Anita's Super Awesome Recitation Slides

15/18-213: Introduction to Computer Systems

Assembly and GDB, 16 Sept 2013

Anita Zhang

Management and Stuff

- Bomb Lab due Tues, 24 Sept 2013, 11:59 PM
  - This is my favorite lab!
- Buf Lab out Tues, 24 Sept 2013, 11:59 PM
  - One week long lab

What's on the Menu Today?

- Help (again)
- Books (again)
- Motivation
- Registers & Assembly
- Bomb Lab Overview
- GDB
- More Bomb Lab
- Must Know Unix Commands
- Walkthrough

Helping Us, Helping You?

- Email us: 15-213-staff@cs.cmu.edu
  - Please attach C files if you have a specific question
  - TAs + Professors → More coverage, fast replies
- All projects on Autolab: autolab.cs.cmu.edu
- Office Hours: Sun-Thurs, 5:30PM – 8:30 PM
  - Wean 5207
- Peer Tutoring: Tues 8:30 – 11 PM
  - Mudge Reading Room

What Have You Read?

- Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective, Second Edition*, Prentice Hall, 2011

- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, Second Edition*, Prentice Hall, 1988

- Koenig, Andrew. *C Traps and Pitfalls*. Reading, MA: Addison-Wesley, 1988

- Kernighan, Brian W., and Rob Pike. *The Practice of Programming*. Reading, MA: Addison-Wesley, 1999

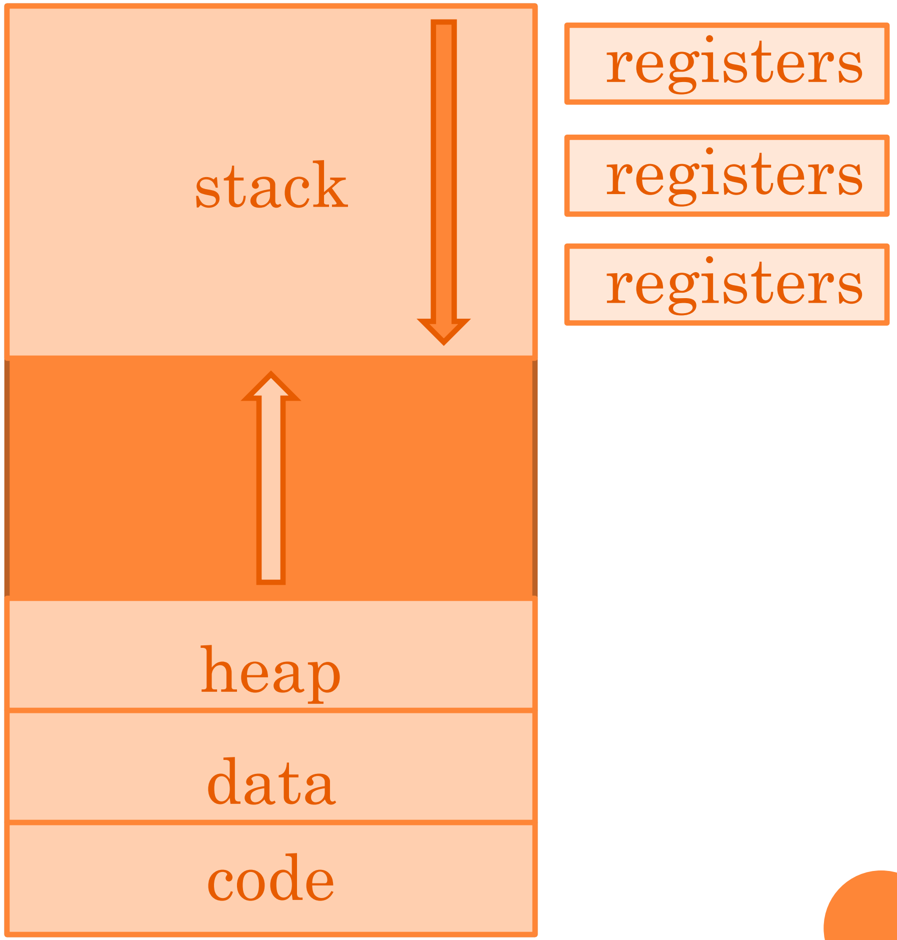
WHY ARE WE DOING THIS AGAIN?

