# Anita's Super Awesome Recitation Slides

15/18-213: Introduction to Computer Systems

Stacks and Buflab, 10 Feb 2014

Anita Zhang

# What's New (or Not)

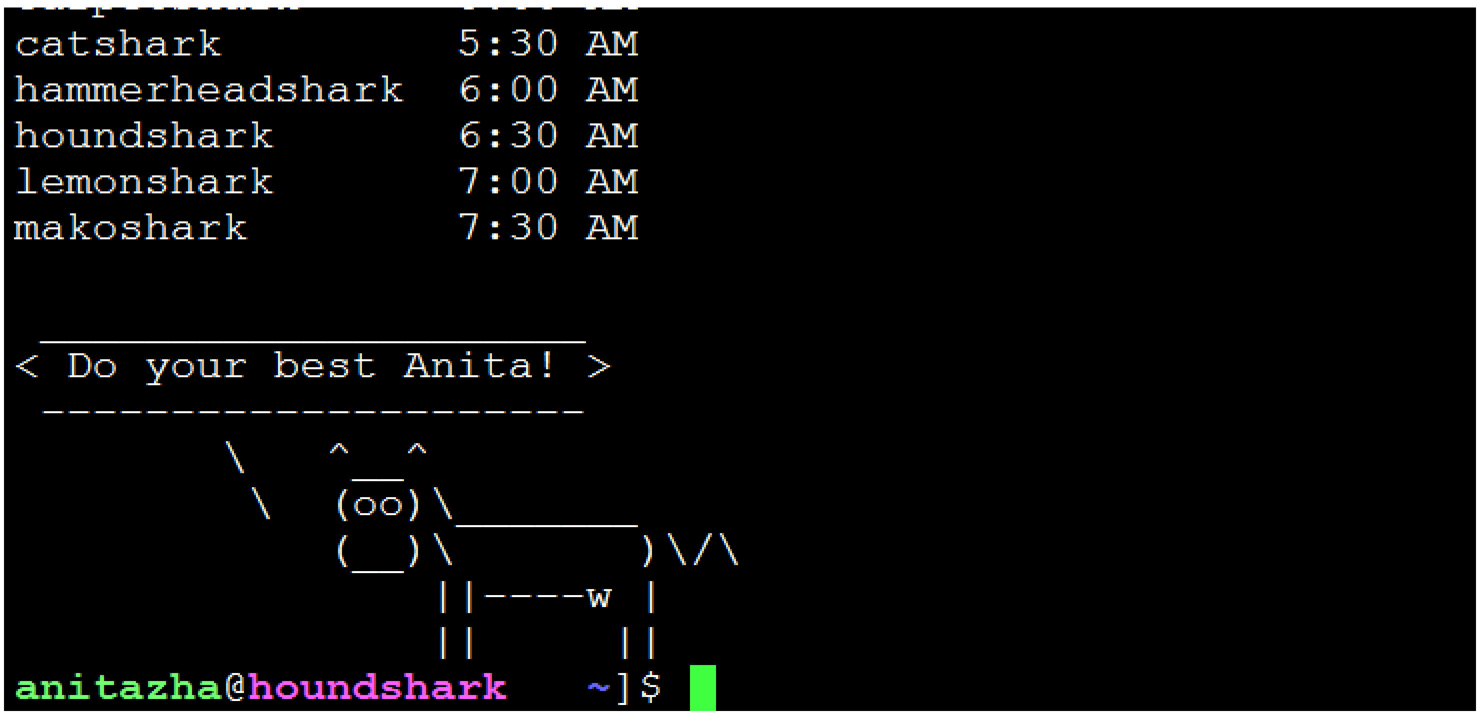
- Bomb Lab is due Tuesday (tomorrow), 11:59 PM
  - Your late days are wasted here
  - Student: *"But if you wait until the last minute, then it only takes a minute!"*
- Buf Lab out Wednesday, 11:59 PM
  - Hacking the stack
  - One week long lab
- Stacks will be on the exams
  - They're tough at first, but I believe in you ☺

# Speaking of the Exam…

- **Midterm: Week of 3 March 2014**
- Covers everything up to, and including, caches.
  - Chapters 1-3 and 6 of textbook.
  - Lectures up to and including Caches (18 Feb 2014).
- Recitation exam review the week of exam.
- "Read each chapter 3 times, work the practice problems, and do previous exams."
  - Do enough midterms until you feel comfortable with the material (at least 5 recent ones).
    - Depending on the semester, caches can be found in Exam 2.

# To Those Who Want A Cool Shell

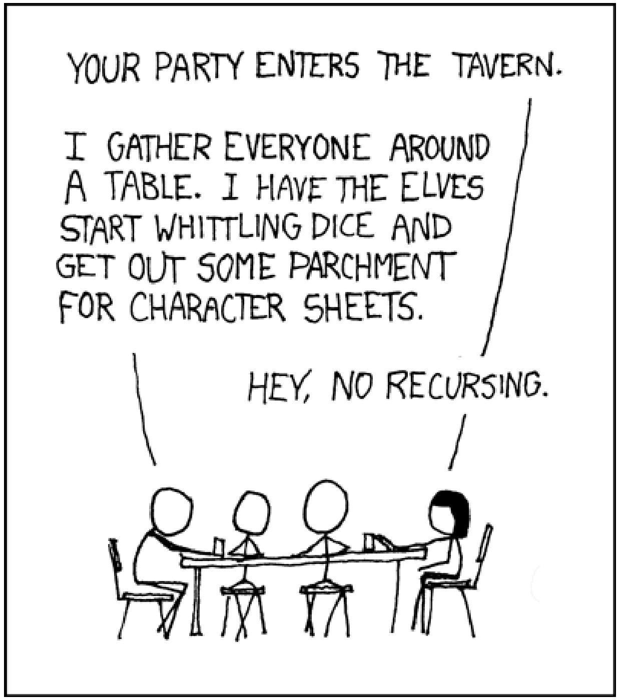
- http://www.contrib.andrew.cmu.edu/~anitazha/15213_tips.html
  - Scroll down to the part about "Shell of Choice"
  - Follow the directions
  - WARNING: THIS SITE IS MESSY AND OUTDATED



# Journey Through Time

- Stacks
  - IA32 Stack Discipline
  - Function Call Overview
  - Stack Walkthrough
  - Extras on x86(_64) stacks
- Buf Lab Quick Start
  - Essential Items of Business
  - Miscellany
- Demo…?

