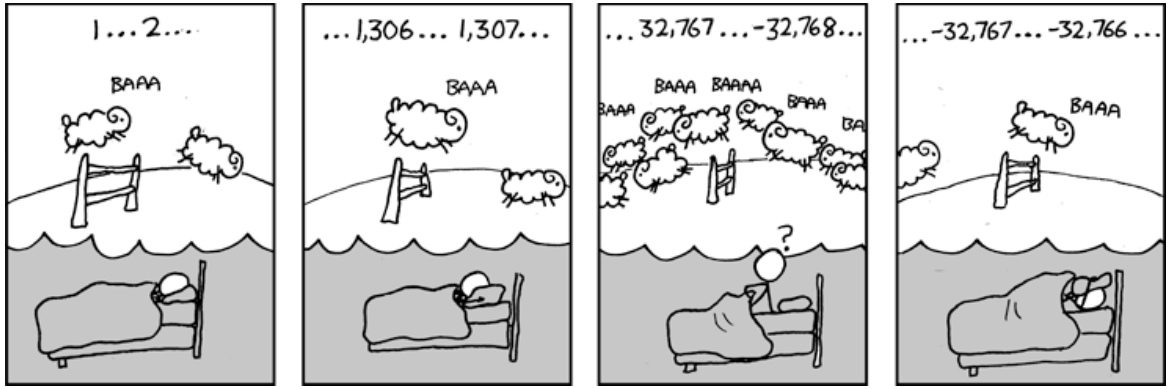
- Course theme
- Five realities
- How the course fits into the our curriculum
- Logistics

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

- Most CENG/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
  - Become more effective programmers
    - Able to find and eliminate bugs efficiently
    - Able to understand and tune for program performance
  - Prepare for "systems" classes
    - Computer Organization, Operating Systems, Computer Networks, Computer Architecture, Microprocessors

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 → ?? -1794967296
- 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) --> ?? 0.00

Source: xkcd.com/571

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* 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

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* 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;
}
```

```
#define MSIZE 528

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

Malicious Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* 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;
}
```

```
#define MSIZE 528

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

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
  - 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

```
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

```
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");
}
```

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

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  →   3.14
fun(1)  →   3.14
fun(2)  →   3.1399998664856
fun(3)  →   2.00000061035156
fun(4)  →   3.14, then segmentation fault
```

volatile keyword is intended to prevent the compiler from applying any optimizations on the code that assume values of variables cannot change "on their own."

- Result is architecture specific

Memory Referencing Bug Example

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  →   3.14
fun(1)  →   3.14
fun(2)  →   3.1399998664856
fun(3)  →   2.00000061035156
fun(4)  →   3.14, then segmentation fault
```

Explanation:

Saved State	4
d7 ... d4	3
d3 ... d0	2
a[1]	1
a[0]	0

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

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];
}
```