Insight for the Inquisitive

- Why are we not learning about the stack yet?
  - Because x86_64
- "Technology note"
  - x86(_64) only

What are Registers?

- Register
  - Some place in hardware that stores bits
  - It is NOT on the stack or in main memory
- Important
  - When moving data between registers and memory, only the DATA moves, not the register



Registers and All Them Bits

%rax – 64 bits

%eax – 32 bits

%ax – 16 bits

%ah
8 bits

%al
8 bits

- Quad = 64 bits
- Doubleword = 32 bits
- Word = 16 bits
- Byte = 8 bits

These are all parts of the same register

What We're Working With

- x86_64 conventions on the next slide
- Specials
  - %eip – instruction pointer
    - Points to the NEXT instruction to execute
  - %esp – stack pointer
    - Points to top of the stack
  - %eax – holds the return value
    - Also general purpose
- Conditional Flags
  - Sit in a special register of its own
  - Don't really need to worry about it

x86_64, Lots of Registers!

64 bits wide	32 bits wide	16 bits wide	8 bits wide	8 bits wide	Use
%rax	%eax	%ax	%ah	%al	Return Value
%rbx	%ebx	%bx	%bh	%bl	Callee Save
%rcx	%ecx	%cx	%ch	%cl	4^{th} Argument
%rdx	%edx	%dx	%dh	%dl	3^{rd} Argument
%rsi	%esi	%si		%sil	2^{nd} Argument
%rdi	%edi	%di		%dil	1^{st} Argument
%rbp	%ebp	%bp		%bpl	Callee Save
%rsp	%esp	%sp		%spl	Stack Pointer
%r8	%r8d	%r8w		%r8b	5^{th} Argument
%r9	%r9d	%r9w		%r9b	6^{th} Argument
%r10	%r10d	%r10w		%r10b	Caller Save
%r11	%r11d	%r11w		%r11b	Caller Save
%r12	%r12d	%r12w		%r12b	Callee Save
%r13	%r13d	%r13w		%r12b	Callee Save
%r14	%r14d	%rw		%14b	Callee Save
%r15	%r15d	%r15w		%15b	Callee Save

Some More Definitions

- Memory Addressing
  - How assemblers denote memory locations
    - Direct
    - Indirect
    - Relative
    - Absolute
    - …
  - Many different syntactical ways to represent the same address

Reasons Why Intel is Ridiculous and Awesome

- Operations can take several forms:
  - Register-to-Register
  - Register-to-Memory / Memory-to-Register
  - Immediate-to-Register / Immediate-to-Memory
  - One address operations (push, pop)
  - Did I miss any?
- Fun fact: Why not memory to memory?

Representing Addresses

- x86(_64) Common Addressing
  - Offset(Base, Index, Scale)
  - D(Rb, Ri, S) → Mem[Rb + Ri*S + D]
    - D can be any signed integer
    - Scale is 1, 2, 4, 8 (assume 1 if omitted)
    - Assume 0 for base if omitted

Representing Addresses

- Using parenthesis
  - Most of the time parenthesis means dereference
    - This is still only x86(_64)
- Examples of parenthesis usage:
  - `(%eax)`
    - Contents of memory at address stored, %eax
  - `(%ebx, %ecx)`
    - Contents of memory stored at address, %ebx + %ecx
  - `(%ebx, %ecx, 8)`
    - Contents of memory stored at address, %ebx + 8*%ecx
  - `4(%ebx, %ecx, 8)`
    - Contents of memory stored at address, %ebx + 8*%ecx + 4

Representing Addresses

- Using parenthesis
  - Sometimes parenthesis are used just for addressing
    - This is still only x86(_64)
- Example
  - `leal (%ebx, %ecx, 8), destination`
    - Take only the values ➔ %ebx + 8*%ecx
    - Does not dereference, uses the calculated value directly
  - `jmpq *0x402660(,%rax,8)`
    - The * does the dereference
- Examples of not using parenthesis
  - %eax
    - Use the value in %eax!
  - $0x213
    - A constant value

Review of Conditionals/ Flags

- Most operations will set conditional flags
  - Bit operations
  - Arithmetic
  - Comparisons…
- Core idea: For conditionals, look one instruction before it to see whether it is true or false
  - Will be explained

Flags We (Might) Care About

- Carry (CF)
  - Arithmetic carry/ borrow
- Parity (PF)
  - Odd or even number of bits set
- Zero (ZF)
  - Result was zero
- Sign (SF)
  - Most significant bit was set
- Overflow (OF)
  - Result does not fit into the location

