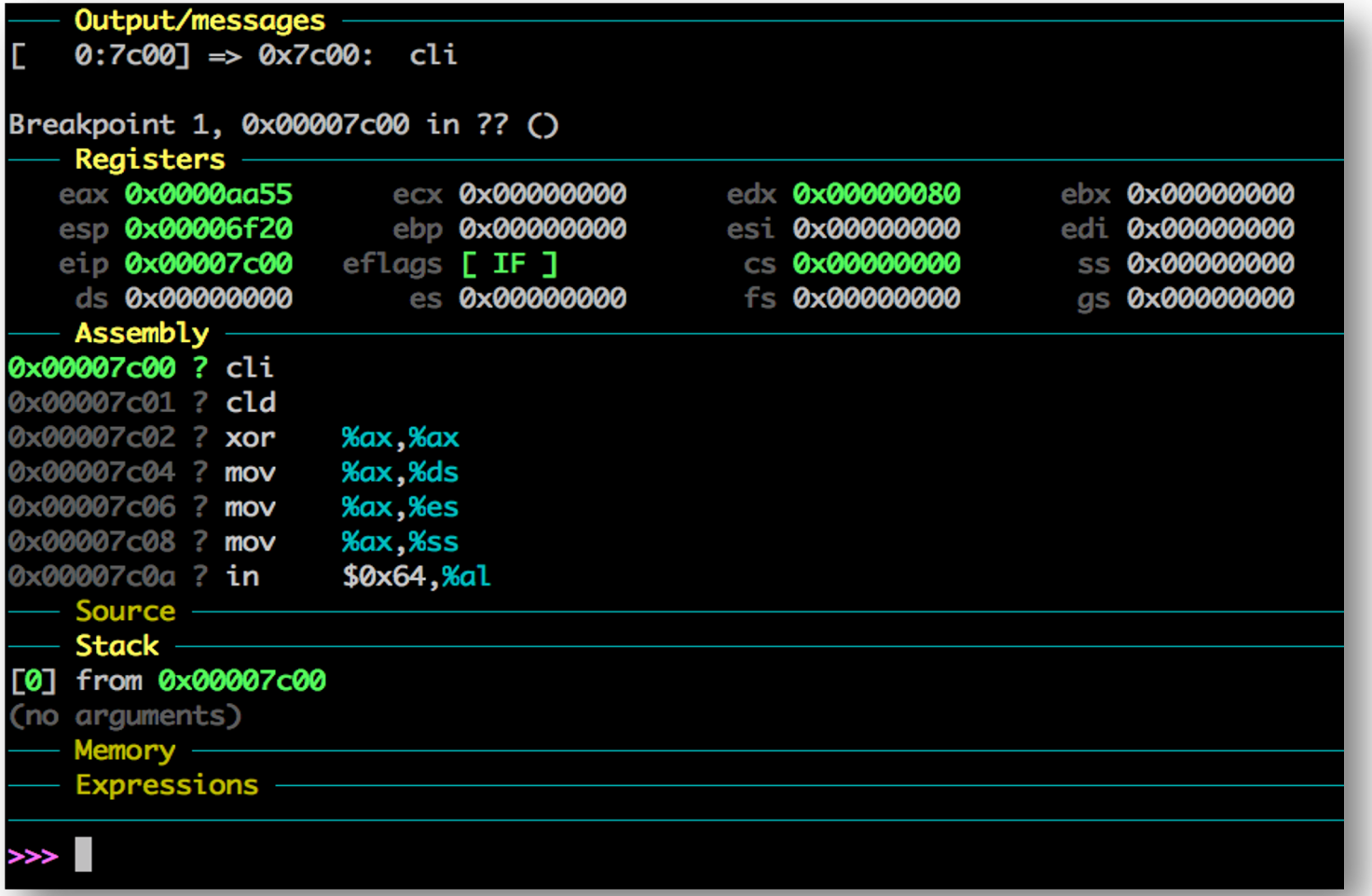
CS 444/544 OS II
Lab Tutorial #2

Booting and Calling Covention

Prof. Sibin Mohan | Spring 2022

Adapted from content originally created by: Prof. Yeongjin Jang

Lab Setup Check



Contents

FOLLOWING THE BOOTING SEQUENCE

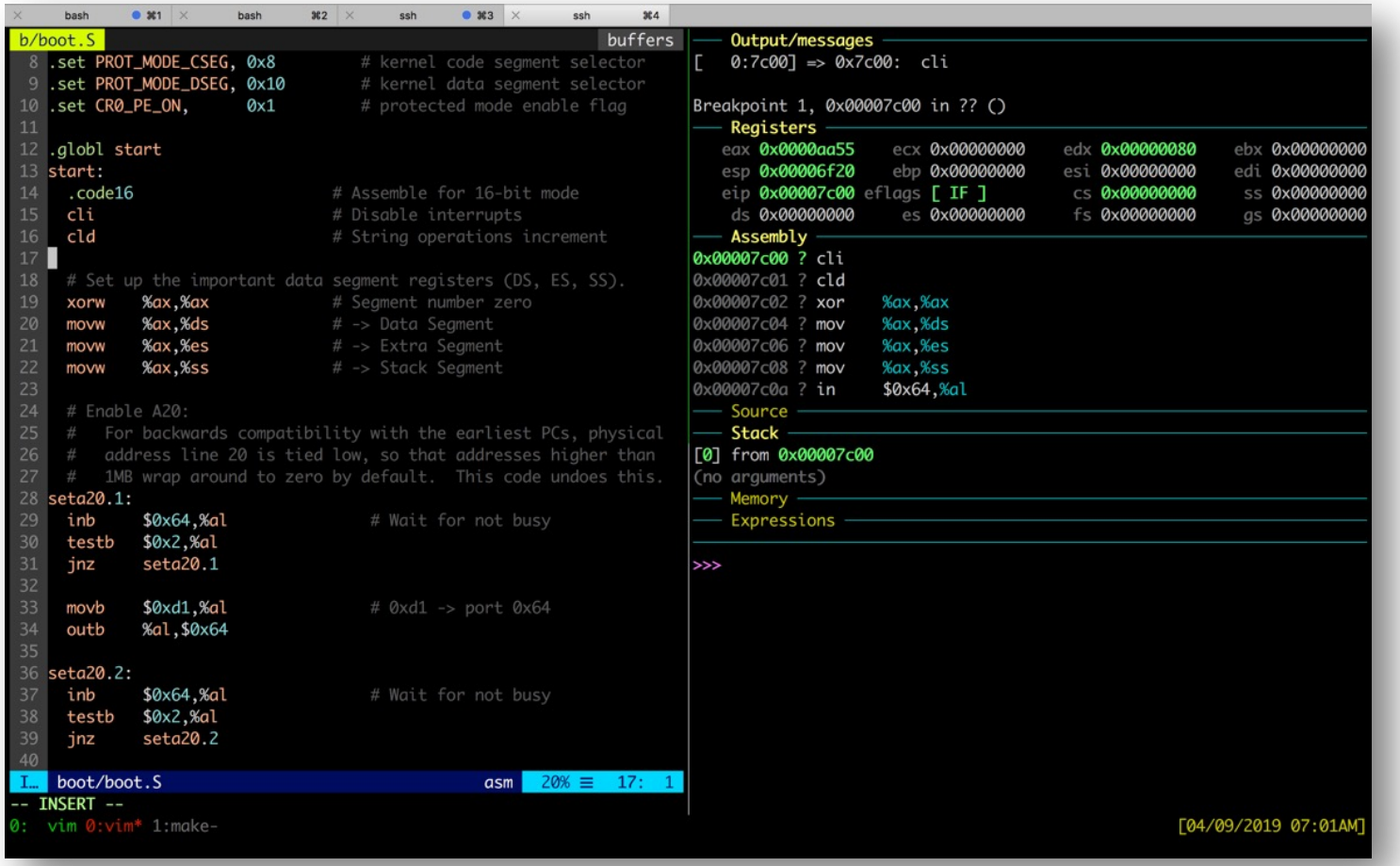
STACK AND CALLING CONVENTION

Exercise 3

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the *x/i* command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Exercise 3: How?