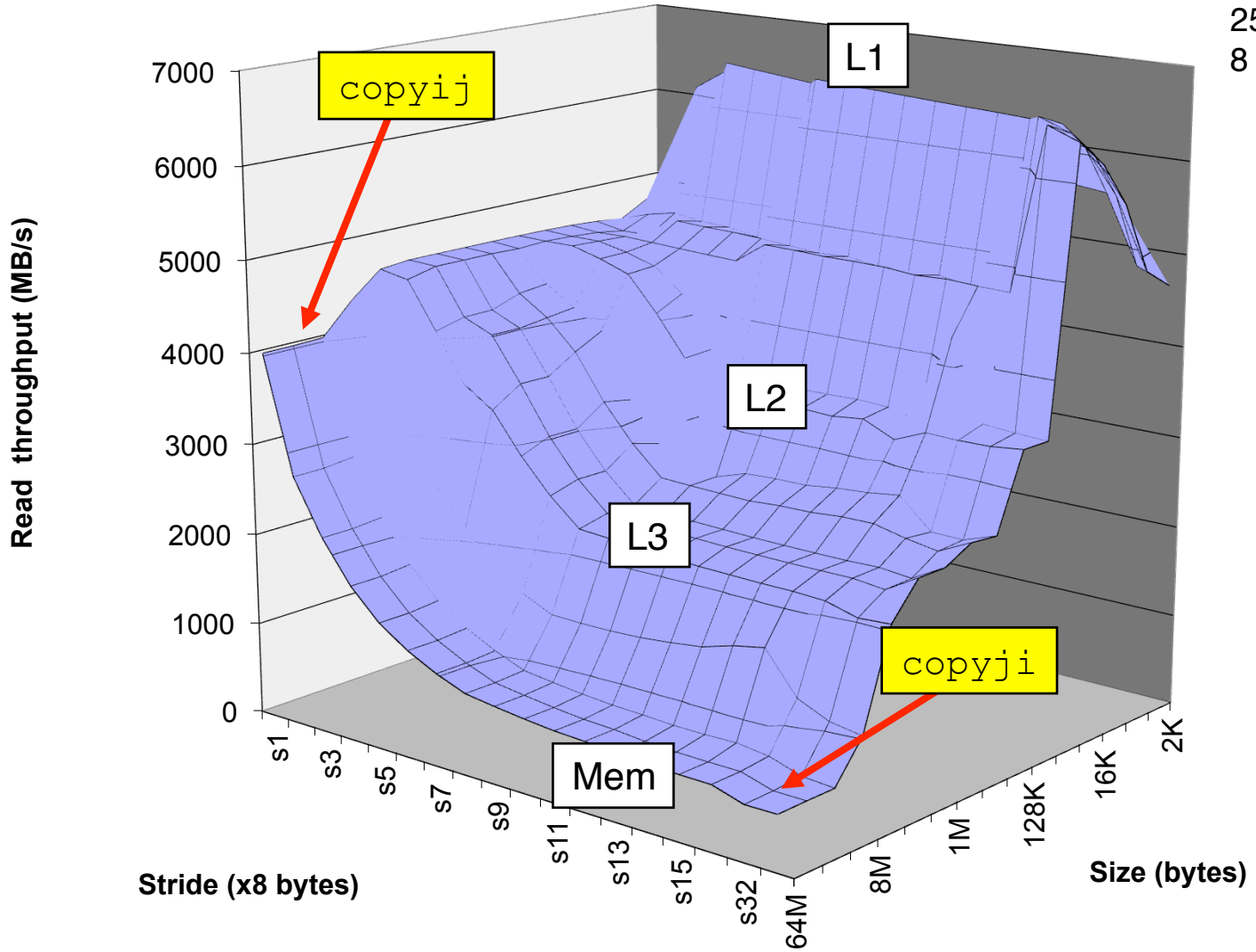
```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

21 times slower (Pentium 4)

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

The Memory Mountain

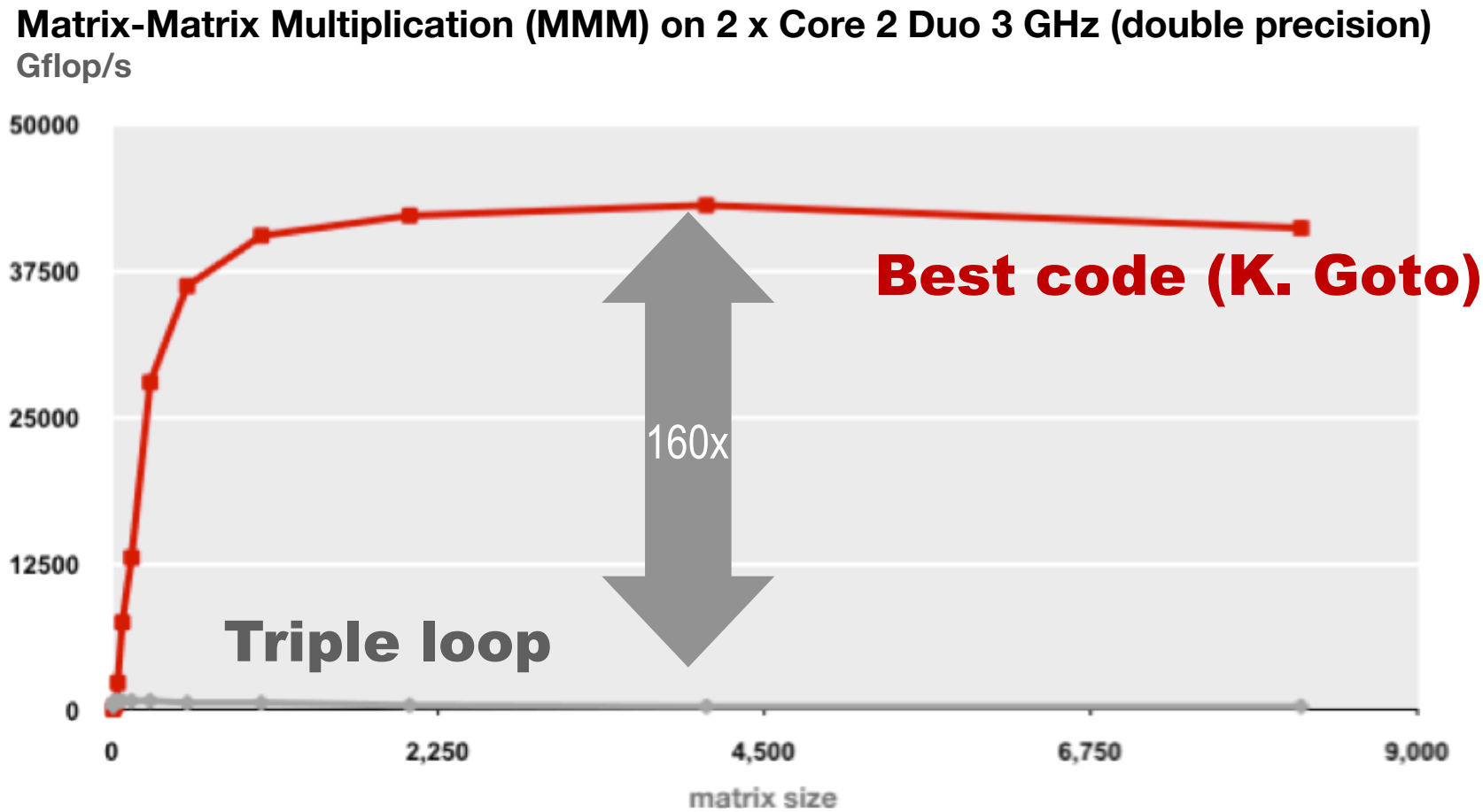
Intel Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache



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

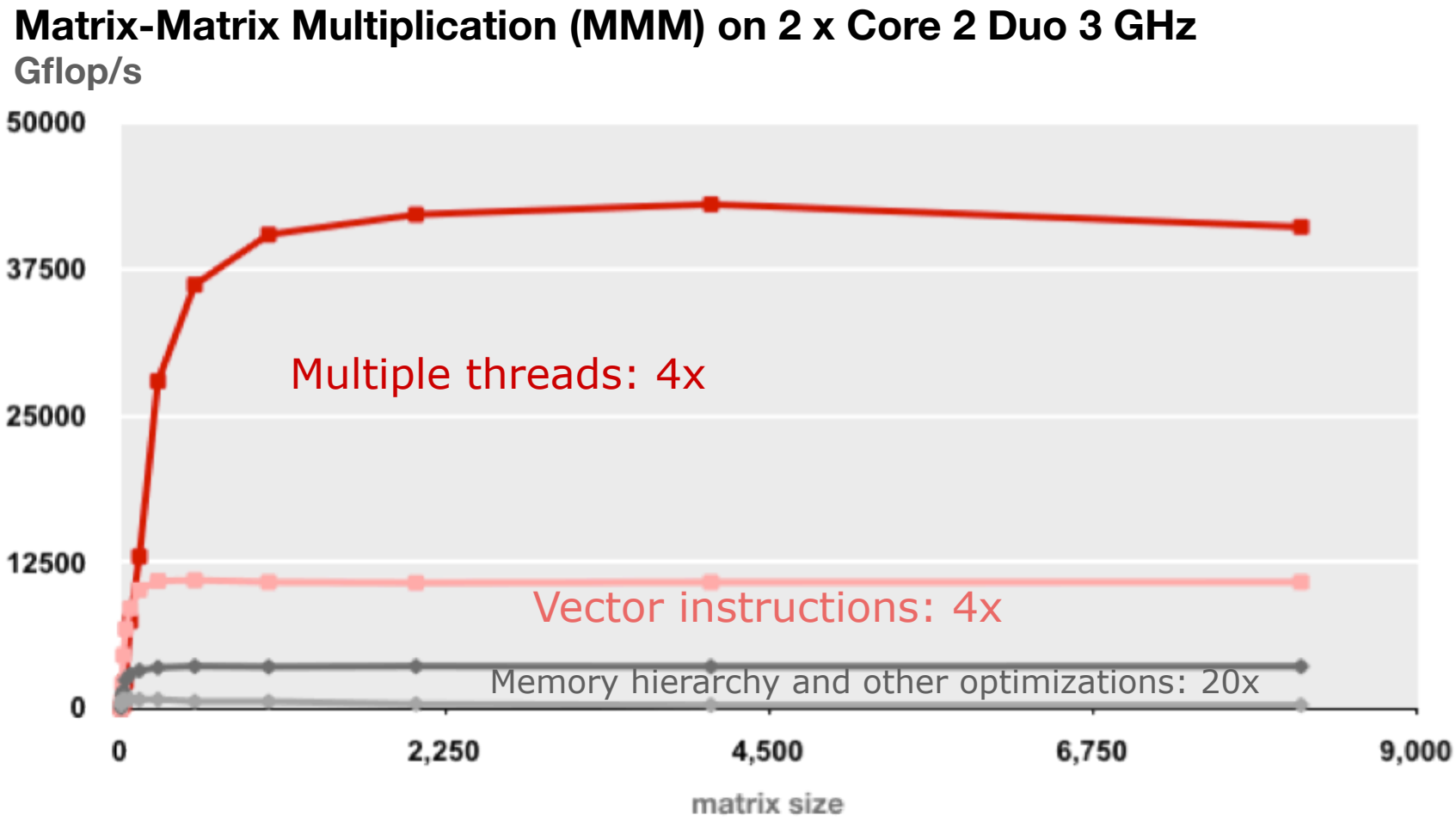
- Constant factors matter too!
- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Example Matrix Multiplication



- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- What is going on?

MMM Plot: Analysis



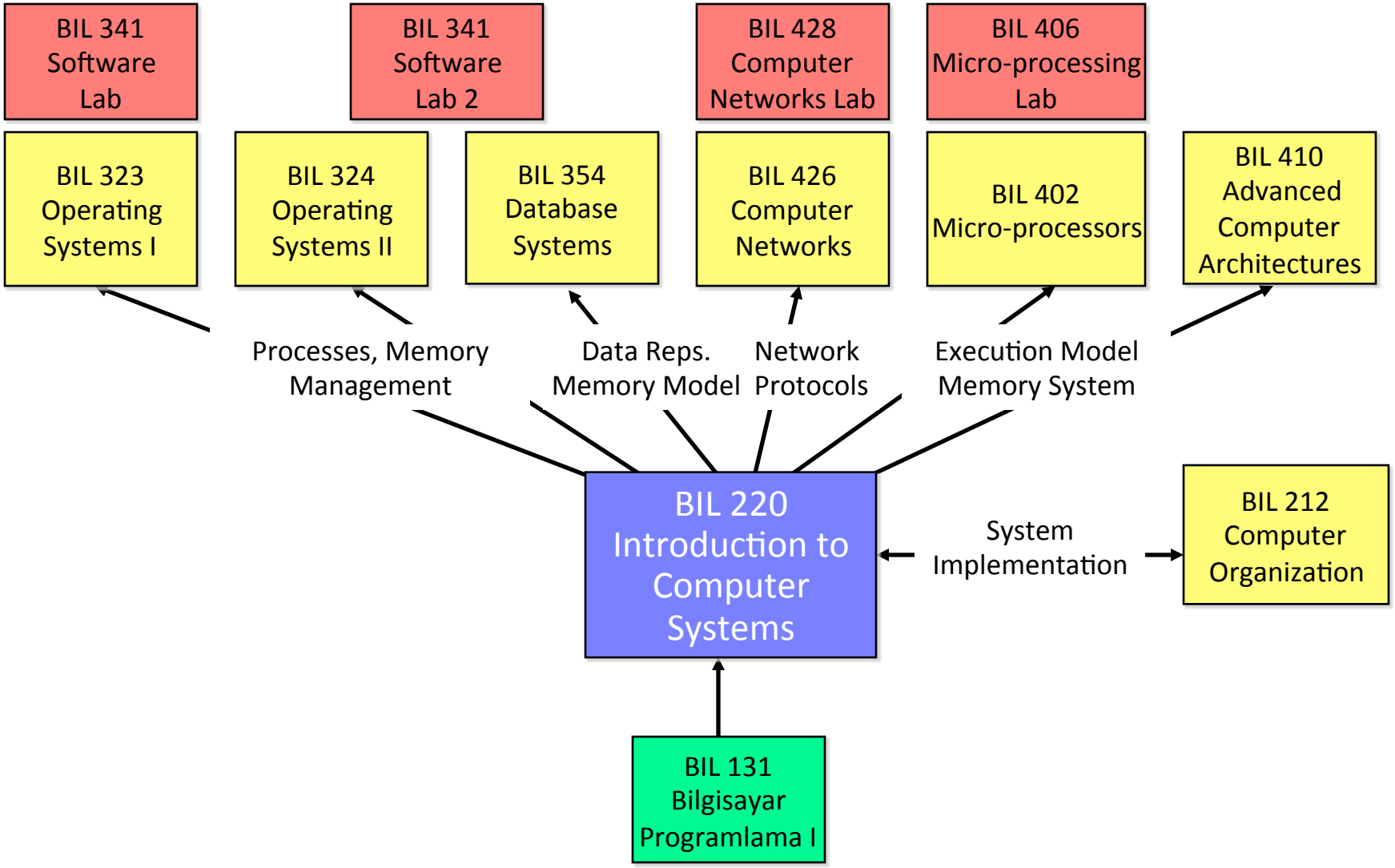
- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- Effect: fewer register spills, L1/L2 cache misses, and TLB misses

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

Role within Hacettepe CENG Curriculum



Course Perspective

- Most Systems Courses are Builder-Centric
  - Computer Architecture
    - Design pipelined processor in Verilog
  - Operating Systems
    - Implement large 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 how 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
  - Not just a course for dedicated hackers
    - We bring out the hidden hacker in everyone
  - Cover material in this course that you won't see elsewhere

Teaching staff

Instructors

Aykut Erdem
aykut@cs.hacettepe.edu.tr
Office: 111
Tel: 297 7500 / 146

Office Hours:
Tuesday, 13:00-15:00

Erkut Erdem
erkut@cs.hacettepe.edu.tr
Office: 114
Tel: 297 7500 / 149

Office Hours:
Friday, 13:00-15:00

Teaching Assistants

Ali Caglayan
alicaglayan@cs.hacettepe.edu.tr
Tel: 297 7500 / 165

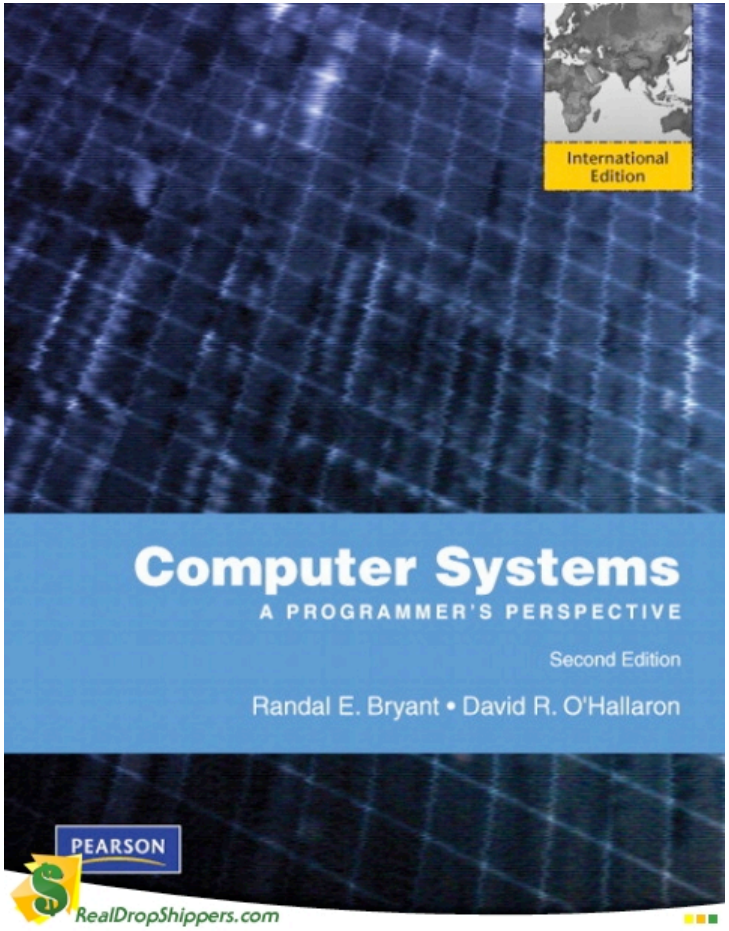
Office Hours:
To be announced..

Oguzhan Guclu
oguzhanguclu@cs.hacettepe.edu.tr
Tel: 297 7500 / 149

Office Hours:
To be announced..

Textbook

- Randal E. Bryant and David R. O'Hallaron,
  - "Computer Systems: A Programmer's Perspective, Second Edition" (CS:APP2e), Prentice Hall, 2011
  - http://csapp.cs.cmu.edu
- **This book really matters for the course!**
  - How to solve labs
  - Practice problems typical of exam problems



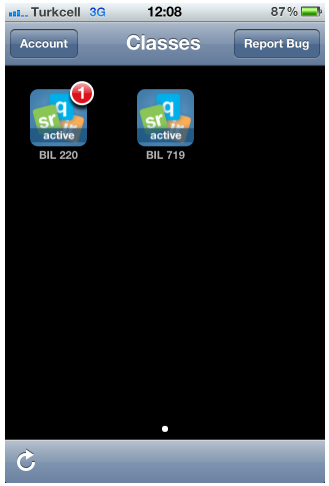
Course Components

- Lectures
  - Higher level concepts
- Recitations
  - Applied concepts, important tools and skills for labs, clarification of lectures, exam coverage
- Programming Assignments (4)
  - The heart of the course
  - 2-3 weeks each
  - Provide in-depth understanding of an aspect of systems
  - Programming and measurement
- Exams (2 midterms + final)
  - Test your understanding of concepts & mathematical principles

Getting Help

- Couse web page:
  - `http://web.cs.hacettepe.edu.tr/~bil220`
  - Complete schedule of lectures, exams, and assignments
  - Lecture notes and assignments
- Communication
  - Announcements and course related discussions
  - Through piazza:

    `https://www.piazza.com/hacettepe.edu.tr/spring2012/bil220`



Policies: Grading

- Midterm 1 (15%) – 29 March 2012
- Midterm 2 (15%) – 26 April 2012
- Final (40%) – *To be announced*
- 4 Programming Assignments (25%)
- Class participation (5%)

Course Overview (1)

- Programs and Data
  - Bits operations, arithmetic, assembly language programs
