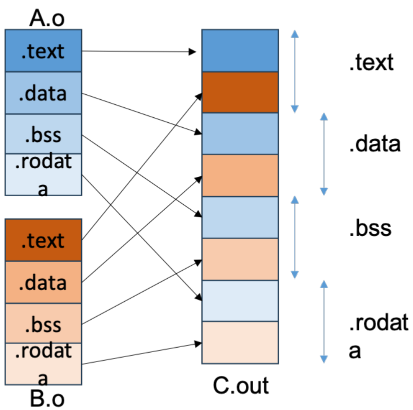
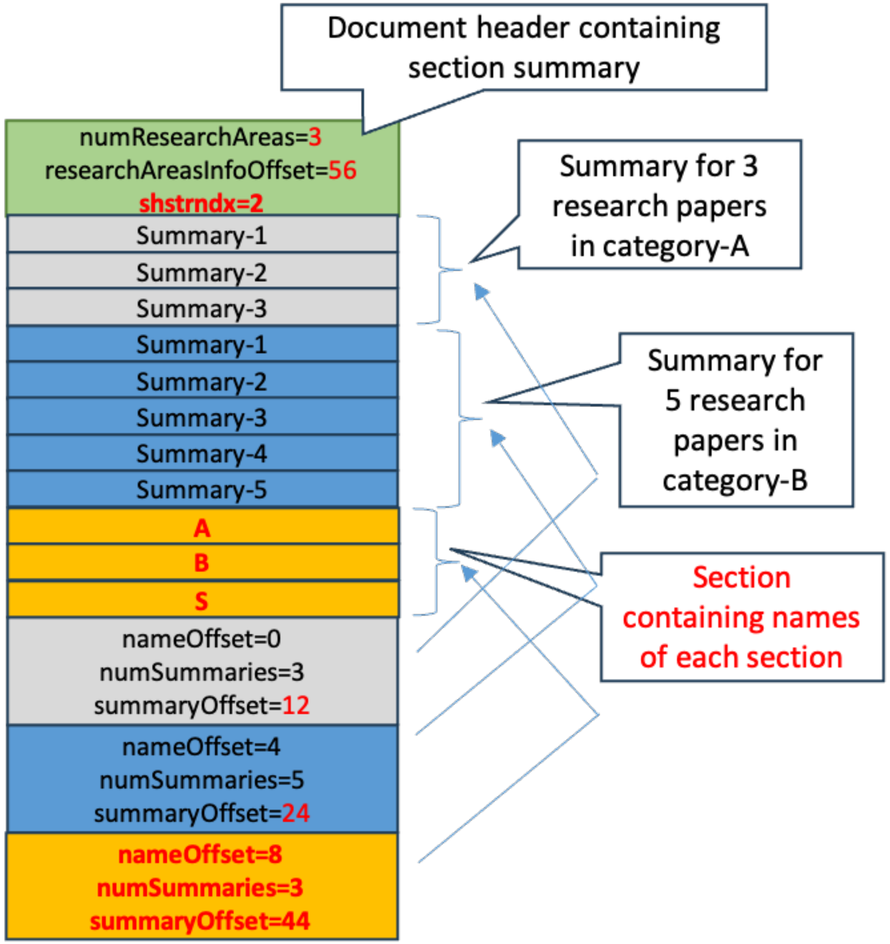
Lecture 04: Procedure Calling Convention

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture






```
[vivek@possum]$ cat fib.c
int get_number();

int fib(int n) {
  if(n<2) return n;
  else return fib(n-1)+fib(n-2);
}

int compute() {
  int n = get_number();
  return fib(n);
}
```

```
[vivek@possum]$ cat call-fib.c
int compute();
int get_number() {
  return 40;
}
int main() {
        int result = compute();
        return 0;
}
```

Compilation

fib.o

SYM Name	Type	Location
get_number	External	NULL
fib	Internal	Addr_A
compute	Internal	Addr_B

call-fib.o

SYM Name	Type	Location
compute	External	NULL
get_number	Internal	Addr_C

Resolve symbols by linking fib.o and call-fib.o together

a.out

SYM Name	Location
get_number	Addr_D
fib	Addr_E
compute	Addr_F

Today's Lecture

- **Loader**
- Call stack
- Procedure calling convention

Loader

- When an executable file is run, the operating system loads it into memory and start running it
- Loader uses the information stored inside the Program Header Table (PHT)
    - It doesn't require information stored in the Section Header Table (SHDR)
    - Linker combines multiple sections of similar type to create one segment
    - There are multiple segments in an executable file, and for each segment, there is one entry in the PHDR
    - Several sections can be removed from the executable without affecting program execution

Example: Loading Executable in Plain C

```
[vivek@possum]$ cat fib.c
int fib(int n) {
  if(n<2) return n;
  else return fib(n-1)+fib(n-2);
}

int main() {
        int val = fib(40);
        return val;
}
```

```
[vivek@possum]$ gcc -m32 -no-pie -nostdlib fib.c
/usr/bin/ld: warning: cannot find entry symbol _start; de
```

Why ??