- Use tmux
  - open boot/boot.S and gdb
  - at the same time
- Use ni,si, breakpoint



Exercise 3: Enabling Protected Mode

- `PROT_MODE_CSEG 0x8`
- **`bootmain()`** in **boot/main.c**

```
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call bootmain
```

Exercise 3: In boot/main.c → `bootmain()`

```c
void
bootmain(void)
{
    struct Proghdr *ph, *eph;

    // read 1st page off disk
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC)
        goto bad;

    // load each program segment (ignores ph flags)
    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++)
        // p_pa is the load address of this segment (as well
        // as the physical address)
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
```

```
# Set up the stack pointer and call into C.
movl    $start, %esp
call bootmain
```

```
0x00007c45 ? call   0x7d0a
0x00007c4a ? jmp    0x7c4a
0x00007c4c ? add    %al,(%eax)
0x00007c4e ? add    %al,(%eax)
0x00007c50 ? add    %al,(%eax)
0x00007c52 ? add    %al,(%eax)
0x00007c54 ? (bad)
```

Exercises 4-6

- Ex 4: Understand why **pointer.c** works like that
  - Read about **ELF header** and the **ELF file**
  - https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

- Ex 5: Use **si, ni** to follow the instructions after changing **0x7c00**
  - to other values e.g., **0x6b00** or something else..

```
$(OBJDIR)/boot/boot: $(BOOT_OBJS)
    @echo + ld boot/boot
    $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
    $(V)$(OBJDUMP) -S $@.out >$@.asm
    $(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
    $(V)perl boot/sign.pl $(OBJDIR)/boot/boot
```

- Ex 6: practice **gdb** commands

GDB Command for Reading Memory

- x**100**/w**x** [address or register] translates to
  - Examine
  - **100** values
  - sized as **word** (**w**, **4 bytes**)
    - b → byte
    - g → 8 bytes
  - **In hexadecimal (x)**
    - d → decimal

```
>> x/100wx 0x7c00
0x7c00: 0xc031fcfa      0xc08ed88e      0x64e4d08e      0xfa75
0x7c10: 0x64e6d1b0      0x02a864e4      0xdfb0fa75      0x010f
0x7c20: 0x0f7c6416      0x8366c020      0x220f01c8      0x7c32
0x7c30: 0xb8660008      0xd88e0010      0xe08ec08e      0xd08e
0x7c40: 0x007c00bc      0x00c0e800      0xfeeb0000      0x0000
0x7c50: 0x00000000      0x0000ffff      0x00cf9a00      0x0000
0x7c60: 0x00cf9300      0x7c4c0017      0xba550000      0x0000
0x7c70: 0x83ece589      0x403cc0e0      0xc35df875      0x57e5
0x7c80: 0x0c5d8b53      0xfffe1e8       0x01f2baff      0x01b0
0x7c90: 0xc3b60fee      0x0feef3b2      0xf4b2c7b6      0xb2d8
0x7ca0: 0x10e8c1f5      0xeec0b60f      0xb218ebc1      0x83d8
0x7cb0: 0xb0eee0c8      0xeef7b220      0xffffade8      0x087d
0x7cc0: 0x000080b9      0x01f0ba00      0xf2fc0000      0x5d5f
0x7cd0: 0xe58955c3      0x0c7d8b57      0x10758b56      0x085d
0x7ce0: 0x0109eec1      0xe38146df      0xfffffe00      0x1273
0x7cf0: 0x81534656      0x000200c3      0xff7ee800      0x5a58
0x7d00: 0x658deaeb      0x5f5e5bf4      0x8955c35d      0x6a53
0x7d10: 0x10006800      0x00680000      0xe8000100      0xffff
0x7d20: 0x810cc483      0x0100003d      0x4c457f00      0xa138
0x7d30: 0x0001001c      0x0000988d      0xb70f0001      0x0100
0x7d40: 0x05e0c100      0x3903348d      0xff1673f3      0xc383
0x7d50: 0xf473ff20      0xe8ec73ff      0xffffff75      0xeb0c
0x7d60: 0x1815ffe6      0xba000100      0x00008a00      0xff8a
0x7d70: 0xb8ef66ff      0xffff8e00      0xfeebef66      0x0000
0x7d80: 0x00000000      0x00000000      0x00000000      0x0000
```

Exercise 7: Virtual Memory

- **`0xf0000000`** == KERNBASE

- Virtual address **`0xf0000000 ~ 0xffffffff`**
  - Access **physical address** at:
    (Virtual address – KERNBASE)
- E.g.,
  - **`0xf0123456`** → **`0x123456`**
  - **`0xf0000001`** → **`0x1`**

Exercise 8

- Read **lib/printfmt.c**, for **`vprintfmt()`**
- Look at cases '**x**' and '**u**'
  - as an example of **hexadecimal** and **decimal**
- Implement the case '**o**'
  - Similar to 'x' and 'u'
  - It's easy!

Must understand how **stack** works in x86 architecture

Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the *test_backtrace*` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words? NOTE. you'll have to manually translate all breakpoint and memory addresses to linear addresses.

Exercises 9 – 11 | Stack Backtrace

Function calls in x86

In **kern/init.c**

```
// Test the stack backtrace function (lab 1 only)
test_backtrace(5);
```

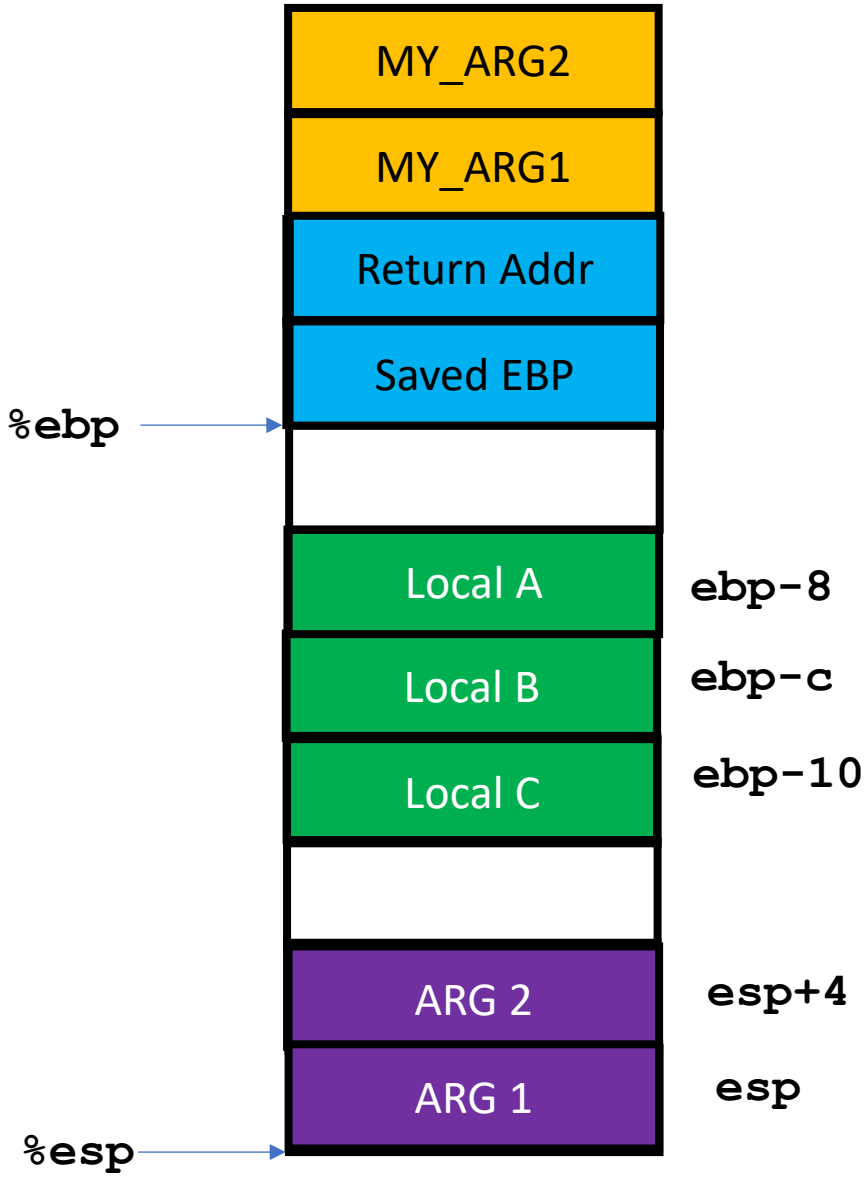
```
10 // Test the stack backtrace function (lab 1 only)
11 void
12 test_backtrace(int x)
13 {
14     cprintf("entering test_backtrace %d\n", x);
15     if (x > 0)
16         test_backtrace(x-1);
17     else
18         mon_backtrace(0, 0, 0);
19     cprintf("leaving test_backtrace %d\n", x);
20 }
```

`test_backtrace(5)` → `test_backtrace(4)` → `test_backtrace(3)` → `2` → `1` → `mon_backtrace(0,0,0)`

How does this recursion work in the x86 architecture?