  - Representation of C control and data structures
  - Includes aspects of architecture and compilers
- Performance
  - Co-optimization (control and data), measuring time on a computer
  - Includes aspects of architecture, compilers, and OS
- The Memory Hierarchy
  - Memory technology, memory hierarchy, caches, disks, locality
  - Includes aspects of architecture and OS

Course Overview (2)

- Linking
  - Ideas from static and dynamic link
  - Includes aspects of architecture and compilers
- Exceptional Control Flow
  - Hardware exceptions, processes, process control, Unix signals, non-local jumps
  - Includes aspects of compilers, OS, and architecture
- Virtual Memory
  - Virtual memory, address translation, dynamic storage allocation
  - Includes aspects of architecture and OS
- System Level I/O
  - Basic concepts of Unix I/O such as files and descriptors
  - Includes aspects of compilers and OS

Assignments

- PA1 (datalab): Manipulating bits
- PA2 (bomblab): Defusing a binary bomb
- PA3 (bufferlab): Hacking a buffer bomb
- PA4 (shelllab): Writing your own Unix shell

You will develop all your programming assignments on a machine running Linux OS!

Assignment Rationale

- Each assignment has a well-defined goal such as solving a puzzle or winning a contest
- Doing the lab should result in new skills and concepts
- We try to use competition in a fun and healthy way
  - Set a reasonable threshold for full credit
  - Post intermediate results (anonymized) on Web page for glory!

Policies: Assignments

- Work groups
  - You must work alone on all assignments stated unless otherwise
- Submission
  - Assignments due at 23:59 on Wednesday evening
  - Electronic submissions (no exceptions!)
- Lateness penalties
  - Get penalized **20% per day**
  - No late submission later than **3 days after due date**

Cheating

- What is cheating?
  - Sharing code: by copying, retyping, looking at, or supplying a file
  - Coaching: helping your friend to write a lab, line by line
  - Copying code from previous course or from elsewhere on WWW
    - Only allowed to use code we supply, or from CS:APP website
- What is NOT cheating?
  - Explaining how to use systems or tools
  - Helping others with high-level design issues
- Penalty for cheating:
  - Removal from course with failing grade
- Detection of cheating:
  - We do check
  - Our tools for doing this are much better than most cheaters think!

Always keep in mind

- This is a **MUST** course.
- In order to get the most out of the course, **try to stay ahead**.
  - By the weekend, make sure you have reviewed the material covered in the lectures of the preceding week.
- If you have any questions or just want to chat about the course, do not hesitate to **go to office hours**.
- **Nothing is easy, but don't give up!**

We aim high!

Photo courtesy of Flickr user richardlamprecht

- Our goal is to make BIL 220 one of the **highly respected** and one of the most **enjoyable** courses in the department!

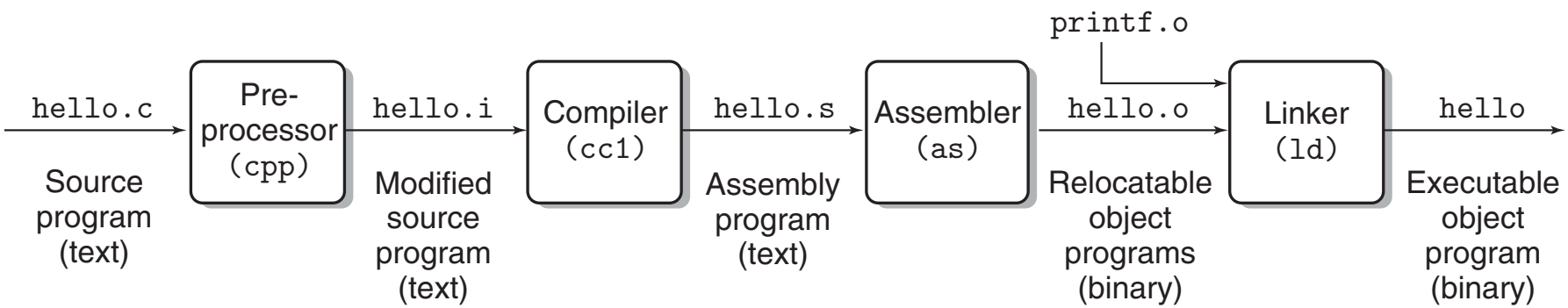
WELCOME
to BIL 220!

Hello World!

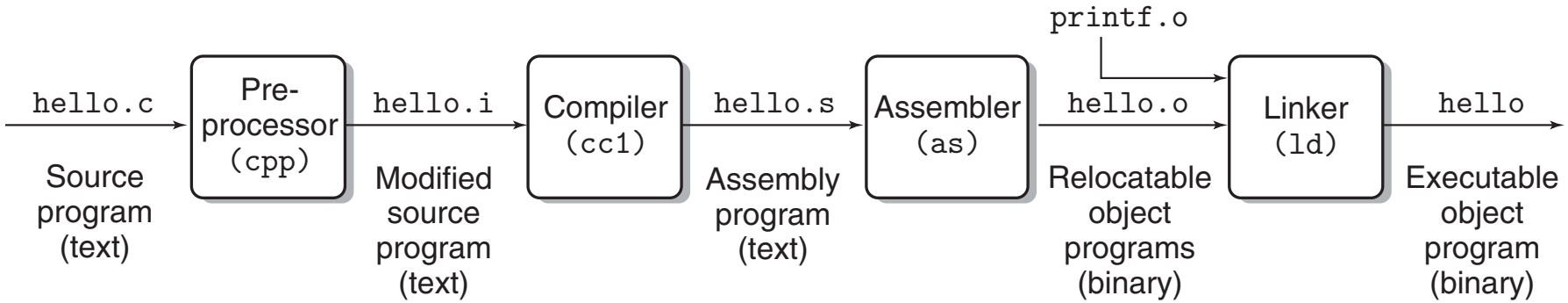
```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

```
unix> gcc -o hello hello.c
```