# Some Kind of Stack Motivation




"In order to support general recursion, a language needs a way to allocate different activation records for different invocations of the same function. That way, local variables allocated in one recursive call can coexist with local variables allocated in a different call." (credits to stack overflow)

# Definitions and Conventions

- Register
  - Some place in hardware that stores bits
    - Like boxes on the side of memory
- Caller save
  - Saved by the caller of a function
  - Before a function call, the caller must save any caller save register values it wants preserved
- Callee save
  - Saved by the callee of a function
  - The callee is required to save/ restore values in these registers if it is using these registers in the function

# Aside: Why both?

- Why do we have both caller and callee save?
  - Performance
  - Not all registers need to be saved

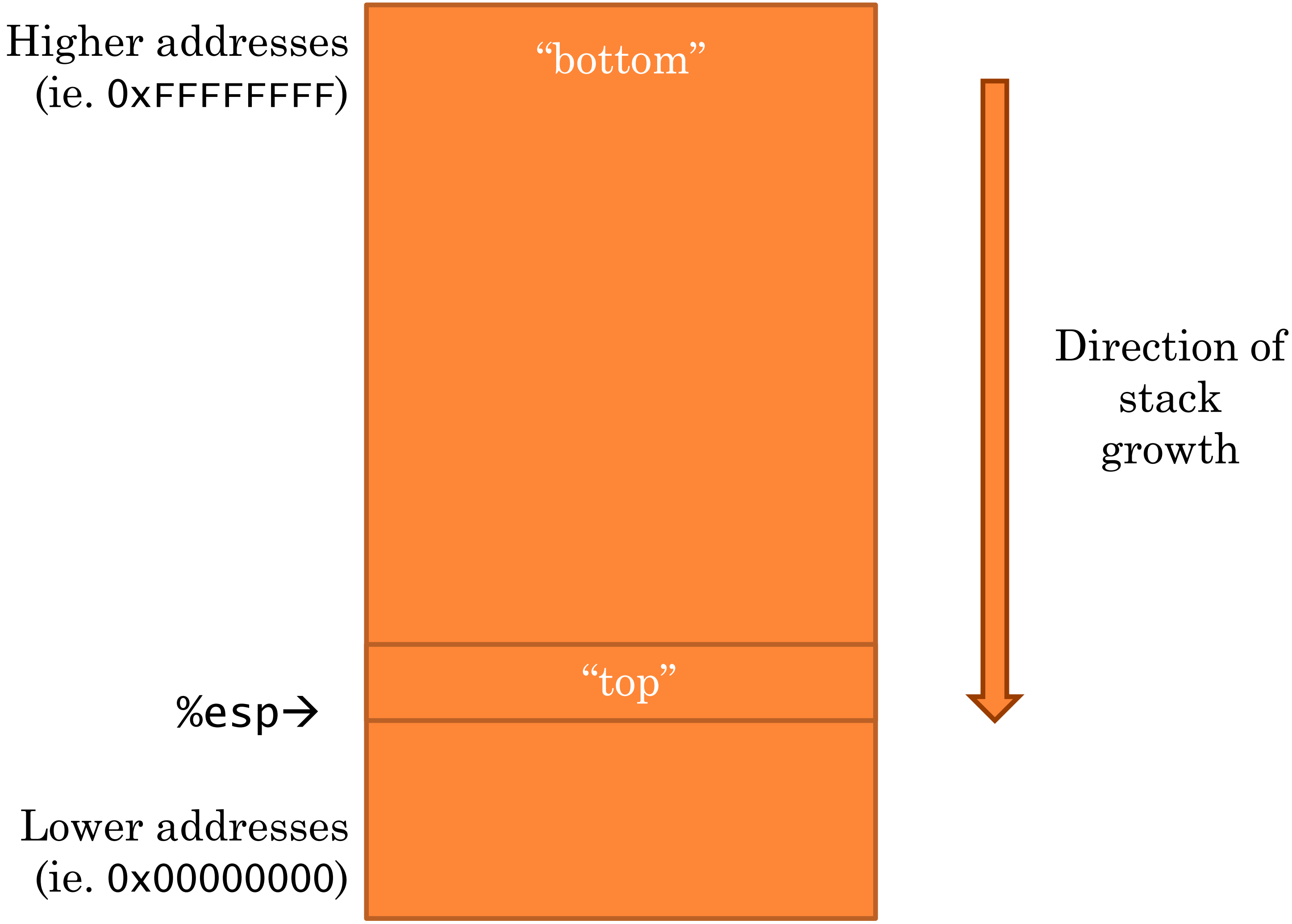
# IA32 Registers

- 6 general purpose registers
  - Caller save
    - `%eax, %ecx, %edx`
    - Saved by the caller of a function
  - Callee save
    - `%ebx, %edi, %esi`
    - Saved by the callee of a function

# Special IA32 Registers

- Base Pointer
  - `%ebp`
  - Points to the "bottom" of the stack frame
    - The location of old `%ebp` that gets pushed on entry
- Stack Pointer
  - `%esp`
  - Points to the "top" of the stack
    - *Usually* whatever was last pushed on the stack
- Instruction Pointer (Program Counter)
  - `%eip`
  - Points to the next instruction to be executed