Prep for All the cheat sheets

- Warning: The following slides contain lots of assembly instructions.
  - All from CS:APP (our textbook BTW)
  - We're not going over every single one…
  - Use it as a reference for Bomb Lab
- Quick note on Intel vs. AT&T
  - This is AT&T syntax (also, Bomb Lab syntax)
    - Looks like: "src, dest"
  - Intel tends to follow "dest, src"
    - Check out their ISA sometime

All the cheat Sheets (Movement)

Instruction		Effect
movb	S, D	Move byte
movw	S, D	Move word
movl	S, D	Move doubleword
movsbw	S, D	Move byte to word (sign extended)
movsbl	S, D	Move byte to doubleword (sign extended)
movswl	S, D	Move word to doubleword (sign extended)
movzbw	S, D	Move byte to word (zero extended)
movzbl	S, D	Move byte to doubleword (zero extended)
movzwl	S, D	Move word to doubleword (zero extended)
pushl	S	Push double word (Mem[%esp] ← S; %esp = %esp 4)
popl	D	Pop double word (D ← Mem[%esp]; %esp = %esp + 4)

All the cheat Sheets (Bit Ops)

Instruction		Effect
LEAL	S, D	D ← &S (Load address of source into destination)
INC	D	D ← D + 1
DEC	D	D ← D – 1
NEG	D	D ← –D
NOT	D	D ← ~D
ADD	S, D	D ← D + S
SUB	S, D	D ← D – S
IMUL	S, D	D ← D * S
XOR	S, D	D ← D ^ S
OR	S, D	D ← D | S
AND	S, D	D ← D & S
SAL	k, D	D ← D << k
SHL	k, D	D ← D << k
SAR	k, D	D ← D >> k (arithmetic shift)
SHR	k, D	D ← D >> k (logical shift)

All the cheat Sheets (Specials)

Instruction		Effect
imull	S	R[%edx]:R[%eax] ← S * R[%eax] Signed multiply of %eax by S Result stored in %edx:%eax
mull	S	R[%edx]:R[%eax] ← S * R[%eax] Unsigned multiply of %eax by S Result stored in %edx:%eax
cltd		R[%edx]:R[%eax] ← SignExtend(R[%eax]) Sign extend %eax into %edx
idivl	S	R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S Signed divide of %eax by S Quotient stored in %eax Remainder stored in %edx
divl	S	R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S Unsigned divide of %eax by S Quotient stored in %eax Remainder stored in %edx

All the cheat Sheets (Comparisons)

Instruction		Effect
cmpb	S2, S1	Compare byte S1 and S2, Sets conditional flags based on S1 – S2.
cmpw	S2, S1	Compare word S1 and S2, Sets conditional flags based on S1 – S2.
cmpl	S2, S1	Compare double word S1 and S2, Sets conditional flags based on S1 – S2.
testb	S2, S1	Compare byte S1 and S2, Sets conditional flags based on S1 & S2.
testw	S2, S1	Compare word S1 and S2, Sets conditional flags based on S1 & S2.
testl	S2, S1	Compare double word S1 and S2, Sets conditional flags based on S1 & S2.

All the cheat Sheets (Set)

Instruction		Effect
sete/ setz	D	D ← ZF ("set if equal to 0")
setne/ setnz	D	D ← ~ZF (set if not equal to 0)
sets	D	D ← SF (set if negative)
setns	D	D ← ~SF (set if nonnegative)
setg/ setnle	D	D ← ~(SF ^ OF) & ~ZF (set if greater (signed >))
setge/ setnl	D	D ← ~(SF ^ OF) (set if greater or equal (signed >=))
setl/ setnge	D	D ← SF ^ OF (set if less than (signed <))
setle/ setng	D	D ← (SF ^ OF) | ZF (set if less than or equal (signed <=))
seta/ setnbe	D	D ← ~CF & ~ZF (set if above (unsigned >))
setae/ setnb	D	D ← ~CF (set if above or equal (unsigned >=))
setb/ setnae	D	D ← CF (set if below (unsigned <))
setbe/ setna	D	D ← CF | ZF (set if below or equal (unsigned <=))

All the cheat Sheets (Jump)