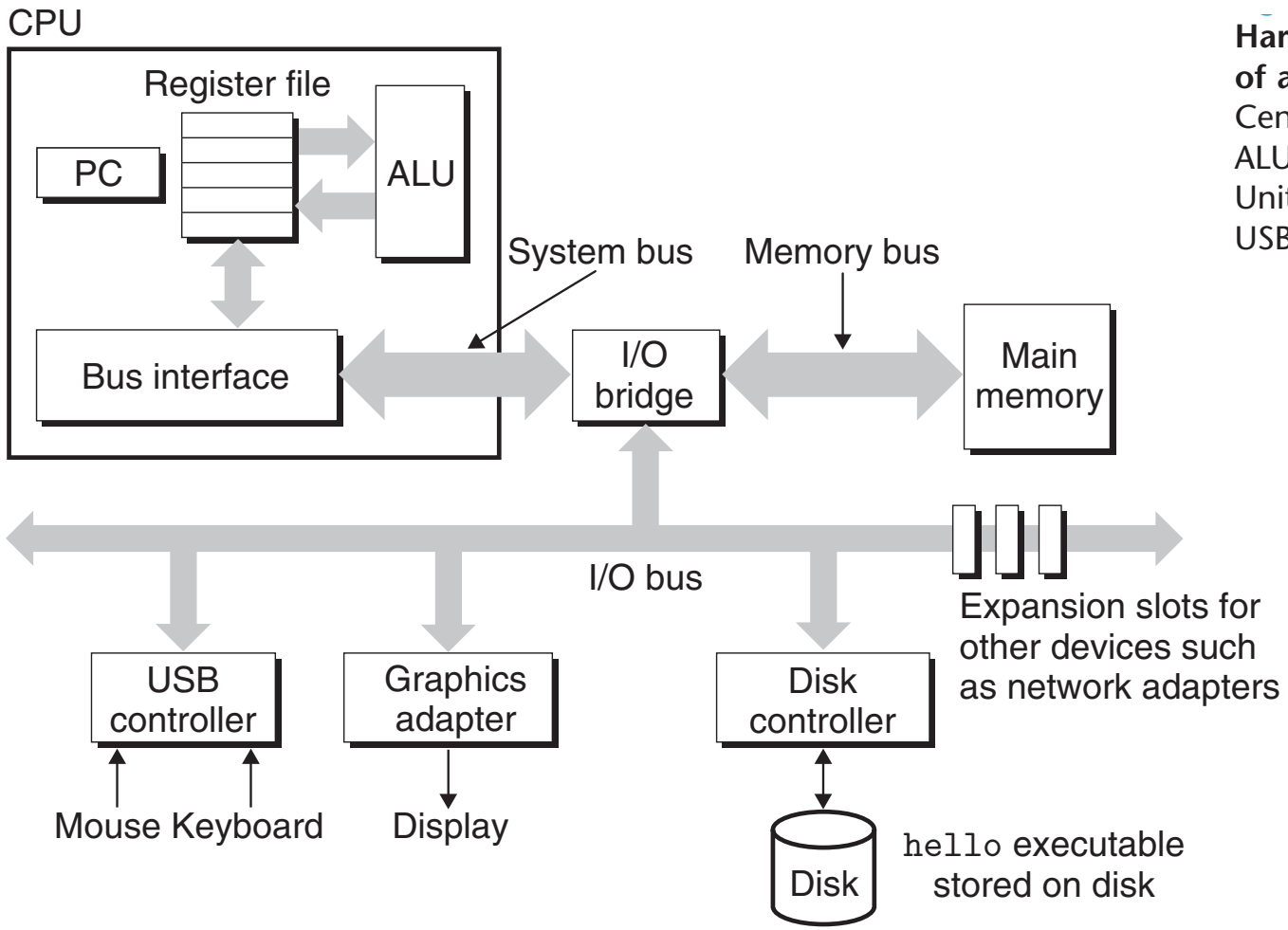


Understanding How Compilation Systems Work is Really Matters!



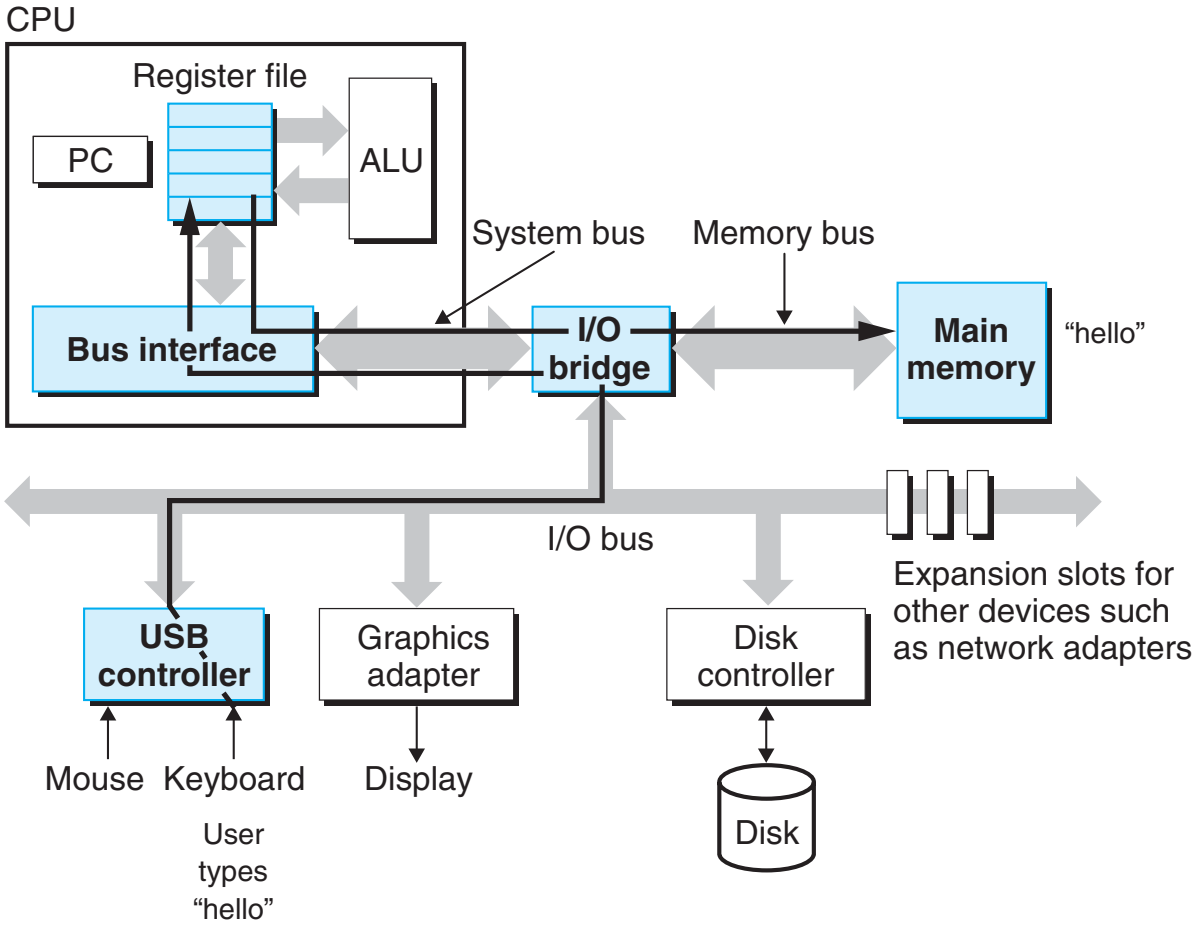
- Optimizing program performance
  - Understanding the basics of machine code and how the compiler translates different C statements into machine code is a must for better C programming
- Understanding link-time errors
  - Some of the most of the puzzling programming errors are related to linking
- Avoiding security holes
  - Understanding the consequences of the way data and control information are stored on the program stack is crucial for secure programming