# IA32 Terminology



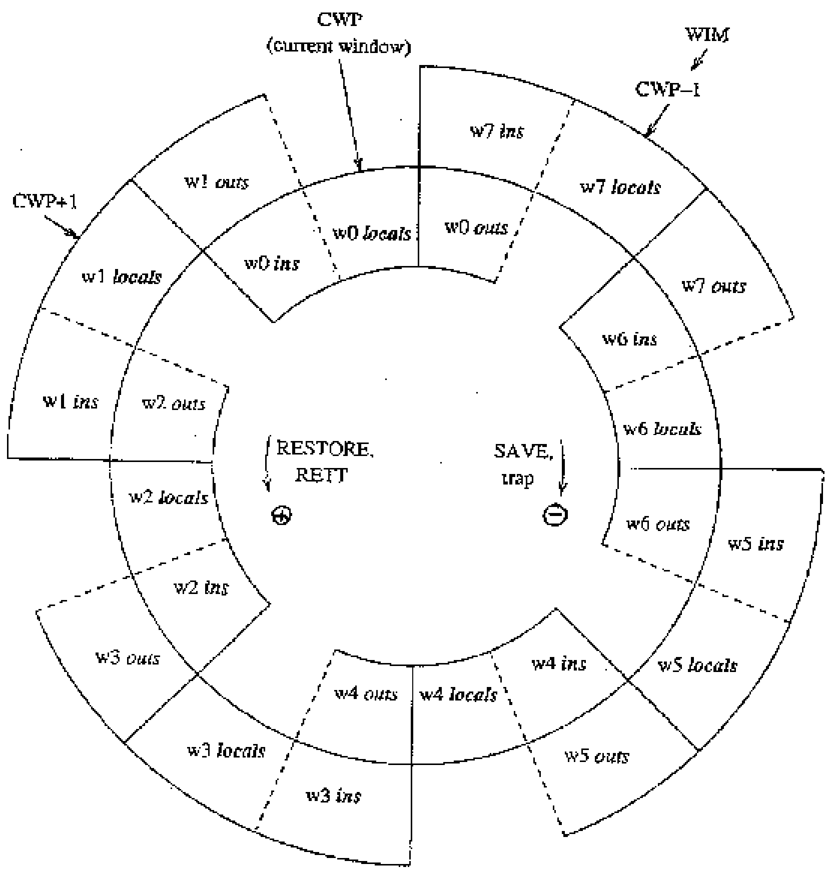
# Aside: Technology Note (Again!)

- This class is (strictly) x86(_64)
  - Other architectures may have different conventions
    - May not use stacks at all (weird, I know)
  - Stacks grow down/ up depending on implementation
    - Very confusing to those new to stacks

# Aside: Direction of Growth

- Stack direction REALLY doesn't matter
  - Direction of growth is dependent on the processor
  - May be selectable for up/down
  - …Or some other direction…?

# BAM! Circular Stack!



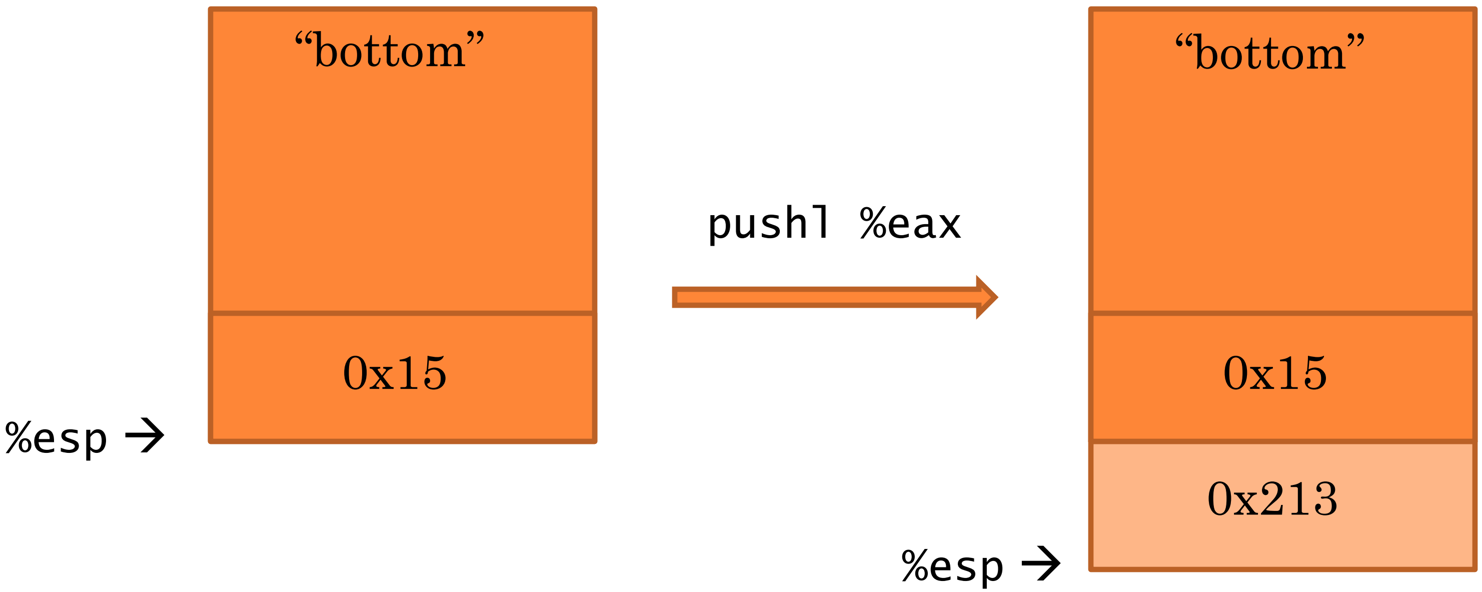

SPARC (scalable processor architecture) Architecture

# Aside: Direction of Growth

- Examples from StackOverflow
  - x86 - down
  - SPARC - in a circle
  - System z  - in a linked list (down, at least for zLinux)
  - ARM - selectable
  - PDP11 - down