<table>
<tr><th colspan="2">Instructions</th><th>Effect</th></tr>
<tr><td>jmp</td><td>Label</td><td>Jump to label</td></tr>
<tr><td>jmp</td><td>*Operand</td><td>Jump to specified locations</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>je/ jz</td><td>Label</td><td>Jump if equal/ zero (ZF)</td></tr>
<tr><td>jne/ jnz</td><td>Label</td><td>Jump if not equal/ nonzero (~ZF)</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>js</td><td>Label</td><td>Jump if negative (SF)</td></tr>
<tr><td>jns</td><td>Label</td><td>Jump if nonnegative (~SF)</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>jg/ jnle</td><td>Label</td><td>Jump if greater (signed) (~(SF ^ OF) & ~ZF)</td></tr>
<tr><td>jge/ jnl</td><td>Label</td><td>Jump if greater or equal (signed) (~(SF ^ OF))</td></tr>
<tr><td>jl/ jnge</td><td>Label</td><td>Jump if less (signed) (SF ^ OF)</td></tr>
<tr><td>jle/ jng</td><td>Label</td><td>Jump if less or equal (signed) ((SF ^ OF) | ZF)</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>ja/ jnbe</td><td>Label</td><td>Jump if above (unsigned) (~CF & ~ZF)</td></tr>
<tr><td>jae/ jnb</td><td>Label</td><td>Jump if above or equal (unsigned) (~CF)</td></tr>
<tr><td>jb/ jnae</td><td>Label</td><td>Jump if below (unsigned) (CF)</td></tr>
<tr><td>jbe/ jna</td><td>label</td><td>Jump if below or equal (unsigned) (CF | ZF)</td></tr>
</table>

ALL THE CHEAT SHEETS (CMOVE)

Instruction		Effect
cmove/ cmovz	S, R	S ← R if Equal/ zero (ZF)
cmovne/ cmovnz	S, R	S ← R if Not equal/ not zero (~ZF)
cmovs	S, R	S ← R if Negative (SF)
cmovns	S, R	S ← R if Nonnegative (~SF)
cmovg/ cmovnle	S, R	S ← R if Greater (signed >) (~(SF ^ OF) & ~ZF)
cmovge/ cmovnl	S, R	S ← R if Greater or equal (signed >=) (~(SF ^ OF))
cmovl/ cmovnge	S, R	S ← R if Less (signed <) (SF ^ OF)
cmovle/ cmovg	S, R	S ← R if Less or equal (signed <=) ((SF ^ OF) | ZF)
cmova/ cmovnbe	S, R	S ← R if Above (unsigned >) (~CF & ~ZF)
cmovae/ cmovnb	S, R	S ← R if Above or equal (unsigned >=) (~CF)
cmovb/ cmovnae	S, R	S ← R if Below (unsigned <) (CF)
cmovbe/ cmovna	S, R	S ← R if Below or equal (unsigned <=) (CF | SF)

All the cheat Sheets (Calling)

Instruction		Effect
call	Label	Push return and jump to label
call	*operand	Push return and jump to specified location
leave		Prepare stack for return. Set stack pointer to %ebp and pop top stack into %ebp. In assembly: *mov %ebp, %esp* *pop %ebp*
ret		Pop return address from stack and jump there

Jumps, In Depth

```
test %al,%al
jne  4011ed
```

```
if ((%al & %al) != 0)
  jump to 4011ed
```

- The test instruction is usually followed by jump if equal/ not equal

```
cmpl $0x5,0x14(%rsp)
jg   4011d0
```

```
if (0x14(%rsp) > $0x5)
  jump to 4011d0
```

- For conditional jumps, it is usually the second argument greater/less than first argument

JE, JNE, JLE, JGE, ETC

- Jump if equal == Jump if zero
  - If the previous result was 0, jump
- Jump if not equal == Jump if not zero
  - If the previous result was not 0, jump
- Don’t worry about the conditional flags
  - Just remember “if second argument greater/less than first argument”

Dr. Evil and Bomblab

- 6 stages, each asking for input
  - Wrong input → bomb explodes (lose 1/2 point)
    - Score rounds up, so first explosion is free
  - Each stage may have multiple answers

- You get:
  - Bomb executable
  - Partial source of Dr. Evil mocking you

- Speed up next phase traversal with a text file
  - Place answers on each line
  - Run with bomb as ./bomb <solution file>

How it Works

- "But how do I find the solutions if I don't have C code to work from?"
  - Read a lot of bomb disassembly
    - All of the phases are just loops and patterns
  - **GDB**

- If you're not working on a shark machine, your bomb won't work.
  - Will get an "illegal host" error

Working Through This thing

- Read the disassembly
  - `phase_1, phase_2, phase_3…`
  - `explode_bomb`
  - Possible to reason through solutions without using GDB
- GNU Debugger
  - Step through each instruction, examine registers..
  - Set up breakpoints
  - Make sure to run "`kill`" when you hit the `explode_bomb` breakpoint
    - You're screwed once you hit here, so why not exit?