Hardware Organization of a System



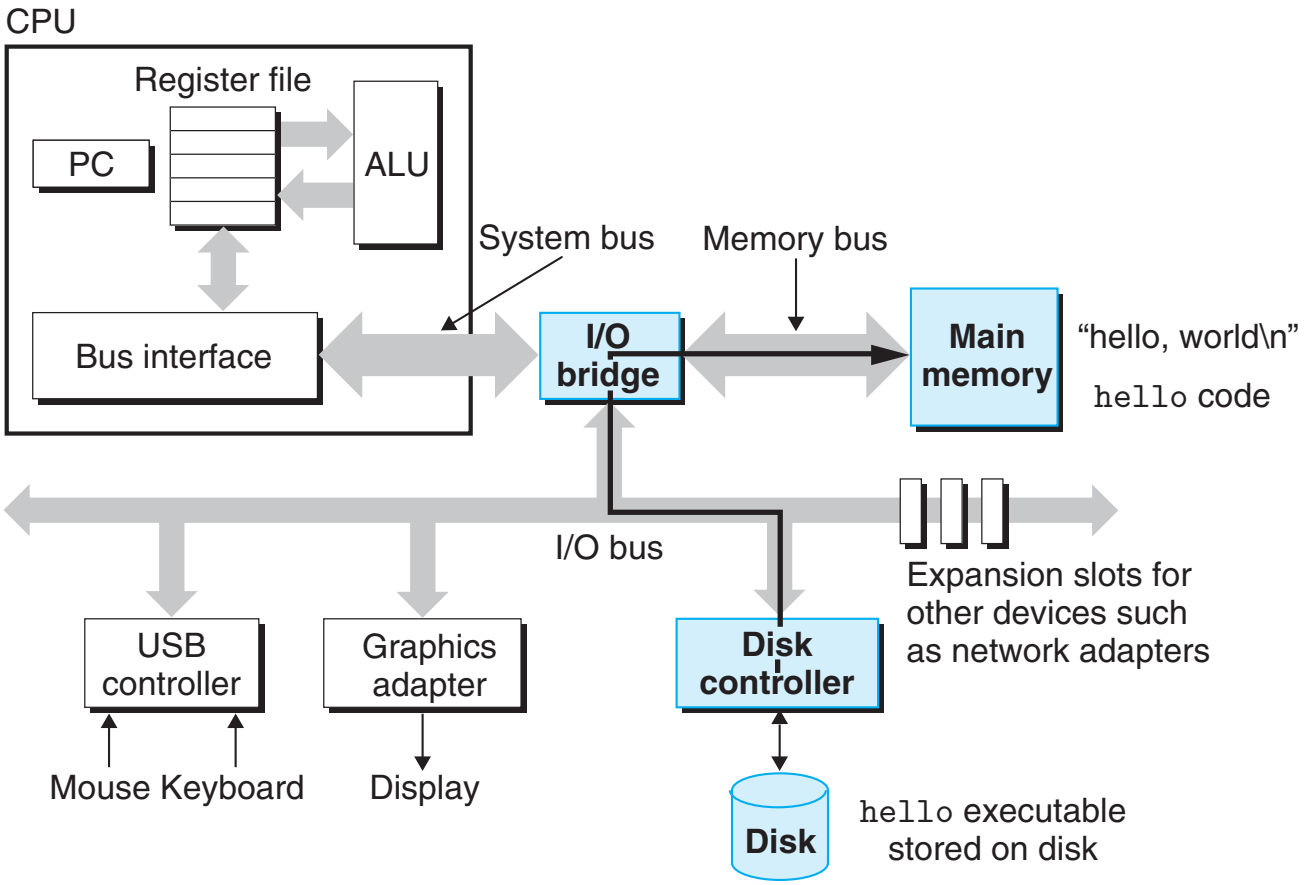
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

Running the `hello` Program



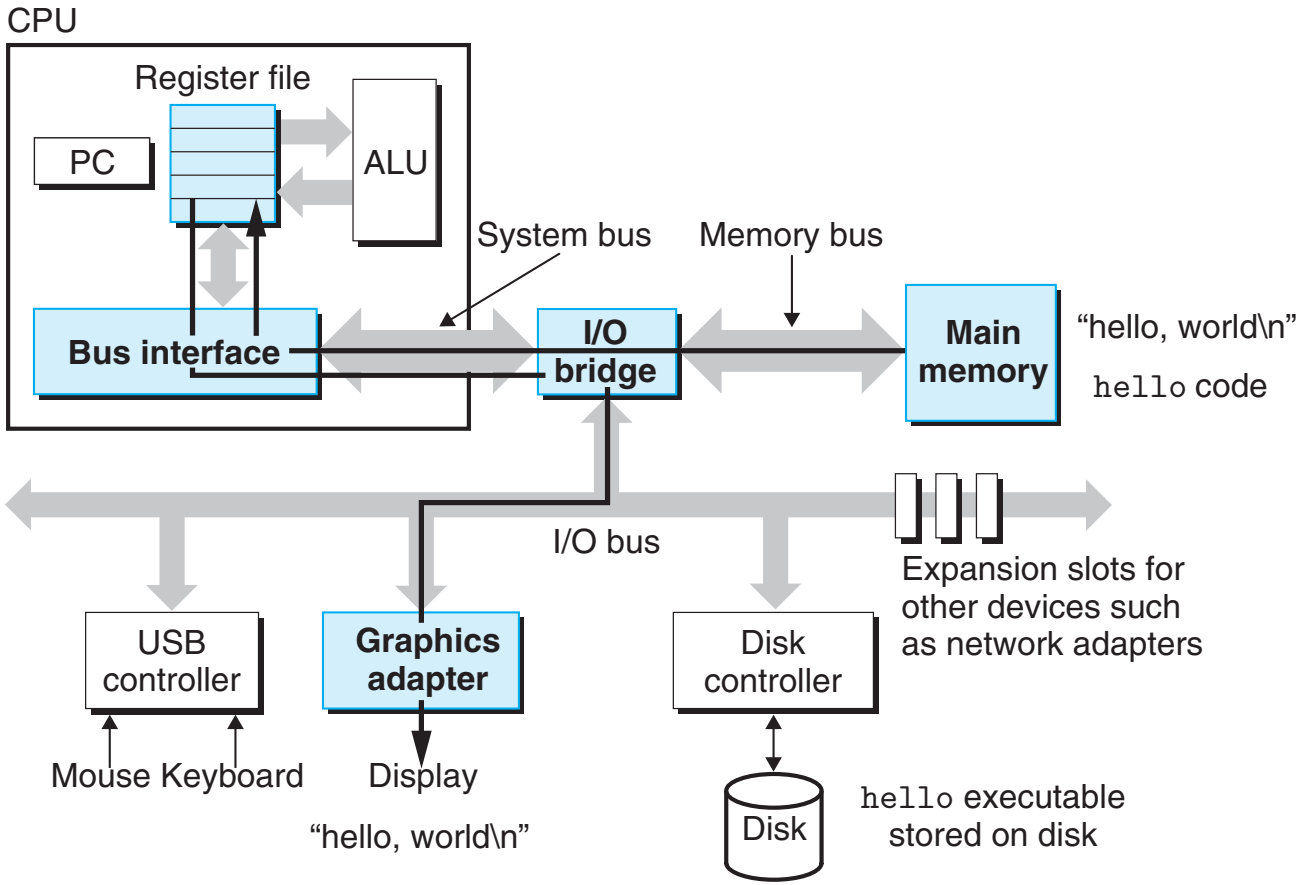
Reading the `hello` command from the keyboard.

Running the `hello` Program



Loading the executable from disk into main memory.