# WHAT HAPPENS IN IA32: PUSH

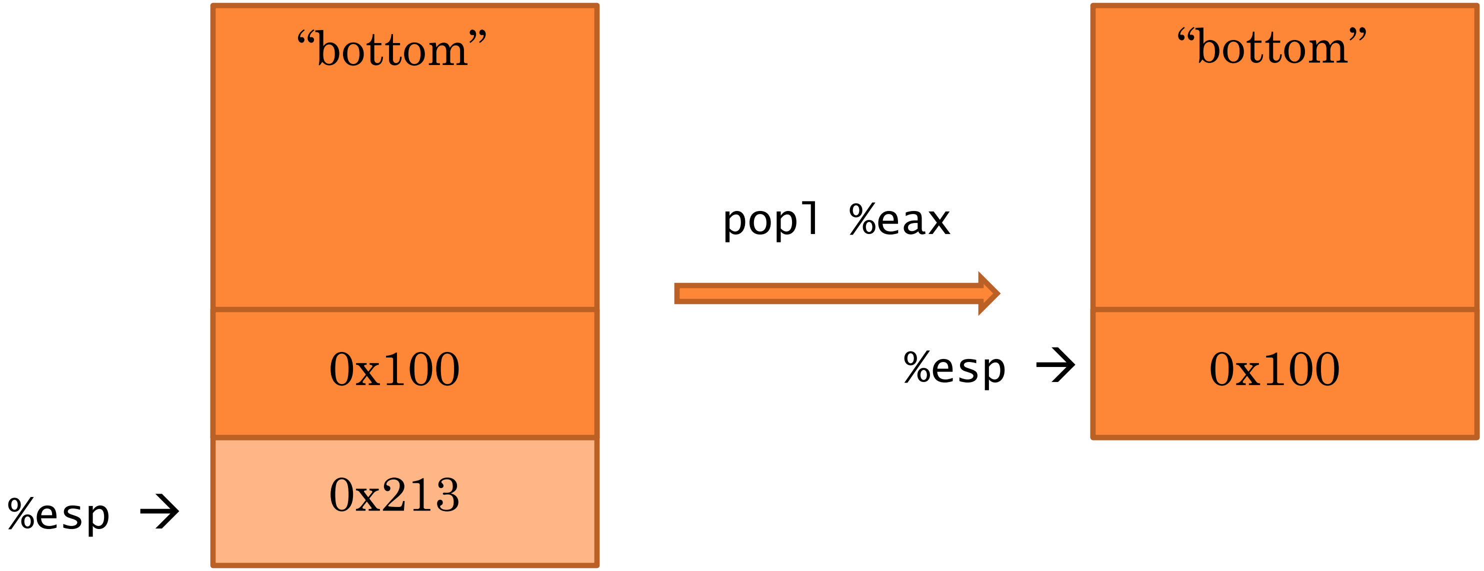
- Pushing on the stack




- In general, `pushl` translates to (in AT&T syntax):
  - ```
    subl $0x4, %esp
    movl src, (%esp)
    ```
    ```

# What Happens in IA32: Pop

- Popping off the stack

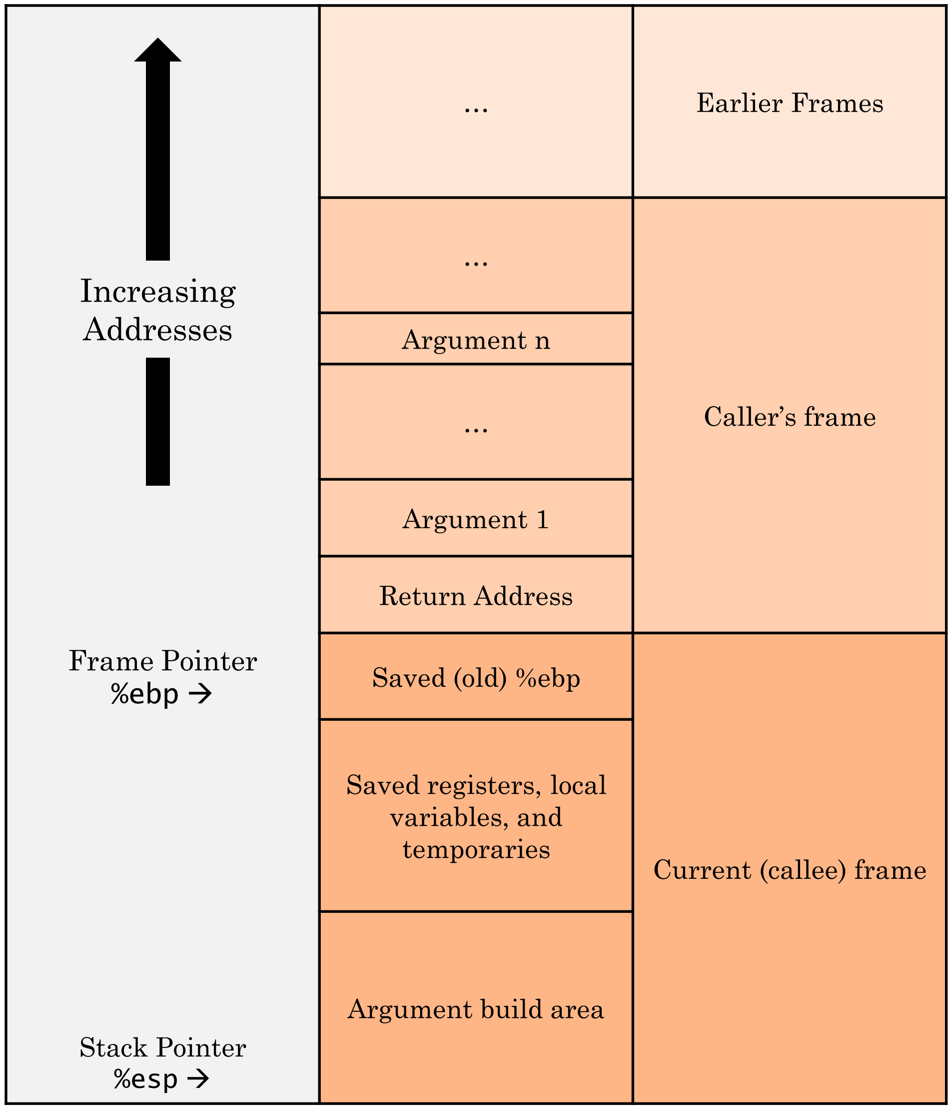



- In general, `popl` translates to (in AT&T syntax):
  - `movl (%esp), dest`
    `addl $0x4, %esp`

# Stack Frames Whatchamacallits?

- Every function call gets a “stack frame”
- All the useful stuff can go on the stack!
  - Local variables (scalars, arrays, structs)
    - What the compiler couldn’t fit into registers
  - Callee/caller save registers
  - Temporary variables
  - Arguments
- Stacks can make recursion work!
- Key idea: “Storage for each *instance* of procedure call” (stolen out of 15-410 slides)

# So THAT'S What it Looks Like…



# Roles of a (Function) Caller

- Caller
  - Save (push) relevant caller save registers
  - Push arguments
  - Call function
- Caller after function return
  - "Remove" (add to `%esp` or pop) arguments
  - Restore (pop) saved caller save registers

# Roles of a (Function) Callee

- Callee
  - Push `%ebp` (save stack frame)
  - Copy (move) `%esp` into `%ebp`
  - Save (push) callee save registers it wants to use
- Callee before return
  - Restore (pop) callee save registers previously saved
  - Copy (move) `%ebp` into `%esp`