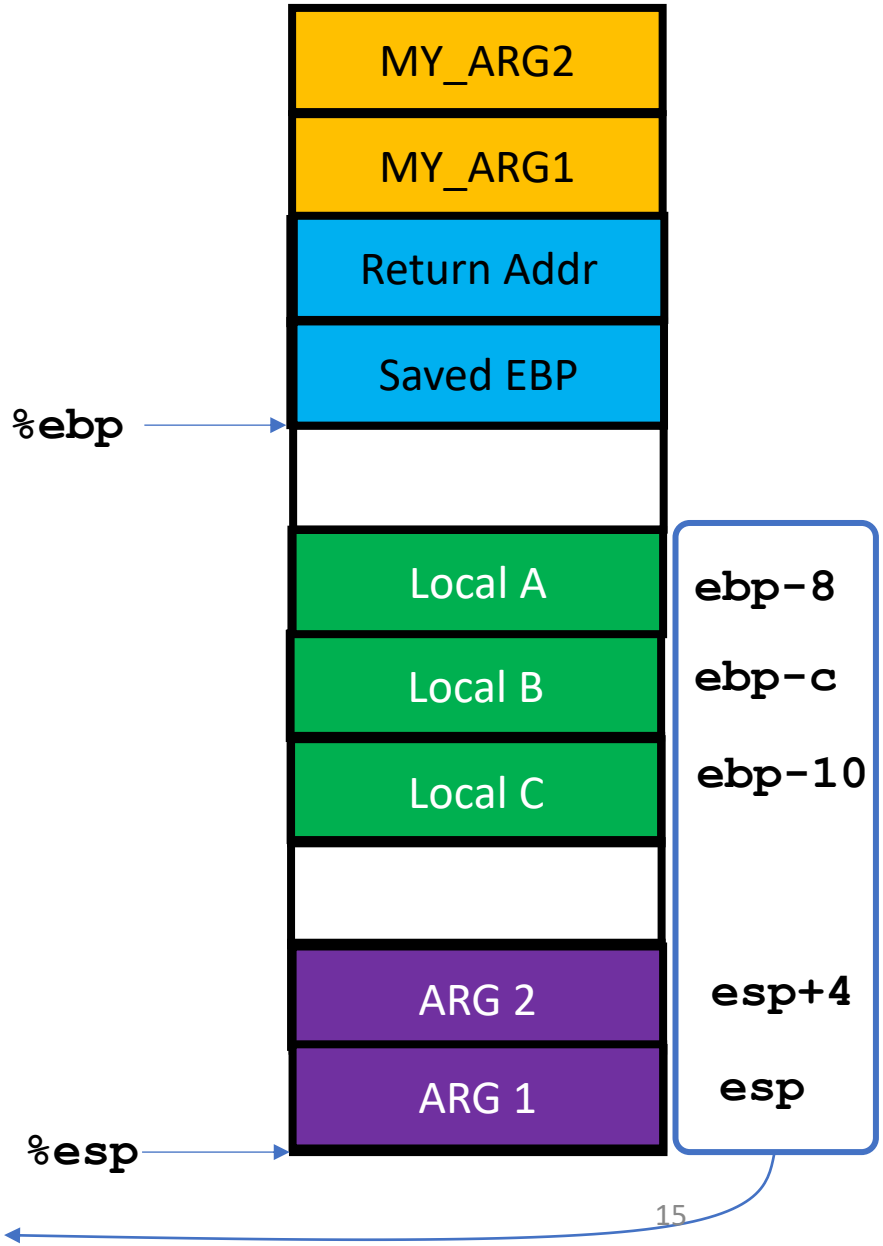
x86 Stack

- All **local variables** are stored in the **stack**
- A **function call** creates a **new stack**
  - Start with **`esp`** ends with **`ebp`**
- Grows **downward**!
  - Push(A)→ subtract 4 from **`esp`** and store A there
  - Pop→ get the value at **`esp`** and add **4** to **`esp`**



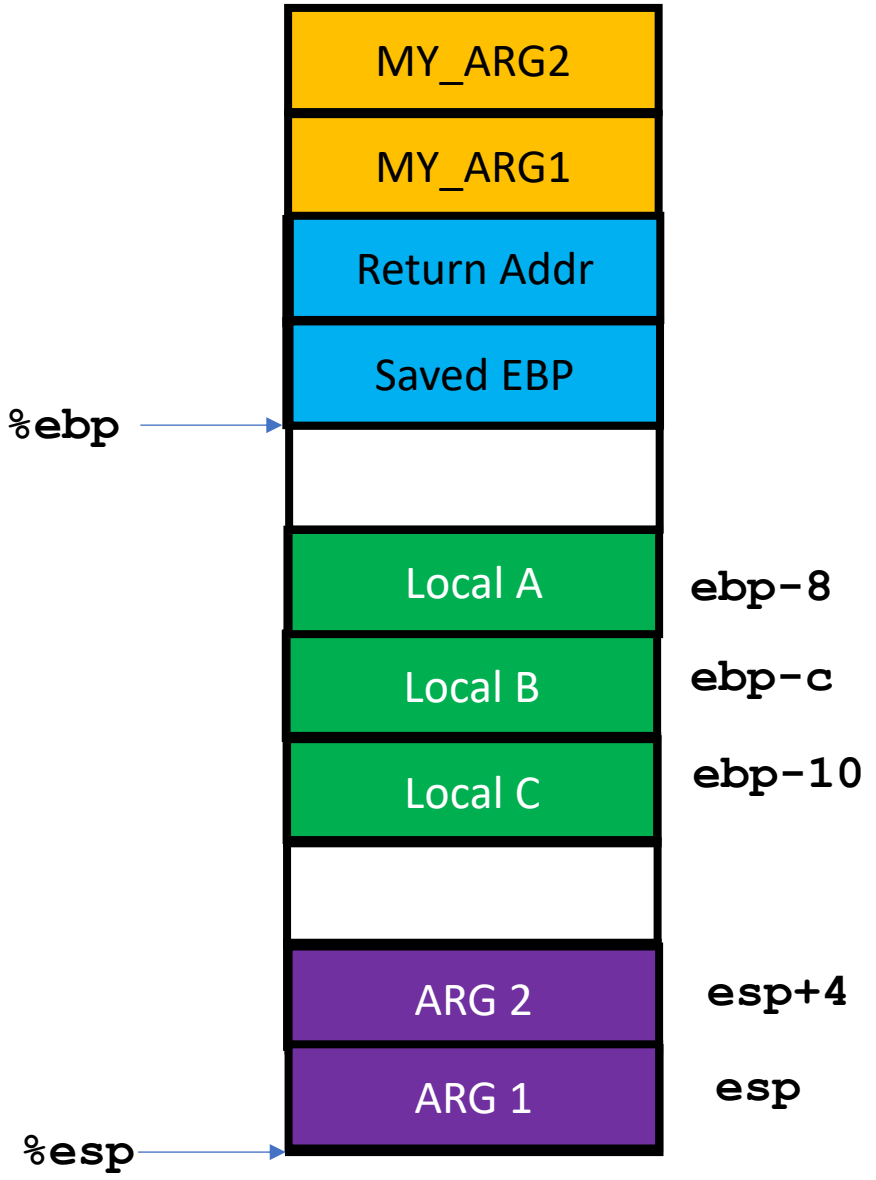
x86 Stack

- All **local variables** are stored in the **stack**
- A **function call** creates a **new stack**
  - Start with **`esp`** ends with **`ebp`**
- Grows **downward**!
  - Push(A)→ subtract 4 from **`esp`** and store A there
  - Pop→ get the value at **`esp`** and add **4** to **`esp`**



Function call example

```
my_function(MY_ARG1, MY_ARG2) {
   int A;
   int B;
   int C;
   other_function(ARG1, ARG2)
}
```



How does x86 manage the stack?

- Let's debug calling **`test_backtrace()`**
- Set the breakpoint at ***`i386_init`**

```
Breakpoint 1, i386_init () at kern/init.c:24
24      {
── Registers ──
   eax 0xf010002f    ecx 0x00000000    edx 0x0000009d    ebx 0x00010094
eflags [ PF SF ]     cs 0x00000008     ss 0x00000010     ds 0x00000010
── Assembly ──
0xf010009d i386_init+0 push   %ebp
0xf010009e i386_init+1 mov    %esp,%ebp
0xf01000a0 i386_init+3 sub    $0x18,%esp
0xf01000a3 i386_init+6 mov    $0xf0112940,%eax
── Source ──
19      cprintf("leaving test_backtrace %d\n", x);
20 }
21
22 void
23 i386_init(void)
24 {
25      extern char edata[], end[];
26
27      // Before doing anything else, complete the ELF loading process.
28      // Clear the uninitialized global data (BSS) section of our program.
29      // This ensures that all static/global variables start out zero.
── Stack ──
[0] from 0xf010009d in i386_init+0 at kern/init.c:24
(no arguments)
── Memory ──
── Expressions ──
>>>
1:  gdb 0:make- 1:gdb*
```

```
+ symbol-file obj/kern/kernel
>>> b *i386_init
Breakpoint 1 at 0xf010009d: file kern/init.c, line 24.
>>> c
```

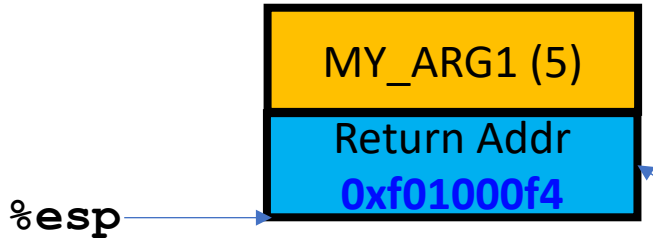
How does x86 manage the stack?