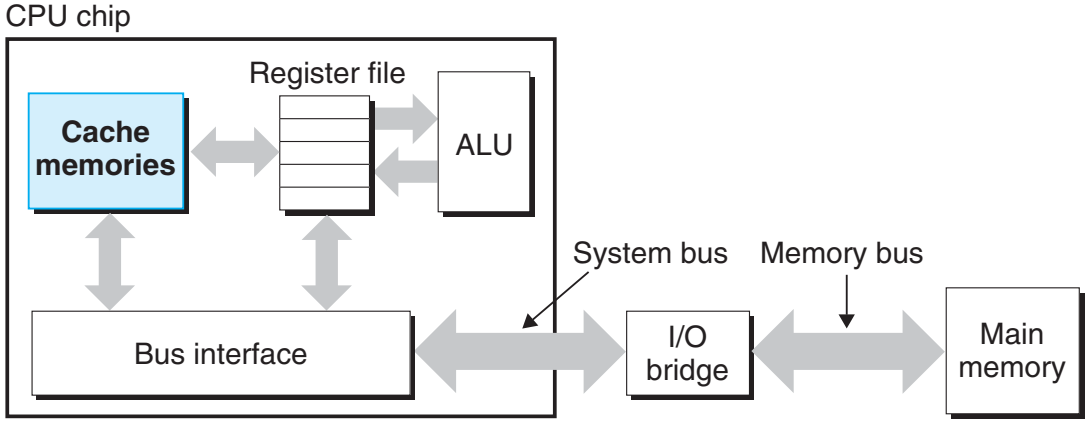
Running the `hello` Program



Writing the output string from memory to the display.

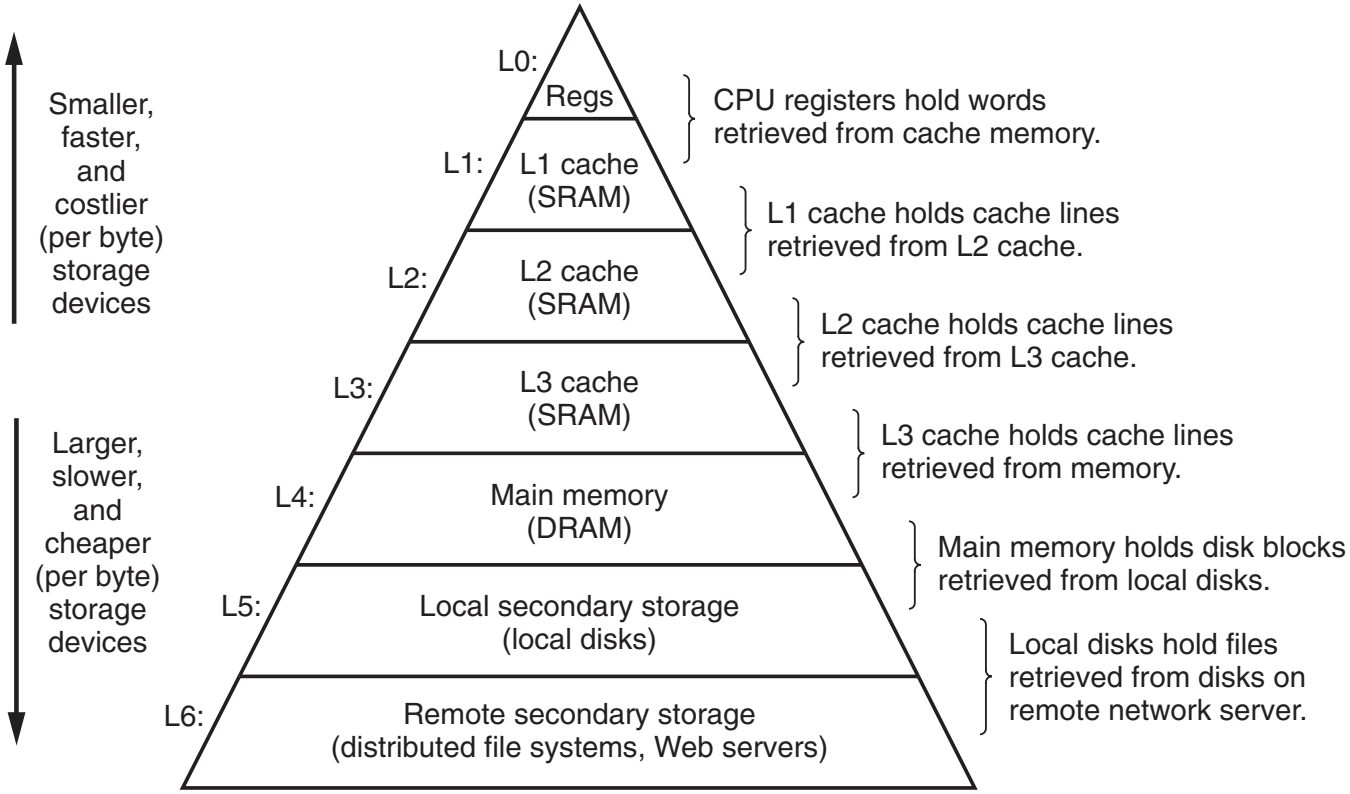
Caches Matter

- A system spends a lot of time moving information from one place to another.
- Larger storage devices are slower than smaller storage devices.
  - e.g., the processor can read data from the register file almost 100 times faster than from main memory.
- Faster devices are more expensive to build than their slower counterparts.
- Cache memories as temporary staging areas



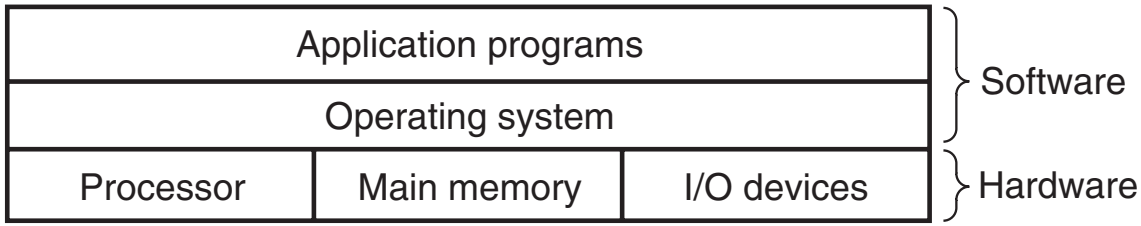
The Memory Hierarchy

- Storage devices form a hierarchy
- Storage at one level serves as a cache for storage at the next lower level.



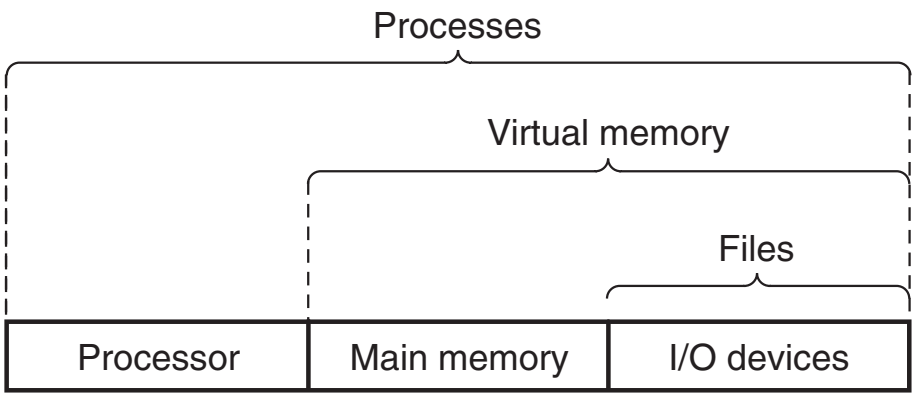
The Operating System Manages the Hardware

- The operating system as a layer of software interposed between the application program and the hardware
- All attempts by an application program to manipulate the hardware must go through the operating system.
- The operating system has two primary purposes:
  1. To protect the hardware from misuse by runaway applications, and
  2. To provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices.