    - Moves stack pointer to the saved `%ebp`
  - Restore (pop) `%ebp`

# Function Calls, Other Operations

- Implied operations
  - "`call`" implicitly pushes return address
    - Return address is always of the instruction after the call
  - "`ret`" implicitly pops return address into `%eip`
    - Becomes the next instruction to execute!

# Stack Frames in Action

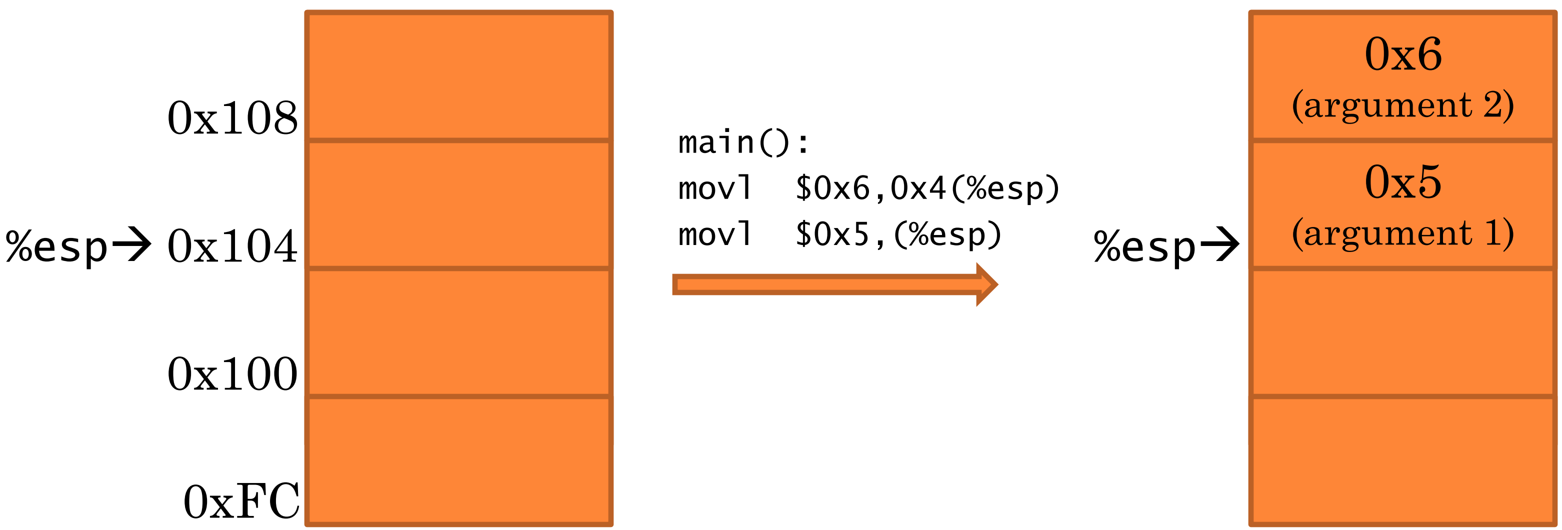
<table>
<tr><th>C Code</th><th>Disassembly</th></tr>
<tr><td><pre>
int main() {
    return addition(5, 6);
}

int addition(int x, int y) {
    return x+y;
}
</pre></td><td><pre>
08048394 <main>:
 8048394:       55                      push   %ebp
 8048395:       89 e5                   mov    %esp,%ebp
 8048397:       83 e4 f0                and    $0xfffffff0,%esp
 804839a:       83 ec 10                sub    $0x10,%esp
 804839d:       c7 44 24 04 06 00 00    movl   $0x6,0x4(%esp)
 80483a4:       00
 80483a5:       c7 04 24 05 00 00 00    movl   $0x5,(%esp)
 80483ac:       e8 02 00 00 00          call   80483b3 <addition>
 80483b1:       c9                      leave
 80483b2:       c3                      ret

080483b3 <addition>:
 80483b3:       55                      push   %ebp
 80483b4:       89 e5                   mov    %esp,%ebp
 80483b6:       8b 45 0c                mov    0xc(%ebp),%eax
 80483b9:       8b 55 08                mov    0x8(%ebp),%edx