- Let's debug calling **`test_backtrace()`**
- Set the breakpoint at ***`i386_init`**
- Examine the instructions


```
gdb-peda$ x/25i $pc
=> 0xf01000a6 <i386_init>:      push   %ebp
   0xf01000a7 <i386_init+1>:    mov    %esp,%ebp
   0xf01000a9 <i386_init+3>:    push   %ebx
   0xf01000aa <i386_init+4>:    sub    $0x8,%esp
   0xf01000ad <i386_init+7>:
    call   0xf01001bc <__x86.get_pc_thunk.bx>
   0xf01000b2 <i386_init+12>:   add    $0x11256,%ebx
   0xf01000b8 <i386_init+18>:   mov    $0xf0113060,%edx
   0xf01000be <i386_init+24>:   mov    $0xf01136a0,%eax
   0xf01000c4 <i386_init+30>:   sub    %edx,%eax
   0xf01000c6 <i386_init+32>:   push   %eax
   0xf01000c7 <i386_init+33>:   push   $0x0
   0xf01000c9 <i386_init+35>:   push   %edx
   0xf01000ca <i386_init+36>:   call   0xf010179a <memset>
   0xf01000cf <i386_init+41>:   call   0xf0100611 <cons_init>
   0xf01000d4 <i386_init+46>:   add    $0x8,%esp
   0xf01000d7 <i386_init+49>:   push   $0x1aac
   0xf01000dc <i386_init+54>:   lea    -0xf6f1(%ebx),%eax
   0xf01000e2 <i386_init+60>:   push   %eax
   0xf01000e3 <i386_init+61>:   call   0xf0100b86 <cprintf>
   0xf01000e8 <i386_init+66>:   movl   $0x5,(%esp)
   0xf01000ef <i386_init+73>:   call   0xf0100040 <test_backtrace>
   0xf01000f4 <i386_init+78>:   add    $0x10,%esp
   0xf01000f7 <i386_init+81>:   sub    $0xc,%esp
   0xf01000fa <i386_init+84>:   push   $0x0
   0xf01000fc <i386_init+86>:   call   0xf01009ce <monitor>
```

How does x86 manage the stack?

- Let's debug calling **`test_backtrace()`**
- Set the breakpoint at ***`i386_init`**
- Examine the instructions

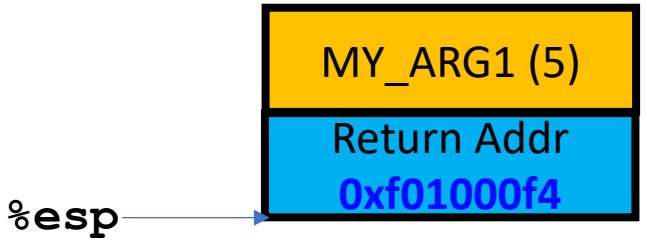


- The call instruction,
  - Push addr of next instr → to return after func()
  - Jump to target

```
gdb-peda$ x/25i $pc
=> 0xf01000a6 <i386_init>:      push   %ebp
   0xf01000a7 <i386_init+1>:    mov    %esp,%ebp
   0xf01000a9 <i386_init+3>:    push   %ebx
   0xf01000aa <i386_init+4>:    sub    $0x8,%esp
   0xf01000ad <i386_init+7>:
    call   0xf01001bc <__x86.get_pc_thunk.bx>
   0xf01000b2 <i386_init+12>:   add    $0x11256,%ebx
   0xf01000b8 <i386_init+18>:   mov    $0xf0113060,%edx
   0xf01000be <i386_init+24>:   mov    $0xf01136a0,%eax
   0xf01000c4 <i386_init+30>:   sub    %edx,%eax
   0xf01000c6 <i386_init+32>:   push   %eax
   0xf01000c7 <i386_init+33>:   push   $0x0
   0xf01000c9 <i386_init+35>:   push   %edx
   0xf01000ca <i386_init+36>:   call   0xf010179a <memset>
   0xf01000cf <i386_init+41>:   call   0xf0100611 <cons_init>
   0xf01000d4 <i386_init+46>:   add    $0x8,%esp
   0xf01000d7 <i386_init+49>:   push   $0x1aac
   0xf01000dc <i386_init+54>:   lea    -0xf6f1(%ebx),%eax
   0xf01000e2 <i386_init+60>:   push   %eax
   0xf01000e3 <i386_init+61>:   call   0xf0100b86 <cprintf>
   0xf01000e8 <i386_init+66>:   movl   $0x5,(%esp)
   0xf01000ef <i386_init+73>:   call   0xf0100040 <test_backtrace>
   0xf01000f4 <i386_init+78>:   add    $0x10,%esp
   0xf01000f7 <i386_init+81>:   sub    $0xc,%esp
   0xf01000fa <i386_init+84>:   push   $0x0
   0xf01000fc <i386_init+86>:   call   0xf01009ce <monitor>
```

How does x86 manage the stack?

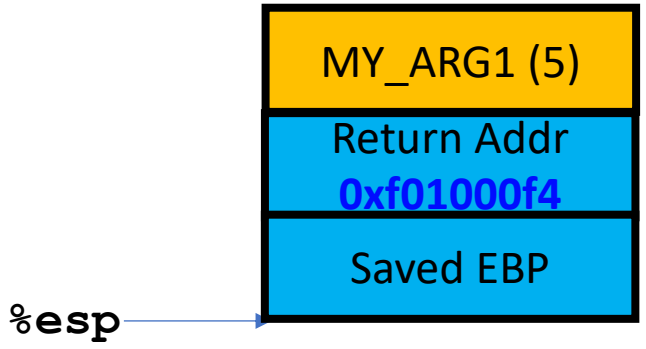
- Inside **`test_backtrace()`**




```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```

How does x86 manage the stack?

- Inside **`test_backtrace()`**

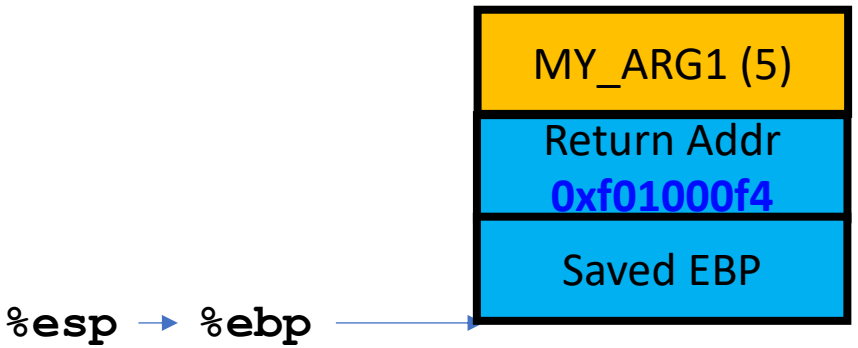



```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```

How does x86 manage the stack?

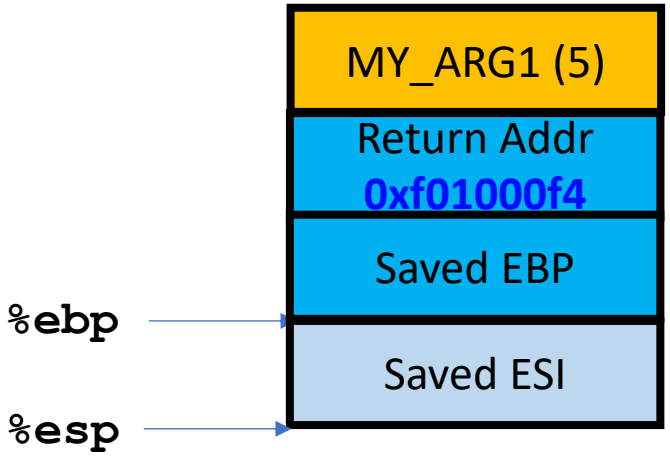
- Inside **`test_backtrace()`**

```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```



How does x86 manage the stack?

- Inside **`test_backtrace()`**

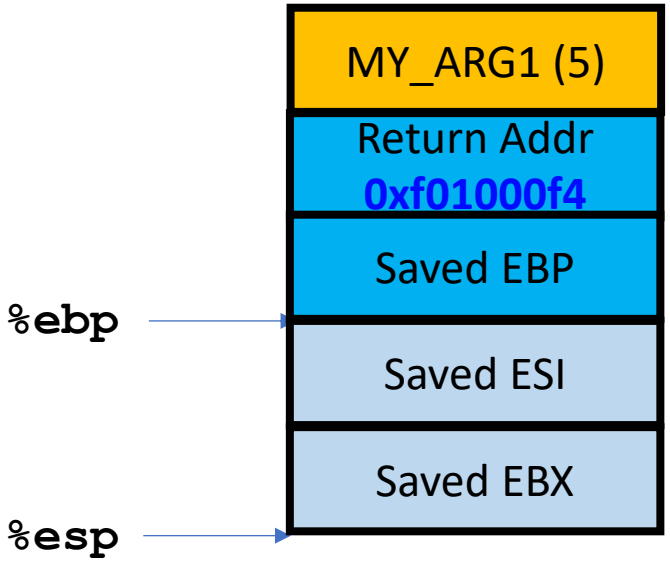



```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```

How does x86 manage the stack?