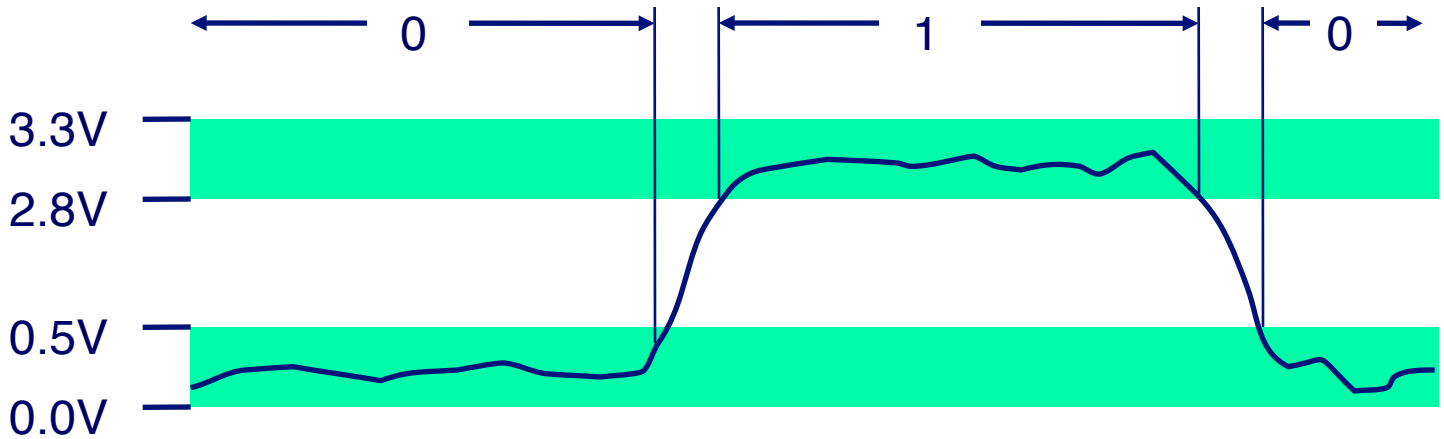


Abstractions provided by an OS

- The operating system achieves both goals via the fundamental abstractions”
  - **Processes:** The operating system’s abstraction for running programs.
  - **Virtual Memory:** An abstraction that provides each process with the illusion that it has exclusive use of the main memory.
  - **Files:** Abstractions for I/O devices



Binary Representations

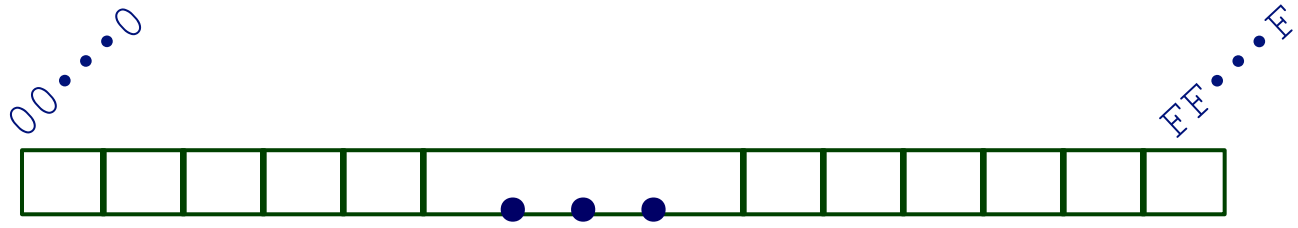


Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - 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
      - 0xfa1d37b

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Byte-Oriented Memory Organization



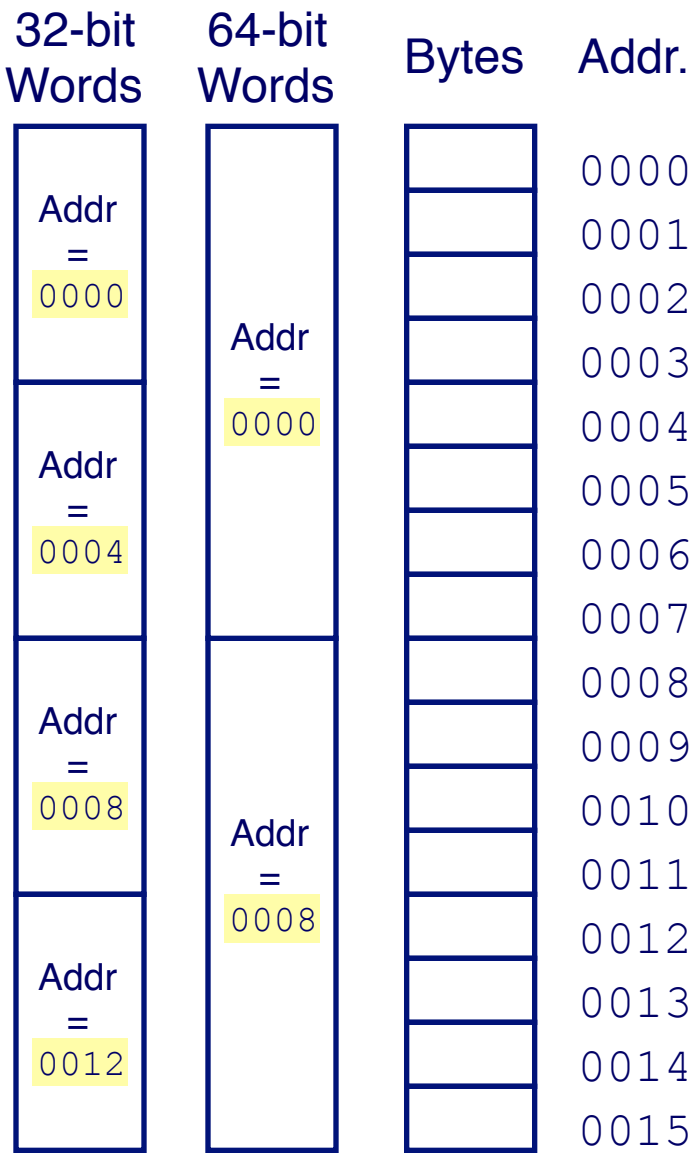
- Programs Refer to Virtual Addresses
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular "process"
    - Program being executed
    - Program can clobber its own data, but not that of others
- Compiler + Run-Time System Control Allocation
  - Where different program objects should be stored
  - All allocation within single virtual address space

Machine Words

- Machine Has "Word Size"
  - Nominal size of integer-valued data
    - Including addresses
  - Most current machines use 32 bits (4 bytes) words
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - High-end systems use 64 bits (8 bytes) words
    - Potential address space ≈ 1.8×10^{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 Organization

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



Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

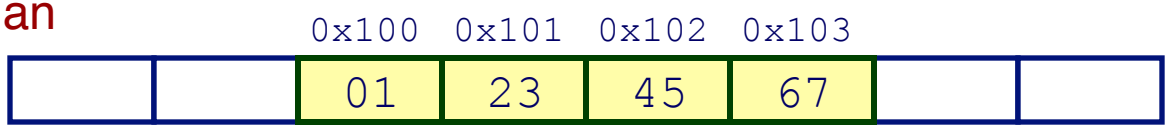
Byte Ordering

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

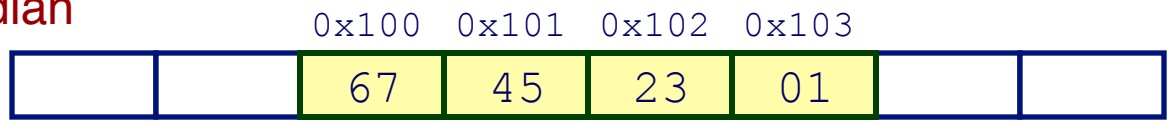
Byte Ordering Example

- 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

Big Endian



Little Endian



The origin of "endian"

"Gulliver finds out that there is a law, proclaimed by the grandfather of the present ruler, requiring all citizens of Lilliput to break their eggs only at the little ends. Of course, all those citizens who broke their eggs at the big ends were angered by the proclamation. Civil war broke out between the Little-Endians and the Big-Endians, resulting in the Big-Endians taking refuge on a nearby island, the kingdom of Blefuscu."

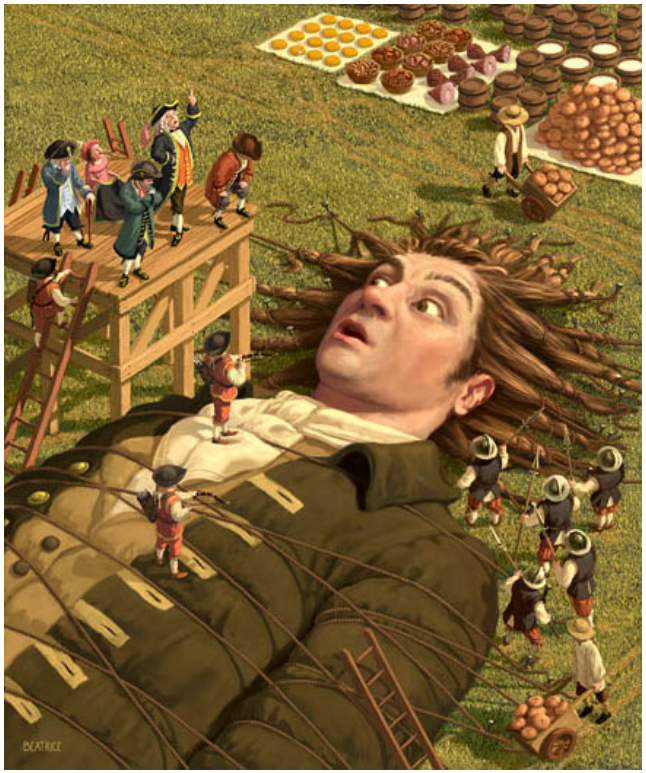


Illustration by Chris Beatrice

– Danny Cohen, *On Holy Wars and A Plea For Peace*, 1980

Reading Byte-Reversed Listings

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

Address	Instruction Code	Assembly Rendition	
8048365:	5b	pop	%ebx
8048366:	81 c3 ab 12 00 00	add	$0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl	$0x0,0x28(%ebx)