 80483bc:       8d 04 02                lea    (%edx,%eax,1),%eax
 80483bf:       c9                      leave
 80483c0:       c3                      ret
</pre></td></tr>
</table>

# Breakdown: Argument Build

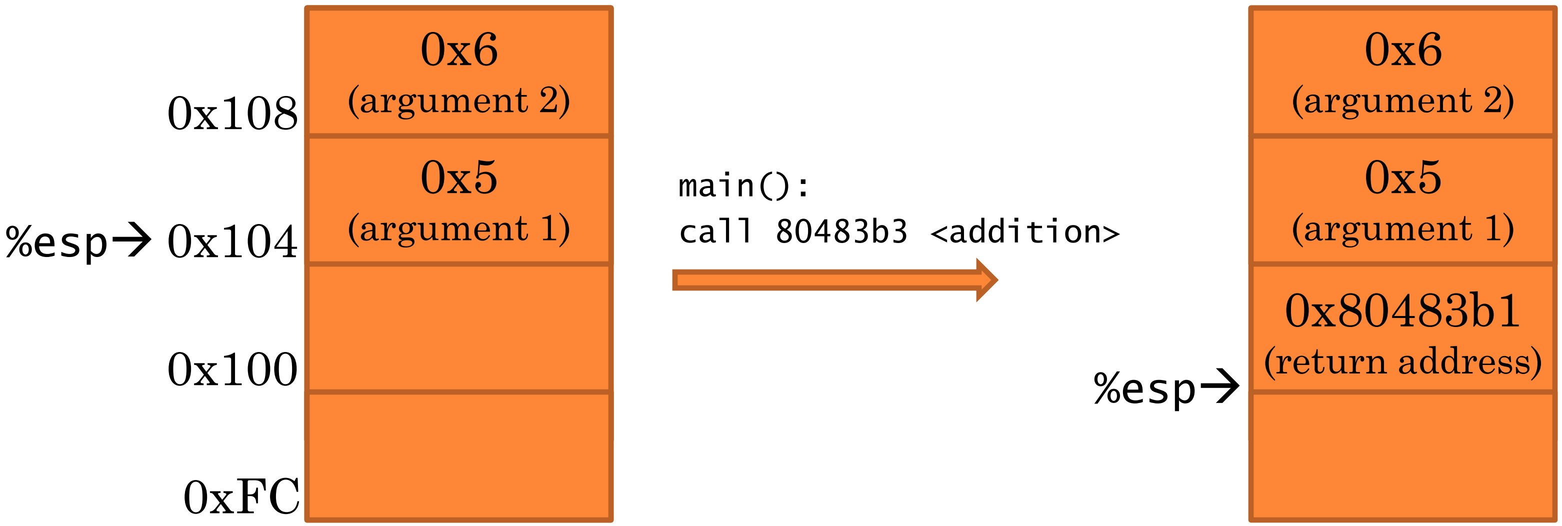
- Build the arguments (note: 2 instructions are executed in this example)




| Before | After |
|---|---|
| %esp = 0x104<br>%ebp = 0x200<br>%eip = 0x804839d | %esp = 0x104<br>%ebp = 0x200<br>%eip = 0x80483ac |

# Breakdown: Function Call

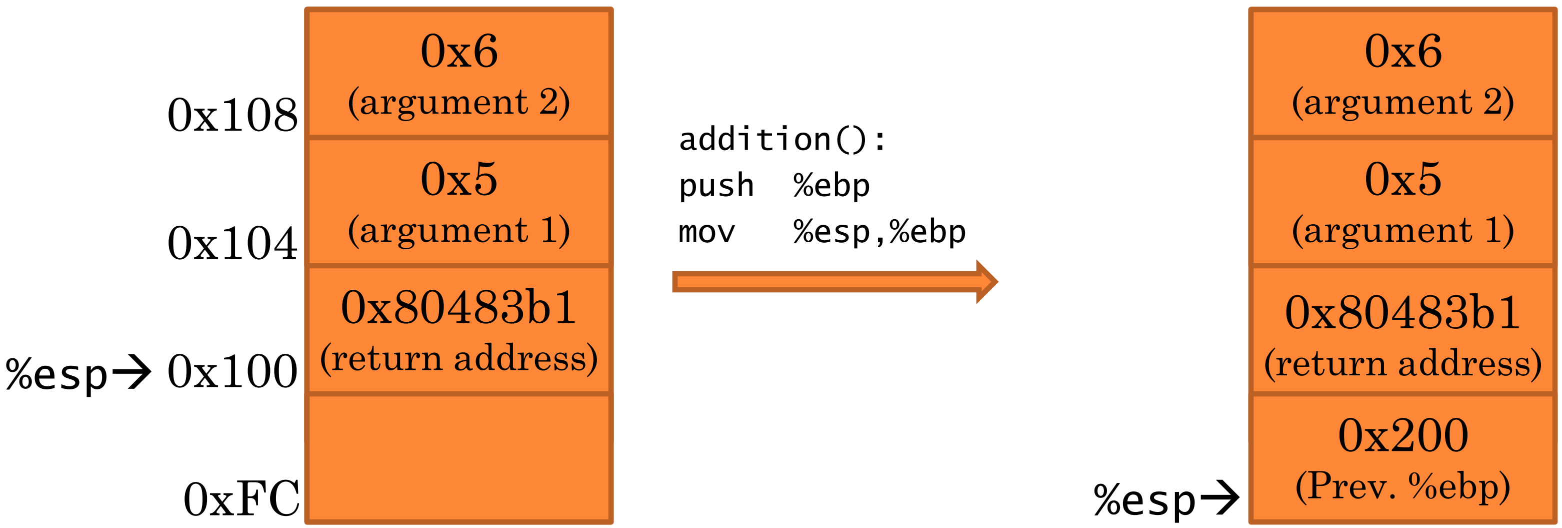
- Call the function




| Before | After |
|---|---|
| %esp = 0x104<br>%ebp = 0x200<br>%eip = 0x80483ac | %esp = 0x100<br>%ebp = 0x200<br>%eip = 0x80483b3 |

# Breakdown: Callee Setup

- Stack frame set up (note: 2 instructions are executed in this example)




| Before | After |
|---|---|
| %esp = 0x100<br>%ebp = 0x200<br>%eip = 0x80483b3 | %esp = 0xFC<br>%ebp = 0xFC<br>%eip = 0x80483b6 |

# BREAK FROM THE EXAMPLE.. KIND OF

- Accessing an argument

| | |
|---|---|
| 0x108 | 0x6 (argument 2) |
| 0x104 | 0x5 (argument 1) |
| 0x100 | 0x80483b1 (return address) |
| 0xFC | 0x200 (Prev. %ebp) |

| Argument | Location |
|---|---|
| Argument 2 | `0xC(%ebp)` |
| Argument 1 | `0x8(%ebp)` |

- In the current frame, arguments are accessed via references to `%ebp`
  - Notice how argument 1 is at `0x8(%ebp)`, not `0x4(%ebp)`

# Let's Review the Code Again

**C Code**