- Inside **`test_backtrace()`**

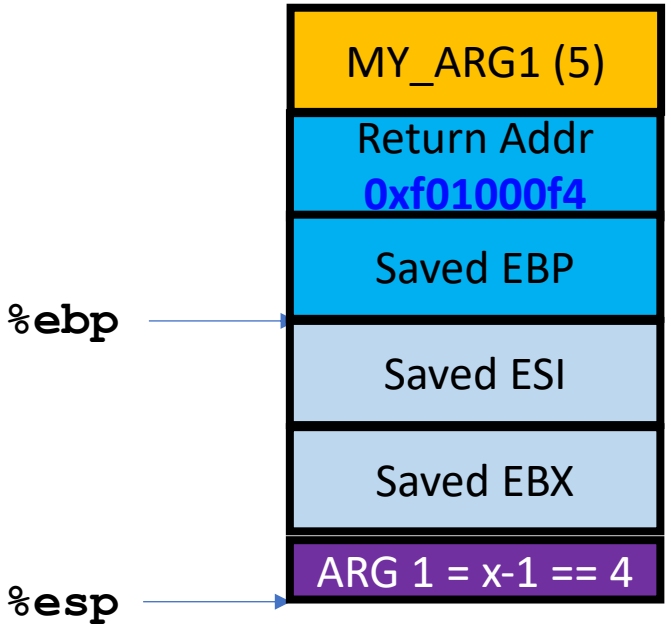
```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```



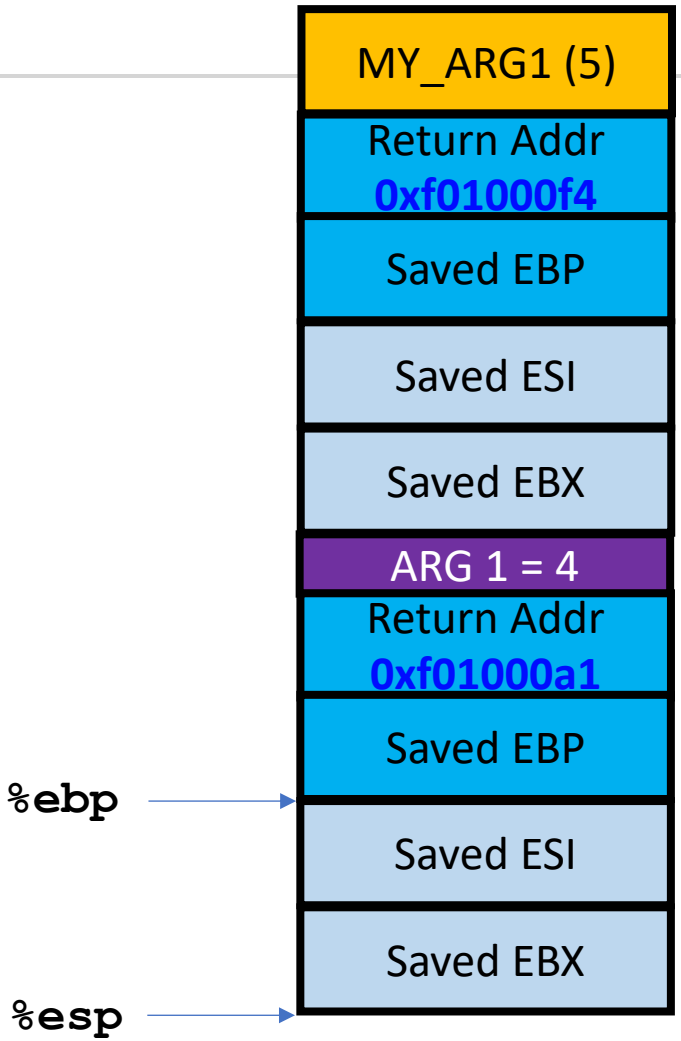
How does x86 manage the stack?

- Inside **`test_backtrace(x-1)`**

```
0xf0100098 <+88>:    lea    -0x1(%esi),%eax
0xf010009b <+91>:    push   %eax
0xf010009c <+92>:    call   0xf0100040 <test_backtrace>
```

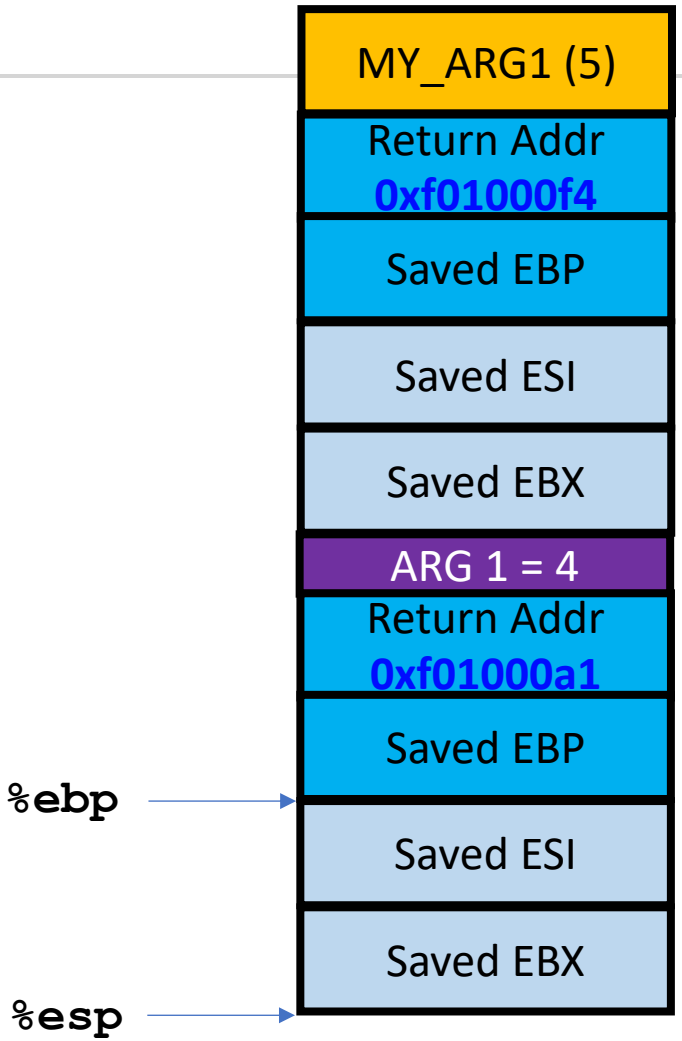


How does x86 manage the stack?




```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```

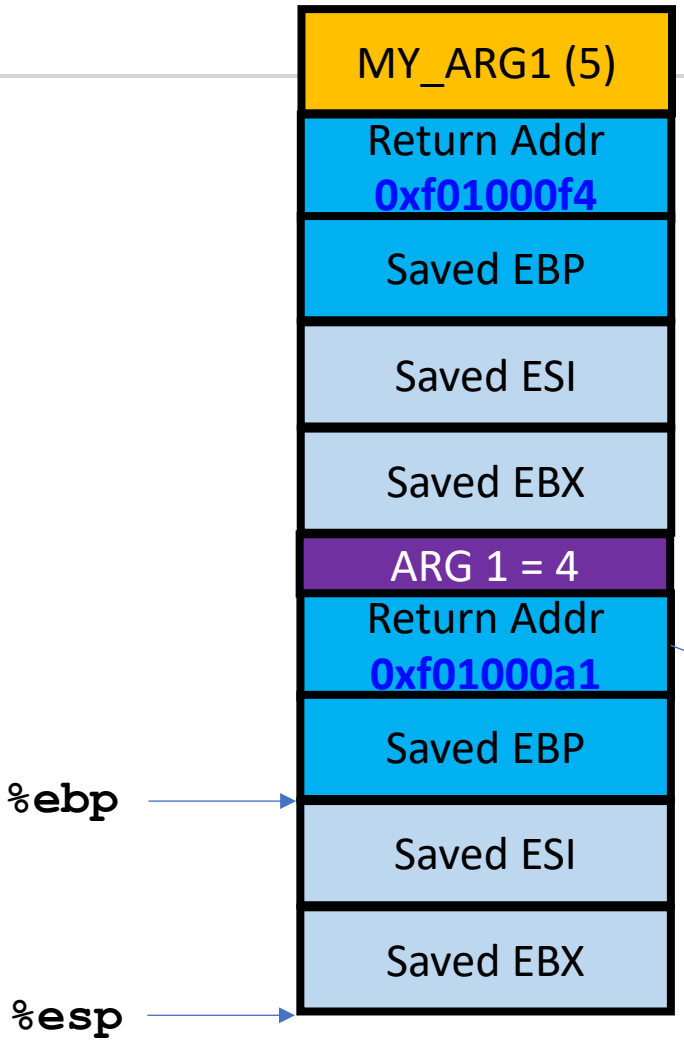
How does x86 manage the stack?




```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```