But I Don't Know How to GDB??

- Here have a cheat sheet
  - http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf
  - Everything you need to use GDB to solve bomb lab
- The Internet has a great range of commands you might find useful

GDB's Most Useful

- `run/ run <arguments>`
  - Runs the program up till the next breakpoint.
- `disassemble/ disas`
  - Shows the current function with an arrow to the next
  - WARNING: shortcut "`disa`" disables all breakpoints
- `step/ stepi/ nexti`
  - `stepi` steps to the next line of Assembly.
  - `nexti` does the same but doesn't stop in function calls.
  - `stepi` *n* or `nexti` *n* steps through *n* lines.

GDB's Most Useful

- `break <location>`
  - Sets breakpoint. Location can be function name or address.
  - Stop at an instruction address with `break *address`
  - You have to reset your break points when you restart GDB!
- `x <address/register>`
  - Dereference the address or value in the register and print the contents to the console
  - Give it a format to print out to, ie. “x/s” prints as string
- `p <address/register/variable>`
  - Print the contents of the register, or the variable, or the address to the console
  - Give it a format to print out to, ie. “p/s” prints as string

Getting Started

- Download and untar ON A SHARK MACHINE
  - `tar xvf` *`labhandout.tar`*
- `shark> objdump -d bomb >` *`filename`*
  - Outputs the whole bomb assembly code to a filename
- `shark> objdump -t bomb >` *`filename`*
  - Contains locations of globals, variables, etc
- `shark> strings bomb >` *`filename`*
  - All printable strings used in your bomb
- `shark> gdb bomb`
  - Prepares to run the bomb in gdb

Speed up the Wait

- When you have solutions, put it into a text file
  - Separate each solution with a newline
  - Your bomb will auto-advance completed phases with pre-filled solutions
- Then when you run gdb next time:
  - `(gdb)>  run` *`solution_file`*

Bomb Lab Specifics

- `int sscanf (const char *s, const char *format, ...);`
  - `s`
    - Source string to retrieve data from
  - `format`
    - Formatting string used to get values from the source string
  - …
    - Depending the format string, one location (address) per formatter used to hold values extracted from source string

SSCANF EXAMPLE

```
#include <stdio.h>

int main () {
  char sentence[]="Rudolph is 12 years old";
  char str[20];
  int i;
  sscanf (sentence,"%s %*s %d", str, &i);
  printf ("%s -> %d\n", str, i);
  return 0;
}
```

- Outputs: `Rudolph -> 12`

Relevance to Bomb Lab

- Why do we care about `sscanf`?
  - Mainly used to read in arguments
  - Keep track of which locations the read in values will be stored
    - Important for knowing where arguments will be stored
    - And how they will be used
    - They will usually be store in memory/ on the stack

More Bomb Lab Specifics

- Jump tables
  - In memory, you can think of it as an “array” of locations to jump to
  - Using assembly it is possible to index into the “array”
  - Each entry of the array will hold addresses of instructions