```c
int main() {
    return addition(5, 6);
}

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

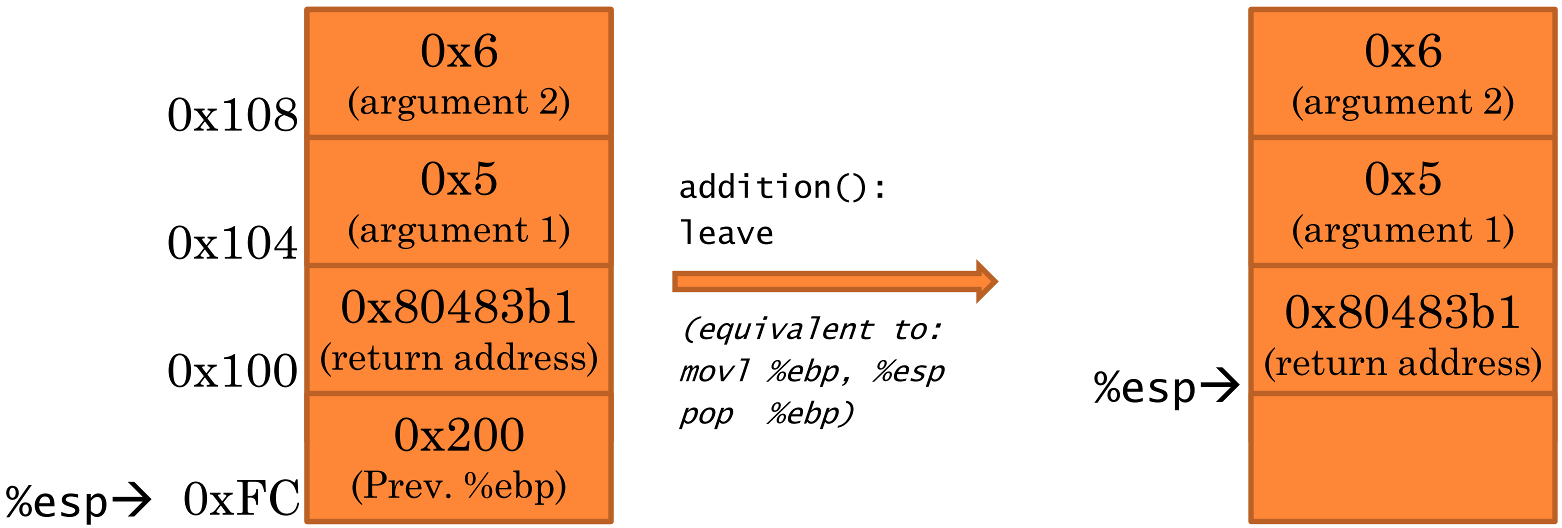
**Disassembly**

```
08048394 <main>:
 8048394:       55                      push   %ebp
 8048395:       89 e5                   mov    %esp,%ebp
 8048397:       83 e4 f0                and    $0xfffffff0,%esp
 804839a:       83 ec 10                sub    $0x10,%esp
 804839d:       c7 44 24 04 06 00 00    movl   $0x6,0x4(%esp)
 80483a4:       00
 80483a5:       c7 04 24 05 00 00 00    movl   $0x5,(%esp)
 80483ac:       e8 02 00 00 00          call   80483b3 <addition>
 80483b1:       c9                      leave
 80483b2:       c3                      ret

080483b3 <addition>:
 80483b3:       55                      push   %ebp
 80483b4:       89 e5                   mov    %esp,%ebp
 80483b6:       8b 45 0c                mov    0xc(%ebp),%eax
 80483b9:       8b 55 08                mov    0x8(%ebp),%edx
 80483bc:       8d 04 02                lea    (%edx,%eax,1),%eax
 80483bf:       c9                      leave
 80483c0:       c3                      ret
```

# Breakdown: Preparing to Return

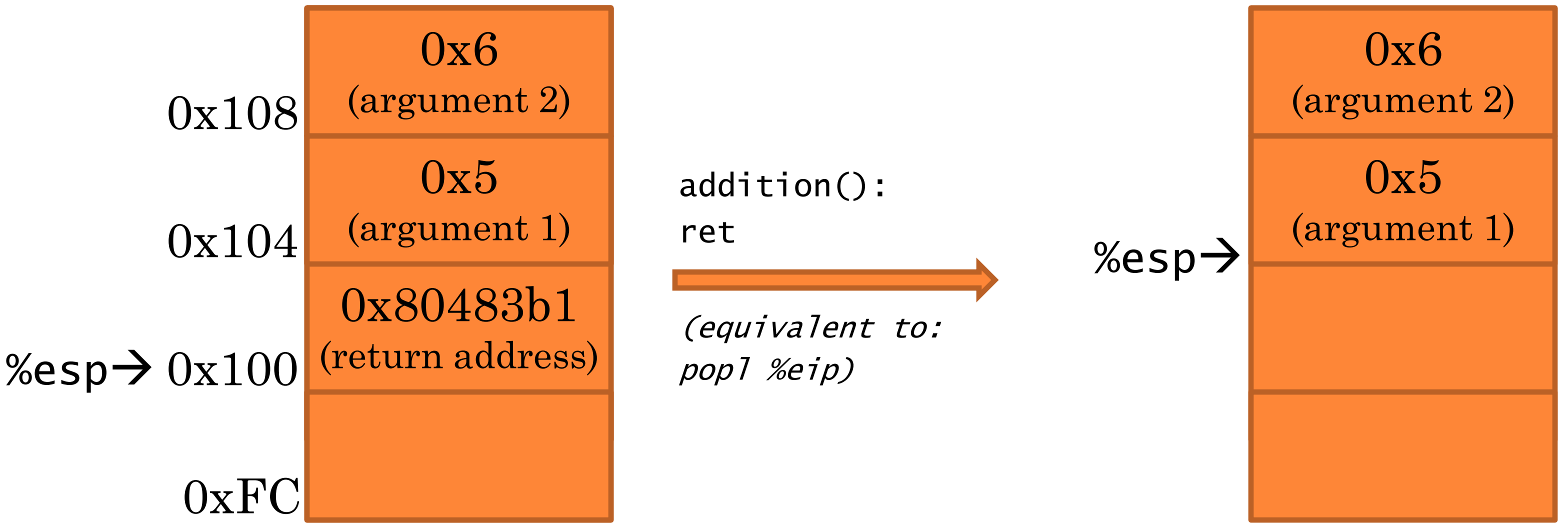
- Preparing to return from a function




| Before | After |
|---|---|
| %esp = 0xFC<br>%ebp = 0xFC<br>%eip = 0x80483bf | %esp = 0x100<br>%ebp = 0x200<br>%eip = 0x80483c0 |

