Computer Systems I

Class 1

Adiministriva

- Instructor: Prof. Will Marrero
- Office hours: Tues & Thurs. 5:00 – 5:45 CST 737
- email: wmarrero@cs.depaul.edu
- Syllabus
- Cheating

Resources

- **Linux account:**
  - Class server is: wmarrero2.cstcis.cti.depaul.edu
  - Username is first initial followed by first 7 letters of your last name
  - Password is your 7 digit DePaul ID number
  - CHANGE YOUR PASSWORD IMMEDIATELY WITH passwd
  - Connect using some form of SSH
- **Getting SSH:**
  - Get PuTTY and PSCP from http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
  - Google "SSHSecureShellClient-3.2.9.exe" to find an old copy

Systems I

C and Unix

- We will be using C and Unix in this course.
- We will cover topics as we need them.
- For now, make sure:
  - You can log into wmarrero2.cstcis.cti.depaul.edu
  - You can write a C program with Emacs (or some other editor on the server)
  - You can compile a simple C program
  - You can execute a simple C program
- Look Content on D2L for short introductions to UNIX, emacs, and gdb.
- Very brief demo using PuTTY.

Systems I

Getting Started

- Linux has plenty of documentation.
  - Use the man command.
    - man man
    - man gcc
    - man gdb
    - man –k *<keyword>*
  - You can usually "kill" the current process with ctrl-C
  - You can usually suspend the current process with ctrl-Z
  - You can return to a suspended process with fg
- Emacs has plenty of documentation.
  - Start with ctrl-H t
  - You can always exit Emacs with ctrl-X ctrl-C
  - Be careful about "backspace" key and "delete" key.

Systems I

Comparing C and Java

- Both are compiled and then executed
- object code vs. byte code
- .h files vs. interfaces (.java files)
- .c files vs. classes (.java files)
- .o files vs. .class files
- compiler and VM vs. compiler, assembler, and linker
- struct vs. class
- Strings vs. char[ ]

Purpose of the course

- Knowing more about the underlying system will make you a better programmer.
- Enable you to write programs that have fewer errors and run faster.
- Give you some practice with reverse engineering.
- Give you some practice with performance tuning.

Coursework

- Most systems courses are "builder-centric" (i.e. learn by building)
  - OS course: implement parts of an OS
  - compilers: implement a simple compiler
  - hardware: design part of a processor
- This course is "experiment-centric" (i.e. learn by fiddling)
- This course will have labs that require you to experiment and stumble your way through them.

Disclaimer

- This course will be a lot of work.
- You will get your hands dirty.
- This course can be fun.
- Labs provide
  - instant feedback.
  - Friendly competition
- What is a hacker?
  - http://www.catb.org/jargon/html/H/hacker.html
- Get in touch with your inner hacker.

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**
  - 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 & SE
    - Compilers, Operating Systems, Networks, Embedded Systems, Distributed Systems, etc.

Reality vs. Abstraction

- **Examples:**
  - Is $x^2 \geq 0$?
    - Float's: Yes!
    - Int's:
      - 40000 * 40000 --> 1600000000
      - 50000 * 50000 --> ??
  - Is (x + y) + z = x + (y + z)?
    - Unsigned & Signed Int's: Yes!
    - Float's:
      - (1e20 + -1e20) + 3.14 --> 3.14
      - 1e20 + (-1e20 + 3.14) --> ??

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**

Typical Usage

```
/* 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;
}
```

```
#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];

/* 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

Assembly is important

- **Chances are, you'll never write a program in assembly**
  - Compilers are much better & more patient than you are
- **Understanding assembly key to machine-level execution model**
  - Behavior of programs in presence of bugs
    - High-level language model breaks down
  - Tuning program performance
    - 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.

Memory Matters

- **Memory is not unbounded**
  - It must be allocated and managed
  - Many applications are memory dominated
- **Memory referencing bugs especially evil**
  - 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
```

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

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

On with the show!

- **That was basically Chapter 1. It will give you a good feel for the big picture.**
- **On to section 2.1**

Bits and Bytes

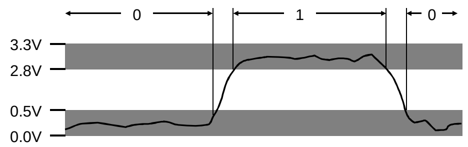
- **Topics**
  - Why bits?
  - Representing information as bits
    - Binary/Hexadecimal
    - Byte representations
      - numbers
      - characters and strings
      - Instructions
  - Bit-level manipulations
    - Boolean algebra
    - Expressing in C

Why Don't Computers Use Base 10?