- Deciphering Numbers
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00

Examining Data Representations

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * creates byte array

```
typedef unsigned char *pointer;

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

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

show_bytes Execution Example

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

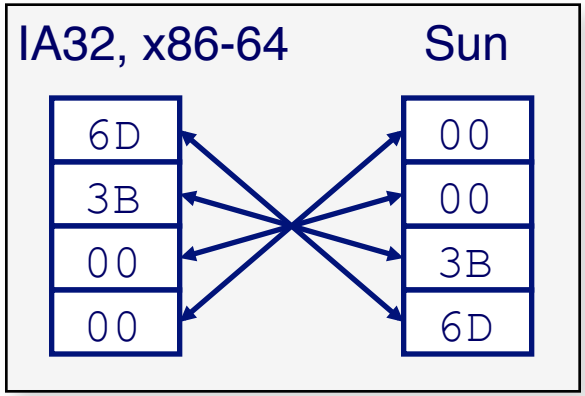
Result (Linux):

```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

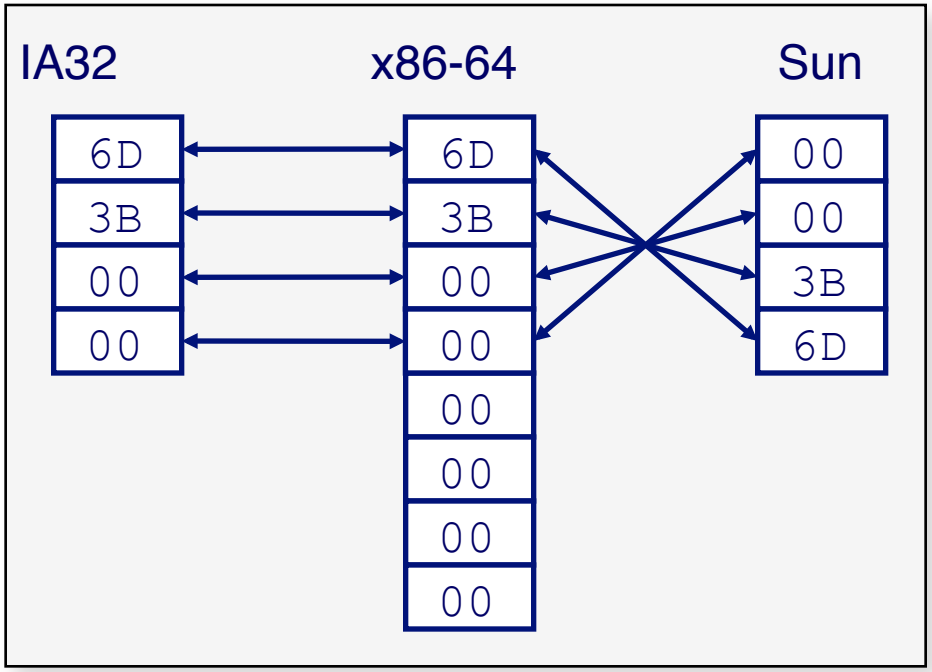
Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

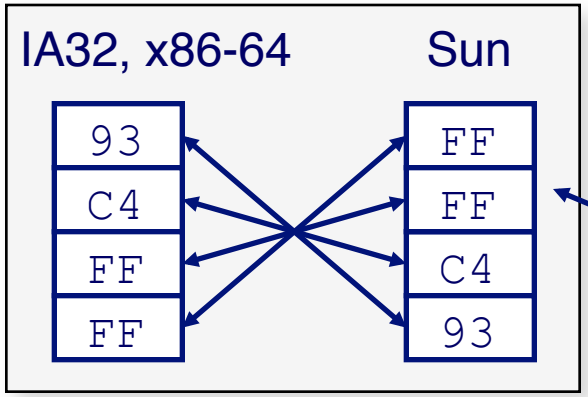
```
int A = 15213;
```




```
long int C = 15213;
```




```
int B = -15213;
```



Two's complement representation
(Covered later)

Representing Pointers

```
int B = -15213;
int *P = &B;
```

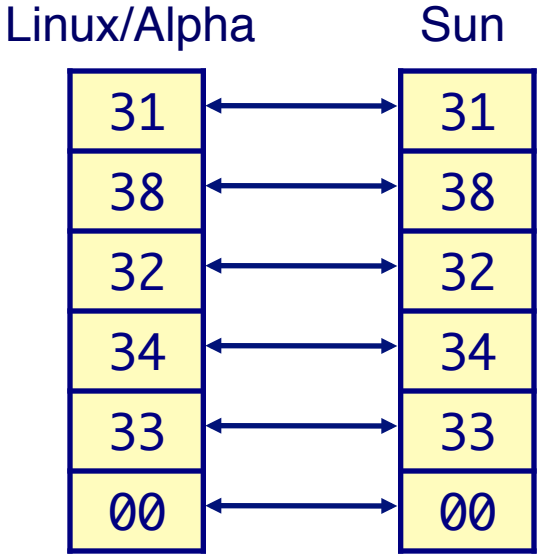
Sun	IA32	x86-64
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

Different compilers & machines assign different locations to objects

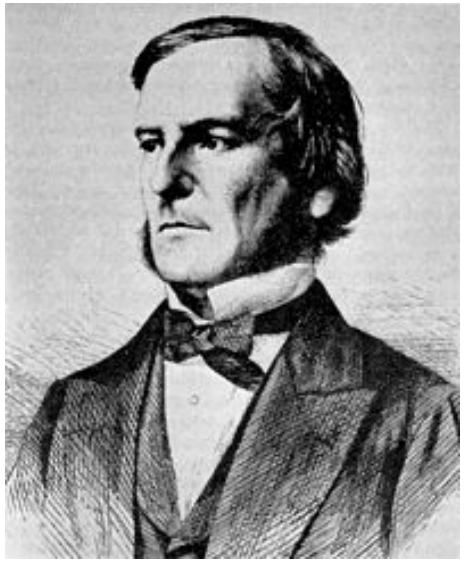
Representing Strings

```
char S[6] = "18243";
```

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit i has code 0x30+i
  - String should be null-terminated
    - Final character = 0
- Compatibility
  - Byte ordering not an issue



Boolean Algebra



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

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

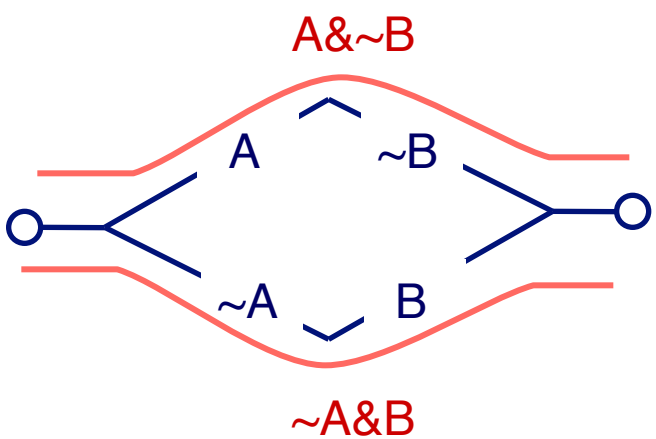
Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	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



Connection when

A&~B | ~A&B

= A^B

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- All of the Properties of Boolean Algebra Apply

Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of {0, …, w–1}
  - aj = 1 if j ∈ A
    - 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- Operations

&	Intersection	01000001	{ 0, 6 }
|	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
~	Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- Operations `&`, `|`, `~`, `^` Available in C
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - `~0x41 → 0xBE`
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - `~0x00 → 0xFF`
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55 → 0x41`
    - $01101001_2 \ \& \ 01010101_2 \rightarrow 01000001_2$
  - `0x69 | 0x55 → 0x7D`
    - $01101001_2 \ | \ 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - `&&, ||, !`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- Examples (char data type)
  - `!0x41 → 0x00`
  - `!0x00 → 0x01`
  - `!!0x41 → 0x01`

  - `0x69 && 0x55 → 0x01`
  - `0x69 || 0x55 → 0x01`
  - `p && *p` (avoids null pointer access)

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector `x` left `y` positions
    - Throw away extra bits on left
  - Fill with `0`'s on right
- Right Shift: `x >> y`
  - Shift bit-vector `x` right `y` positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with `0`'s on left
  - Arithmetic shift
    - Replicate most significant bit on right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument `x`	`01100010`
`<< 3`	`00010000`
Log. `>> 2`	`00011000`
Arith. `>> 2`	`00011000`

Argument `x`	`10100010`
`<< 3`	`00010000`
Log. `>> 2`	`00101000`
Arith. `>> 2`	`11101000`

The Strange Birth and Long Life of Unix

Photo: Alcatel-Lucent

- http://spectrum.ieee.org/computing/software/the-strange-birth-and-long-life-of-unix