# Breakdown: Return

- Return from a function




| **Before** | **After** |
|---|---|
| %esp = 0xFC<br>%ebp = 0x200<br>%eip = 0x80483c0 | %esp = 0x104<br>%ebp = 0x200<br>%eip = 0x80483b1 |

# Stacks and Stuff on x86_64

- Arguments (≤ 6) are passed via registers
  - `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
  - Extra arguments passed via stack!
    - IA32 stack knowledge still matters!
- Don't need `%ebp` as the base pointer
  - Compilers are smarter now
- Overall less stack use
  - Potentially better performance

# And Floating Point?

- Floating point arguments are complicated
  - Out of the scope of this course
  - Some chips have a separate floating point stack
- Example of complication
  - x86_64 stack on function entry needs to be 16 byte aligned for floating point
  - And other potential issues you shouldn’t worry about

# BUFLAB

- A series of exercises asking you to overflow the stack and change execution
  - You do this with inputs that are super long and write over key stack values
  - **Incorrect inputs will not hurt your score**
- Seminal paper on stack corruption
  - *Smashing the Stack for Fun and Profit*

# Basic Approach

- Examine the provided C code/ disassembly
  - Disassembling
    - `> objdump -d bufbomb > outfile`
  - Don't forget that GDB is still used for this lab!
- Find out how long to make your inputs
- Write exploits to divert program execution
- Profit

# Buflab Tools

- `./makecookie` *`andrewID`*
  - Makes a unique “cookie” based on your Andrew ID
- `./hex2raw`
  - Use the hex generated from assembly to pass raw strings into bufbomb
  - Use with –n in the last stage
- `./bufbomb -t` *`andrewID`*
  - The actual program to attack
  - Always pass in with your Andrew ID so your score is logged
  - Use with –n in the last stage

# How to Input Answers

- Put your byte code exploit into a text file
  - Then feed it through `hex2raw`
- Later stages: write (corruption) assembly
  - Compiling
    - `> gcc -m32 -c` *`example.s`*
  - Get the byte codes
    - `> objdump -d` *`example.o`* `>` *`outfile`*
  - Then feed it through `hex2raw`

# Ways to Feed Byte Codes

- Option 1: Pipes
  - `> cat exploitfile | ./hex2raw | ./bufbomb -t andrewID`
- Option 2: Redirects
  - `> ./hex2raw < exploitfile > exploit-rawfile`
  - `> ./bufbomb -t andrewID < exploit-rawfile`
- Option 3: Redirects in GDB
  - `> gdb bufbomb`
  - `(gdb) run -t andrewID < exploit-rawfile`

# Potential Points of Failure

- Don’t use byte value 0A in your exploit
  - ASCII for newline
  - `Gets()` will terminate early if it sees this
- Multiple exploits submitted for the same level always takes the latest submission
  - So if you pass correctly once, but accidently pass the wrong exploit later, just pass the correct one again
- If you manage to execute your exploit….
  - GDB will say weird things
    - “Can’t access memory…” etc.
    - Just ignore it and keep going
- Don’t forget the –n flag on the last level

# BUFLAB

- The writeup contains all the lab knowledge
  - How to use the tools
  - How to write corruption code
  - Even tells you how to solve the level (at a high level)!
- Please don't ask questions answered by the writeup
  - Or I will make this sad face: TT_TT
- The writeup is on Autolab
  - Couple links down from the handout

# A Lesson on Endianness

- We're working with little endian
  - Least significant byte is at the lower address

| | |
|---|---|
| *Higher addresses*<br>… | Caller stack frame |
| Return Address | |
| Saved `%ebp` | ← `%ebp` |
| Saved `%ebx` | |
| Canary | ← Potential way to detect stack corruption |
| *MSB* `[7] [6] [5] [4]` | buf string<br>(each char is a byte) |
| `[3] [2] [1] [0]` *LSB* | |
| …<br>*Lower addresses* | |

# Example of Endian in Buf Lab

- Example byte code input:
  - 01 02 03 04
    05 06 07 08
    09 AA BB CC
    55 44 04 08
- Little Endian
  - Addresses will look as they normally do when they end up on the stack.
  - Here, value 0x08044455 reads as 0x08044455 on the stack.

| | |
|---|---|
| *Higher addresses* … | |
| 08 04 44 55 | ← Potentially overwritten return address |
| CC BB AA 09 | |
| 08 07 06 05 | |
| 04 03 02 01 | ← Input string address |
| … *Lower addresses* | |

# Miscellany but Necessary

- Canaries
  - Attempts to detect overrun buffers
  - Sits at the end of the buffer (array)
    - If the array overflows, *hopefully* we detect this with a change in the canary value….
- NOP sleds
  - The nop instruction means "no-op/ no operation"
    - In computer architecture it's like "pipeline bubbles"
  - Used to "pad" instructions (or exploits!)
    - Place your exploits at the end of the nop sled
    - Allows you to be "sloppier" in providing the return address of your exploit

# Demo Time (if class isn't over yet)

- Walking stacks
- Byte code format
- Byte code feeding
- Example assembly
- Compiling assembly
  - *Not* assembling
- Assembly to byte code

# Stolen Credits & Questions Slide

- xkcd: Tabletop Roleplaying
- StackOverflow: Supporting Recursion
- StackOverflow: Direction of Stack Growth
- Understanding the SPARC Architecture
- CS:APP p. 220 – Stack Frame Structure
- *Smashing the Stack for Fun and Profit*
- CS:APP p.262 – NOP sleds
- CS:APP p.263 – Stack Frame with a canary
- Double Mocha Latte Picture