- **Base 10 Number Representation**
  - No coincidence that there are 10 numerical digits and 10 digits on the hands.
  - Natural representation for financial transactions
    - Floating point number cannot exactly represent $1.20
  - Even carries through in scientific notation
    - 1.5213×10^4
- **Implementing Electronically**
  - Hard to store
    - ENIAC (First electronic computer) used 10 vacuum tubes / digit
  - Hard to transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.

Binary Representations

- **Base 2 Number Representation**
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
  - Represent 1.5213×10^4 as **1.1101101101101 E 1101**
- **Electronic Implementation**
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires



  - Straightforward implementation of arithmetic functions

Byte-Oriented Memory Organization

- **Programs Refer to Virtual Addresses**
  - Conceptually very large array of bytes
  - Indexed via integers encoded in binary
- **Compiler + Run-Time System Control Allocation**
  - Where different program objects should be stored
  - Multiple mechanisms: static, stack, and heap
  - In any case, all allocation within single virtual address space

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`
      - Or `0xfa1d37b`
- **Why hexadecimal?**

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

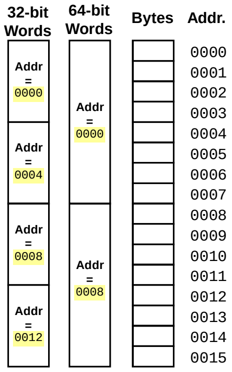
Machine Words

- **Machine Has "Word Size"**
  - Nominal size of integer-valued data
    - Including addresses
  - When book was written most machines were 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Modern PCs are 64 bits (8 bytes)
    - Potentially address ≈ 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

Systems I

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)



Systems I

Data Representations

- **Sizes of C Objects (in Bytes)**

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
char *	4	4	8

- Or any other pointer

Systems I

Byte Ordering

- **How should bytes within 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`

Why not 76543210?

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

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 4 bytes:** `0x000012ab`
- **Split into bytes:** `00 00 12 ab`
- **Reverse:** `ab 12 00 00`

Examining Data Representations

- **Code to Print Byte Representation of Data** See K&R Chap. 5
  - 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("0x%p\t0x%.2x\n",
           start+i, start[i]);
  printf("\n");
}
```

See K&R or man 2 printf

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

Systems I

show_bytes Execution Example

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

Result (Linux):

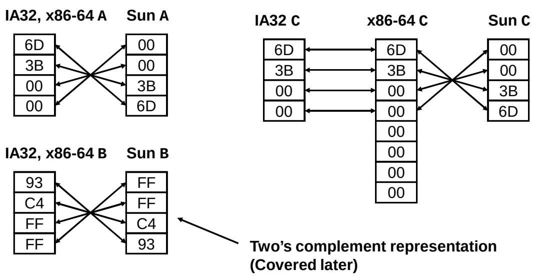
```
int a = 15213;
0x11ffffcb8   0x6d
0x11ffffcb9   0x3b
0x11ffffcba   0x00
0x11ffffcbb   0x00
```

Systems I

Representing Integers

- `int A = 15213;`
- `int B = -15213;`
- `long int C = 15213;`

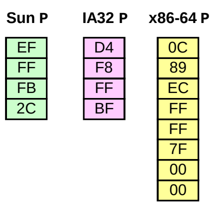
Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D



Systems I

Representing Pointers

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



Different compilers & machines assign different locations to objects

Representing Floats

- `Float F = 15213.0;`

IA32 F	Sun F
00	46
B4	6D
6D	B4
46	00

IEEE Single Precision Floating Point Representation

Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
15213:			1110	1101	1011	01		

Not same as integer representation, but consistent across machines

Can see some relation to integer representation, but not obvious

Representing Strings

```
char S[6] = "15213";
```

Linux/Alpha S	Sun S
31	31
35	35
32	32
31	31
33	33
00	00

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Other encodings exist, but uncommon
    - 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
    - Data are single byte quantities
  - Text files generally platform independent
    - Except for different conventions of line termination character(s)!

Machine-Level Code Representation

- **Encode Program as Sequence of Instructions**
  - Each simple operation
    - Arithmetic operation
    - Read or write memory
    - Conditional branch
  - Instructions encoded as bytes
    - Sun's, Mac's use 4 byte instructions
      - Reduced Instruction Set Computer (RISC)
    - PC's use variable length instructions
      - Complex Instruction Set Computer (CISC)
  - Different instruction types and encodings for different machines
    - Most code not binary compatible
- **Programs are Byte Sequences Too!**

Representing Instructions