```
0xf0100098 <+88>:    lea    -0x1(%esi),%eax
0xf010009b <+91>:    push   %eax
0xf010009c <+92>:    call   0xf0100040 <test_backtrace>
0xf01000a1 <+97>:    add    $0x10,%esp
```

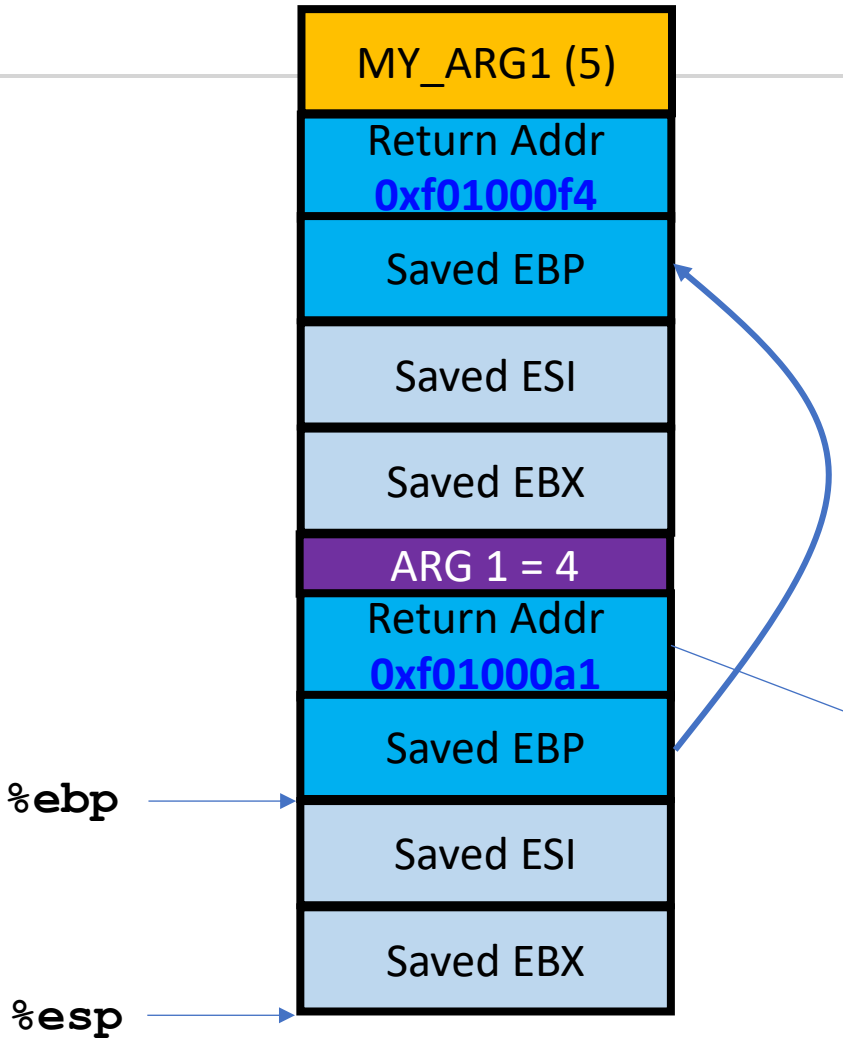
How does x86 manage the stack?




```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```

```
0xf0100098 <+88>:    lea    -0x1(%esi),%eax
0xf010009b <+91>:    push   %eax
0xf010009c <+92>:    call   0xf0100040 <test_backtrace>
0xf01000a1 <+97>:    add    $0x10,%esp
```

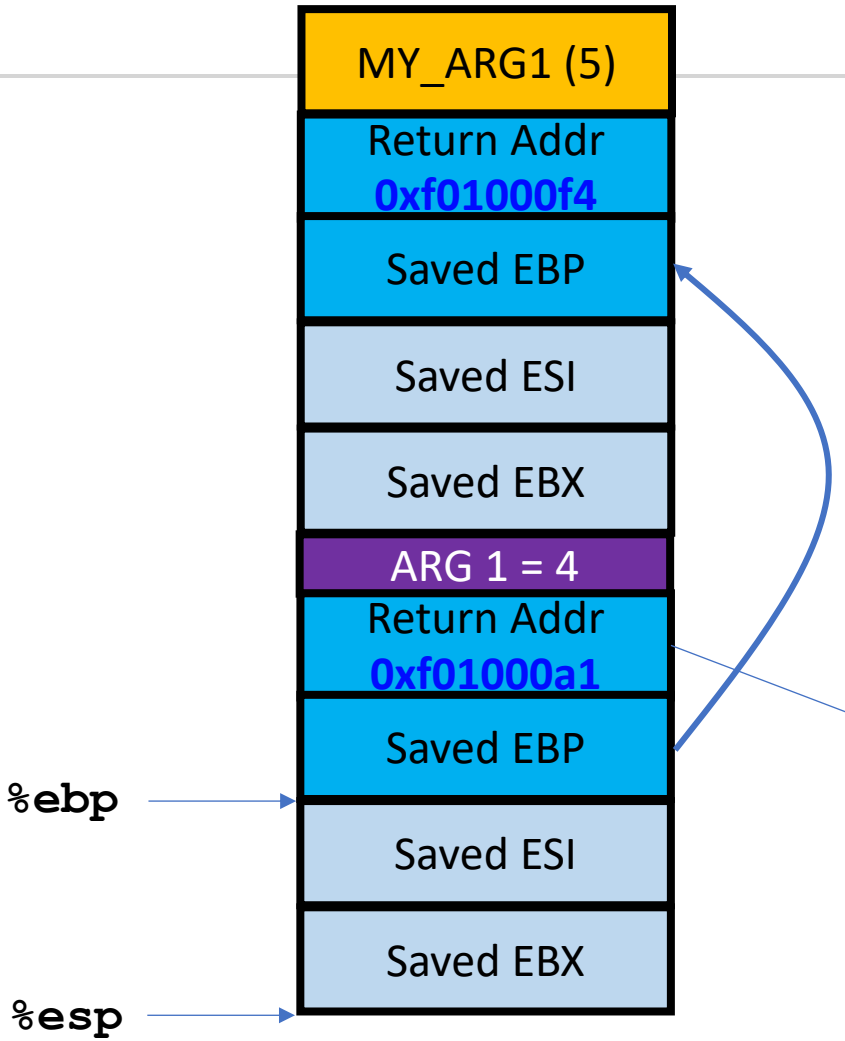
How does x86 manage the stack?




```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```

```
0xf0100098 <+88>:    lea    -0x1(%esi),%eax
0xf010009b <+91>:    push   %eax
0xf010009c <+92>:    call   0xf0100040 <test_backtrace>
0xf01000a1 <+97>:    add    $0x10,%esp
```

How does x86 manage the stack?



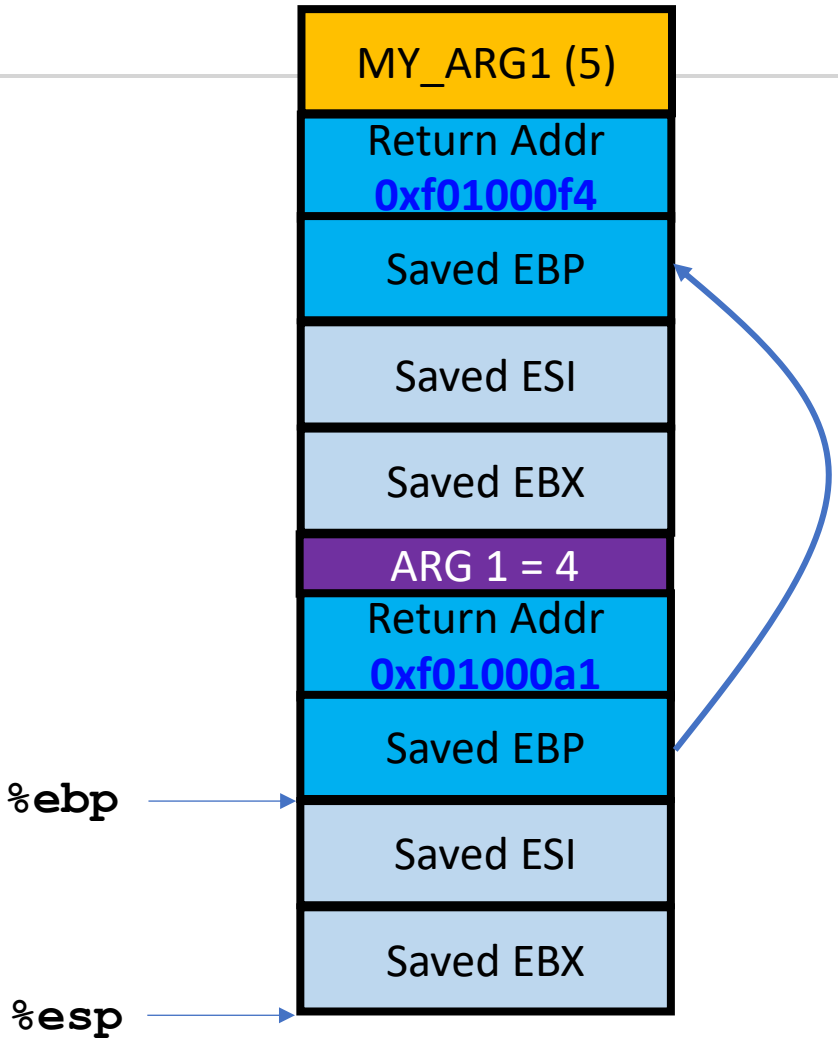