Jump Tables

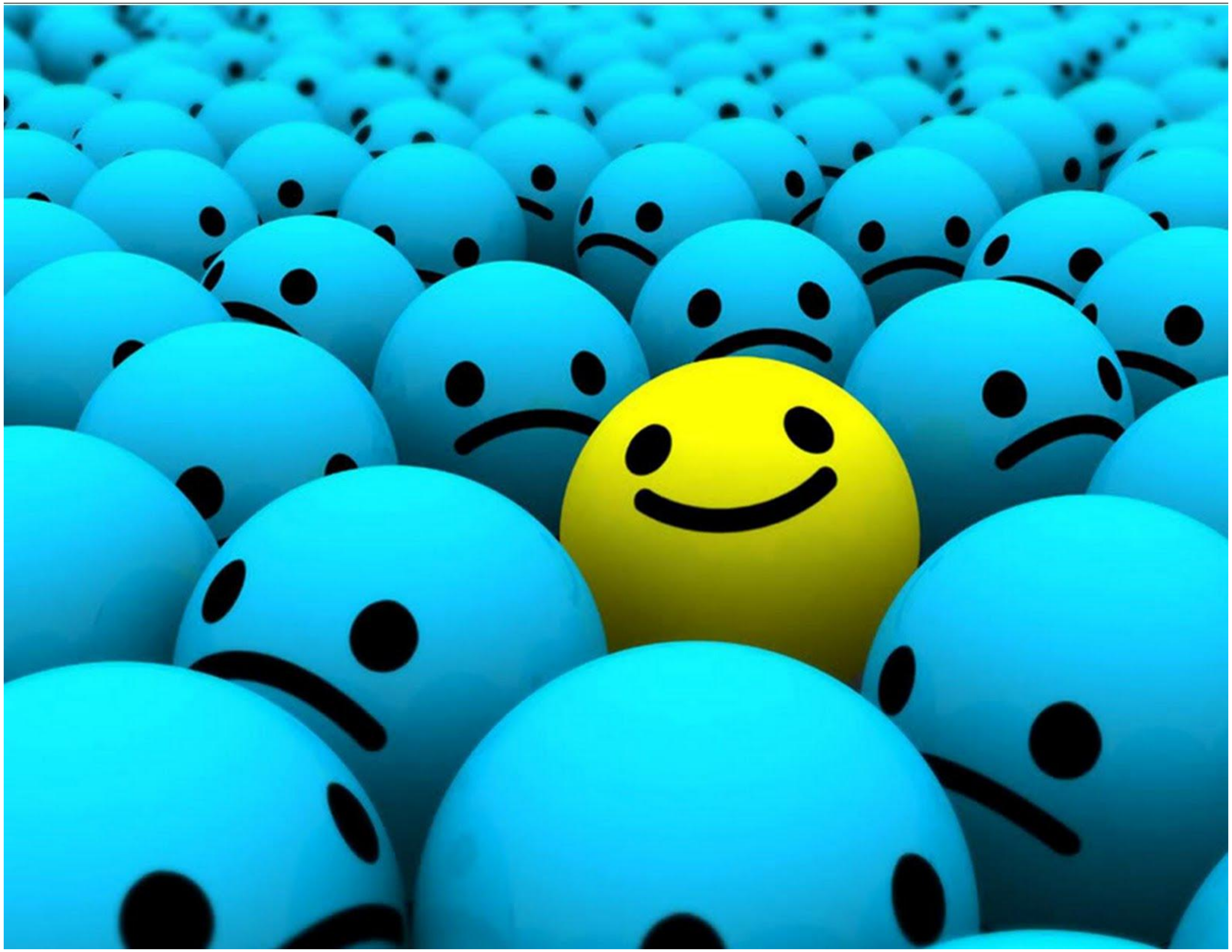
- The tip-off is something like this:
  - `jmpq *0x400600(,%rax,8)`
    - Empty base means implied 0
    - `%rax` is the “index”
    - 8 is the “scale”
      - In a jump tables, 64-bit machine addresses are 8 bytes
    - * indicates a dereference (as in regular C)
      - Like `leal`: does not do a dereference even with parenthesis
  - Put it all together: “Jump to the address stored in the address 0x400600 + %rax*8”
- Using GDB (example output): x/8g 0x400600

```
0x400600: 0x00000000004004d1 0x00000000004004c8
0x400610: 0x00000000004004c8 0x00000000004004be
0x400620: 0x00000000004004c1 0x00000000004004d7
0x400630: 0x00000000004004c8 0x00000000004004be
```

Top Unix Commands

- `man` to read manual pages
- `cd` to change directories
- `ls` to list contents of the current directory
- `ls -l` to list with more info, including permissions
- `scp` to send files between your computer and others
- `ssh` to log into time shares
- `tar` to tar (-cvf) and untar (-xvf) (-z for optional gzip)
- `chmod (ugo)+/-(rwx)` to change permission bits
- Helpful hints
  - Tab auto-completes.
  - An up arrow scrolls up through your last few commands.

Demo Time



Credits & Questions

- StackOverflow on Assembly Projects
- P. 274 of CS:APP – x86_64 Registers
- P. 171 - 221 of CS:APP – Assembly Instructions
- CPlusPlus Reference on sscanf