```
int sum(int x, int y)
{
   return x+y;
}
```

- For this example, Alpha & Sun use two 4-byte instructions
  - Use differing numbers of instructions in other cases
- PC uses 7 instructions with lengths 1, 2, and 3 bytes

Alpha sum	Sun sum	PC sum
00	81	55
00	C3	89
30	E0	E5
42	08	8B
01	90	45
80	02	0C
FA	00	03
6B	09	45
		08
		89
		EC
		5D
		C3

Different machines use totally different instructions and encodings

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

Boolean Algebra ≈ Integer Ring

- *Commutativity*
  - A | B = B | A — A + B = B + A
  - A & B = B & A — A * B = B * A
- *Associativity*
  - (A | B) | C = A | (B | C) — (A + B) + C = A + (B + C)
  - (A & B) & C = A & (B & C) — (A * B) * C = A * (B * C)
- *Product distributes over sum*
  - A & (B | C) = (A & B) | (A & C) — A * (B + C) = A * B + B * C
- *Sum and product identities*
  - A | 0 = A — A + 0 = A
  - A & 1 = A — A * 1 = A
- *Zero is product annihilator*
  - A & 0 = 0 — A * 0 = 0
- *Cancellation of negation*
  - ~ (~ A) = A — – (– A) = A

Boolean Algebra ≠ Integer Ring

- Boolean: *Sum distributes over product*
  A | (B & C) = (A | B) & (A | C) A + (B * C) ≠ (A + B) * (B + C)
- Boolean: *Idempotency*
  A | A = A A + A ≠ A
  - "A is true" or "A is true" = "A is true"
  A & A = A A * A ≠ A
- Boolean: *Absorption*
  A | (A & B) = A A + (A * B) ≠ A
  - "A is true" or "A is true and B is true" = "A is true"
  A & (A | B) = A A * (A + B) ≠ A
- Boolean: *Laws of Complements*
  A | ~A = 1 A + –A ≠ 1
  - "A is true" or "A is false"
- Ring: *Every element has additive inverse*
  A | ~A ≠ 0 A + –A = 0

Properties of & and ^

Boolean Ring
- ⟨{0,1}, ^, &, *I*, 0, 1⟩
- Identical to integers mod 2
- *I* is identity operation: *I* (A) = A
  A ^ A = 0

Property	**Boolean Ring**
Commutative sum	A ^ B = B ^ A
Commutative product	A & B = B & A
Associative sum	(A ^ B) ^ C = A ^ (B ^ C)
Associative product	(A & B) & C = A & (B & C)
Prod. over sum	A & (B ^ C) = (A & B) ^ (B & C)
0 is sum identity	A ^ 0 = A
1 is prod. identity	A & 1 = A
0 is product annihilator	A & 0 = 0
Additive inverse	A ^ A = 0

Relations Between Operations

DeMorgan's Laws
- Express & in terms of |, and vice-versa
  - A & B = ~(~A | ~B)
    - A and B are true if and only if neither A nor B is false
  - A | B = ~(~A & ~B)
    - A or B are true if and only if A and B are not both false

Exclusive-Or using Inclusive Or
  - A ^ B = (~A & B) | (A & ~B)
    - Exactly one of A and B is true
  - A ^ B = (A | B) & ~(A & B)
    - A or B is true, but not both

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}
  - $a_j = 1$ if $j \in A$

```
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`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41`
  - `~0x00`
  - `0x69 & 0x55`
  - `0x69 | 0x55`

Systems I

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)

Systems I

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
    - Useful with two's complement integer representation

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Systems I

Cool Stuff with Xor

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse
  A ^ A = 0

```
{
  int x = ...
  int y = ...
  x = x ^ y;     /* #1 */
  y = x ^ y;     /* #2 */
  x = x ^ y;     /* #3 */
}
```

	x	y
Begin	A	B
1	A^B	B
2	A^B	(A^B)^B = A
3	(A^B)^A = B	A
End	B	A

Systems I

Main Points

- **It's All About Bits & Bytes**
  - Numbers
  - Programs
  - Text
- **Different Machines Follow Different Conventions**
  - Word size
  - Byte ordering
  - Representations
- **Boolean Algebra is Mathematical Basis**
  - Basic form encodes "false" as 0, "true" as 1
  - General form like bit-level operations in C
    - Good for representing & manipulating sets