```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
   0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
```

```
0xf0100098 <+88>:    lea    -0x1(%esi),%eax
0xf010009b <+91>:    push   %eax
0xf010009c <+92>:    call   0xf0100040 <test_backtrace>
0xf01000a1 <+97>:    add    $0x10,%esp
```

Repeat until value reaches 0

```
if (x > 0)
    test_backtrace(x-1);
else
    mon_backtrace(0, 0, 0);
```

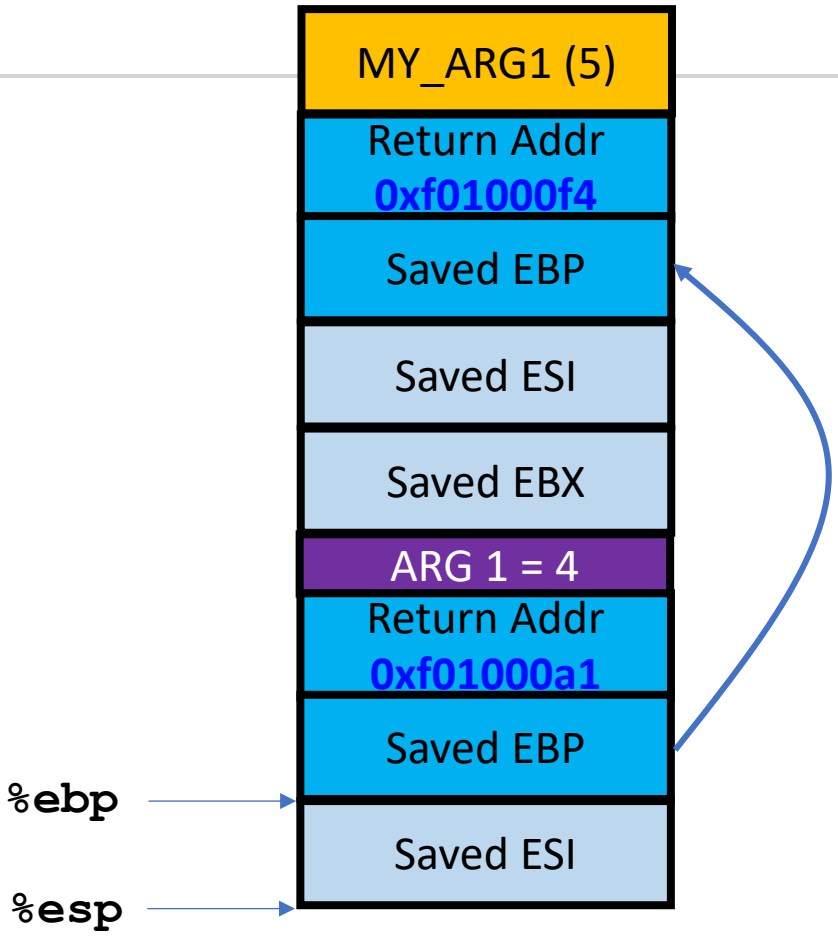
How does x86 manage the stack?




```
0xf0100091 <+81>:    pop    %ebx
0xf0100092 <+82>:    pop    %esi
0xf0100093 <+83>:    pop    %ebp
0xf0100094 <+84>:    ret
```

ret == **`pop %eip`**

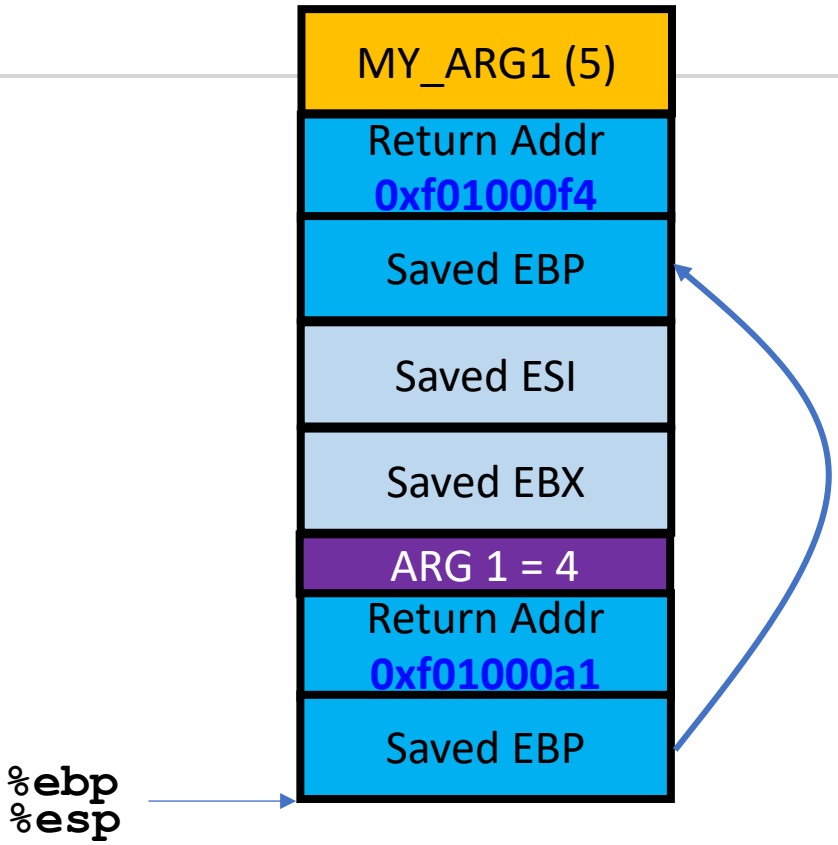
How does x86 manage the stack?




```
0xf0100091 <+81>:    pop    %ebx
0xf0100092 <+82>:    pop    %esi
0xf0100093 <+83>:    pop    %ebp
0xf0100094 <+84>:    ret
```

ret == **`pop %eip`**

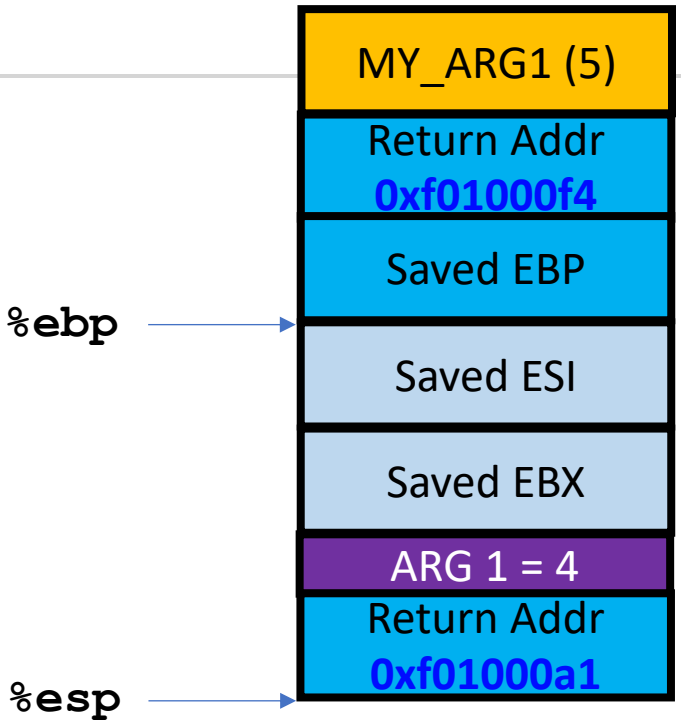
How does x86 manage the stack?




```
0xf0100091 <+81>:    pop    %ebx
0xf0100092 <+82>:    pop    %esi
0xf0100093 <+83>:    pop    %ebp
0xf0100094 <+84>:    ret
```

ret == **`pop %eip`**

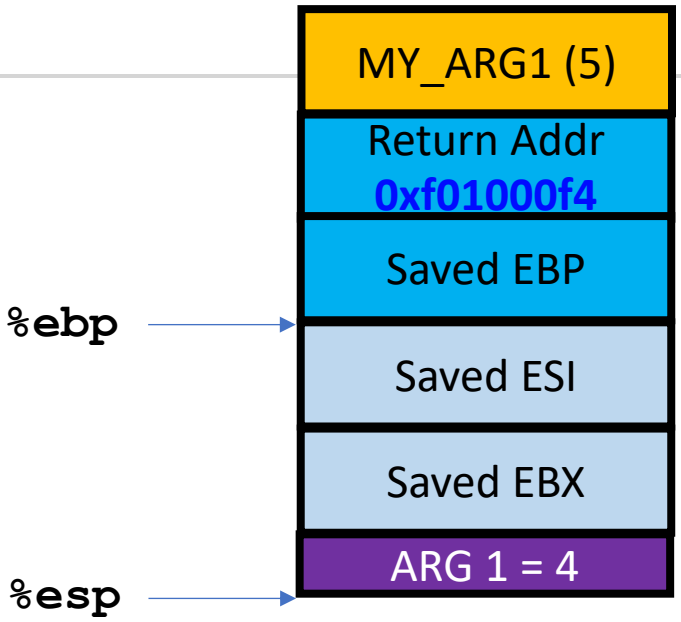
How does x86 manage the stack?




```
0xf0100091 <+81>:    pop    %ebx
0xf0100092 <+82>:    pop    %esi
0xf0100093 <+83>:    pop    %ebp
0xf0100094 <+84>:    ret
```

ret == **pop %eip**

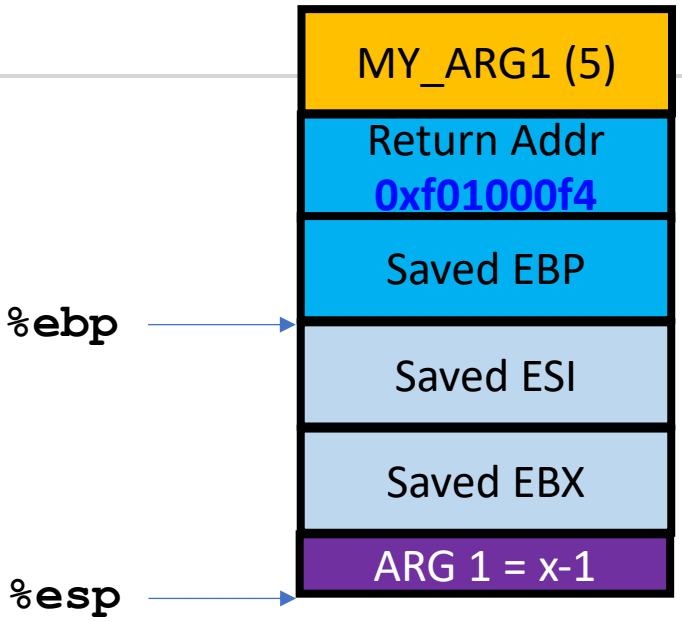
How does x86 manage the stack?




```
0xf0100091 <+81>:    pop    %ebx
0xf0100092 <+82>:    pop    %esi
0xf0100093 <+83>:    pop    %ebp
0xf0100094 <+84>:    ret
```

ret == **`pop %eip`**

How does x86 manage the stack?




```
0xf0100098 <+88>:    lea    -0x1(%esi),%eax
0xf010009b <+91>:    push   %eax
0xf010009c <+92>:    call   0xf0100040 <test_backtrace>
```

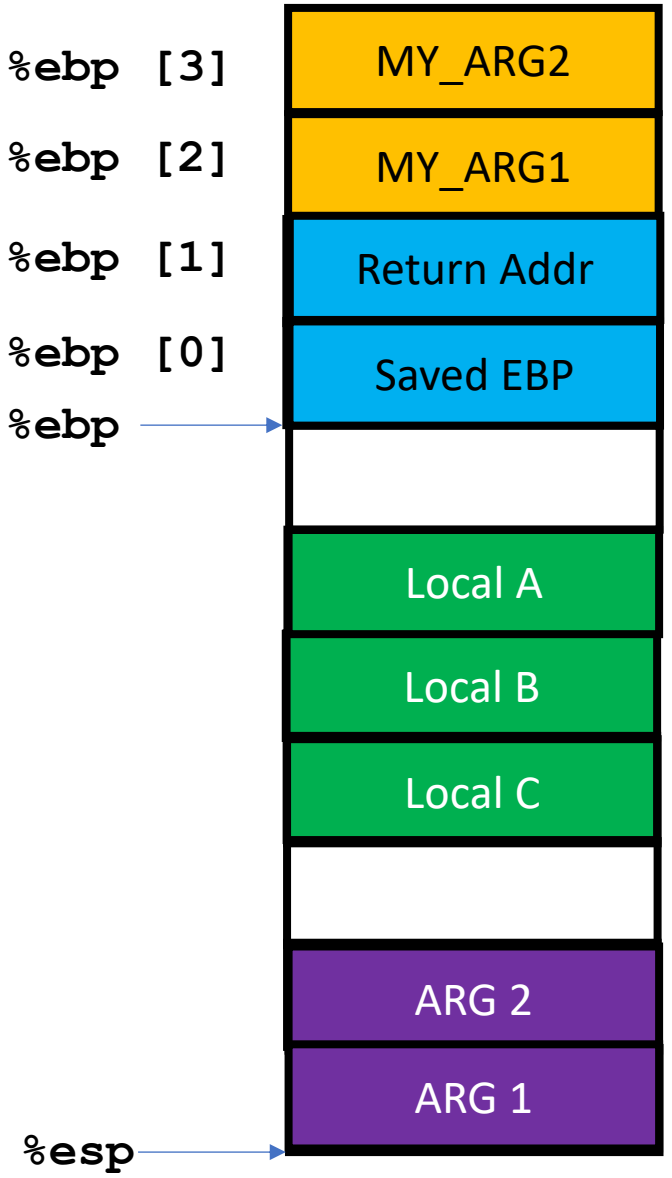
- call **`test_backtrace(x-1)`**

x86 stack

- **`ebp`** points to the **boundary** of the stack (**bottom**)
- **`esp`** points to the **other boundary** of the stack (**top**)

- **`ebp[0]`** stores saved **`ebp`**
- **`ebp[1]`** stores **return address**
- **`ebp[2]`** stores **1st argument**
- **`ebp[3]`** stores 2nd argument

…

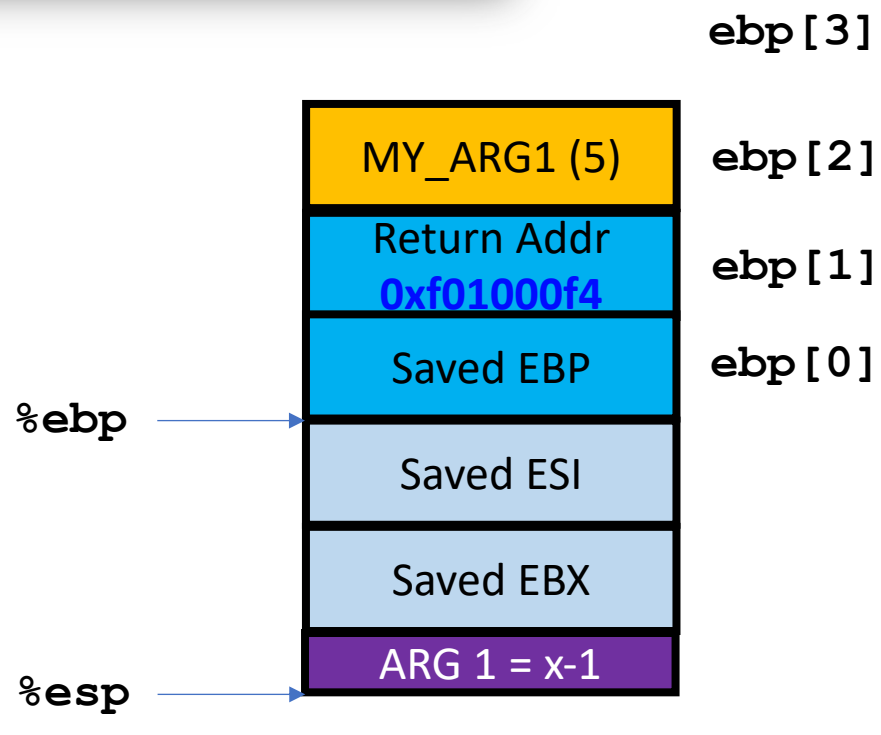


Hint | Exercise 11

```
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
```

```
int *ebp = (int *) read_ebp();
   cprintf("ebp  %08x", ebp)
```

- **eip** == return address
  - ebp[1] – why?
- Args?
  - print **ebp[2 ~ 6]**



Coding Convention

No space after a function name in a call

cprintf(“asdf”) **GOOD**

cprintf (“asdf”) **BAD**

One space after if/for/while/switch

if (a == 1) {. **GOOD**

if(a==1) { **NO**

function_and_variable_names_look_like_this

NoCamelCase

Macros are ALL **UPPERCASE**

e.g., **SEG()**

Coding Convention

1

Pointer types includes a **space before ***

- (uint32_t *) **GOOD**
- (uint32_t*) **NO**

2

Use '**//**' for comment

- All **imported comments** are **/**/**, so we can distinguish yours from those
- FYI, Linux Kernel uses /**/

3

Function with no args

- f(void)
- **not f();**

Coding Convention | Function Definition

- Insert **newline** between the return type and function name

```
int
mon_kerninfo(int argc, char **argv, struct Trapframe *tf)
{
```

- This will make it **easy to find** the **function definition**
- E.g., find the definition of mon_kerninfo would be:

```
[red9057@blue9057-vm-jos (lab1) ~/jos$] grep -nr ^mon_kerninfo
kern/monitor.c:44:mon_kerninfo(int argc, char **argv, struct Trapframe *tf)
```

'^mon_kerninfo' in regexp.