Fix: rename "main" to _start"

- For simplicity, a) assume there are no APIs from the GNU C library, b) 32-bit compilation, and c) no use of global variables
- The loader has to simply iterate through the PHDR and identify the segment of PT_LOAD type
- Load this segment in memory and jump to [where?] for starting the execution

e_entrypoint address in EHDR

Example: Loading Executable in Plain C

```
[vivek@possum]$ readelf -h fib
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's compleme
  Version:                           1 (current)
  OS/ABI:                            UNIX - Syste
  ABI Version:                       0
  Type:                              EXEC (Execut
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048141
  Start of program headers:          52 (bytes in
  Start of section headers:          5200 (bytes
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         5
  Size of section headers:           40 (bytes)
  Number of section headers:         15
  Section header string table index: 14
```

```
[vivek@possum]$ readelf -l fib

Elf file type is EXEC (Executable file)
Entry point 0x8048141
There are 5 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
  LOAD           0x000000 0x08048000 0x08048000 0x00200 0x00200
  LOAD           0x001000 0x0804a000 0x0804a000 0x0000c 0x0000c
  NOTE           0x0000d4 0x080480d4 0x080480d4 0x00024 0x00024
  GNU_EH_FRAME   0x00016c 0x0804816c 0x0804816c 0x00024 0x00024
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .text .eh_frame_hdr .eh_frame
   01     .got.plt
   02     .note.gnu.build-id
   03     .eh_frame_hdr
   04
```

```c
typedef struct {
          Elf32_Word          p_type;
          Elf32_Off           p_offset;
          Elf32_Addr          p_vaddr;
          Elf32_Addr          p_paddr;
          Elf32_Word          p_filesz;
          Elf32_Word          p_memsz;
          Elf32_Word          p_flags;
          Elf32_Word          p_align;
} Elf32_Phdr;
```

- Array of PHDR with total ”e_phnum” number of elements starts at ”e_phoff” bytes from the start of the file a.out
- Load the first segment whose ”p_type” is “PT_LOAD” in memory. This segment starts at “p_offset”
- Move to the virtual address ”e_entrypoint” inside this segment
- Typecast it to the ”_start” function pointer type and simply execute!

Loader in OS: Complete Steps (1/2)

```
vivek@possum:~/os23$ readelf -l hello

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1080
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000660 0x0000000000000660  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x00000000000001b9 0x00000000000001b9  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
```

```
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .
gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.pl
t
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
```

Loader in OS: Complete Steps (2/2)

- When an executable file is run, the operating system loads it into memory and start it running as follows
  1. OS kernel reads the PHT and loads the parts specified in the segments of type “INTERP” and “LOAD”
  2. Control is transferred to the interpreter (/lib/ld-library.so)
  3. Dynamic Linker (DL) is invoked
  4. DL load the shared libraries needed by the executable and carry out the relocations
     - This information is available in “.dynamic” section
     - Connects symbolic references with their corresponding symbolic definitions (e.g., connecting the call to printf with the actual implementation of the printf in GNU C library)
  5. DL transfer the control to the address given by the symbol “_start” (and eventually to user main)

Today's Lecture

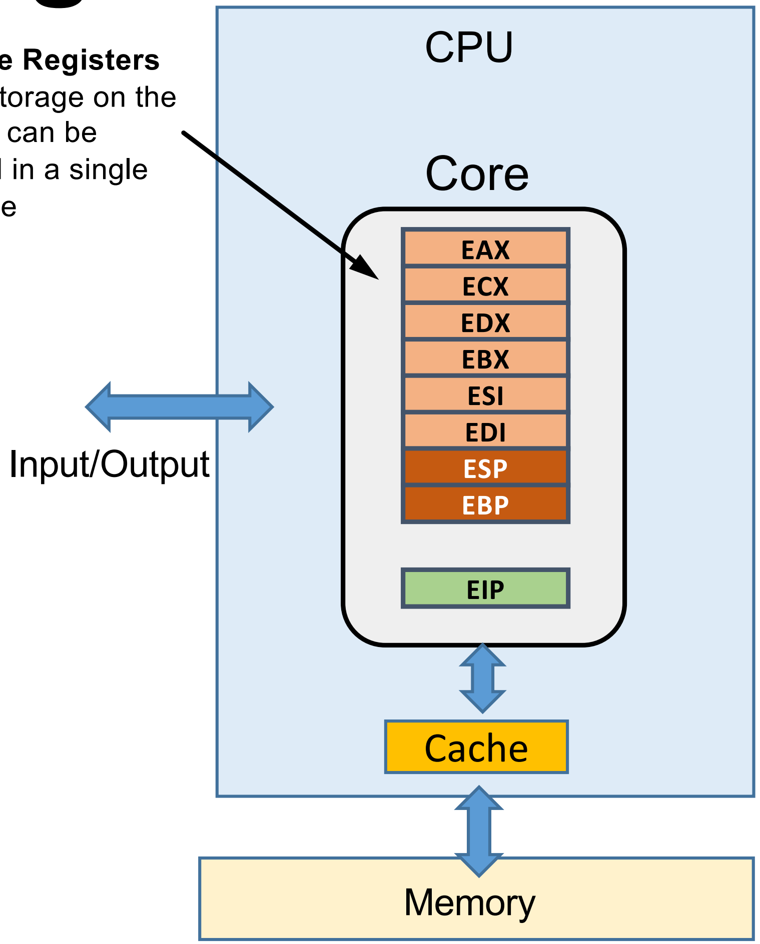
- Loader
- **Call stack**
- Procedure calling convention

Instruction Set Architectures (ISA)

- ISA provides a precise description of features provided by the hardware and how software can invoke and access these features
  - Can be thought of a programmer's manual to run programs on a CPU
    - Interface between the hardware and the software
  - Hides the hardware complexity
- Some of the widely used ISAs
  - **x86** (or **IA32**, i386) – Developed by Intel and widely used in desktop/laptop processors
    - Throughout this course we would only refer to 32-bit x86 (or IA32)
  - x86-64 (or AMD64) – 64-bit version of x86 ISA
  - ARM – Widely used on processors for mobile devices due to their reduced instruction set, which results in small die area and low power consumption

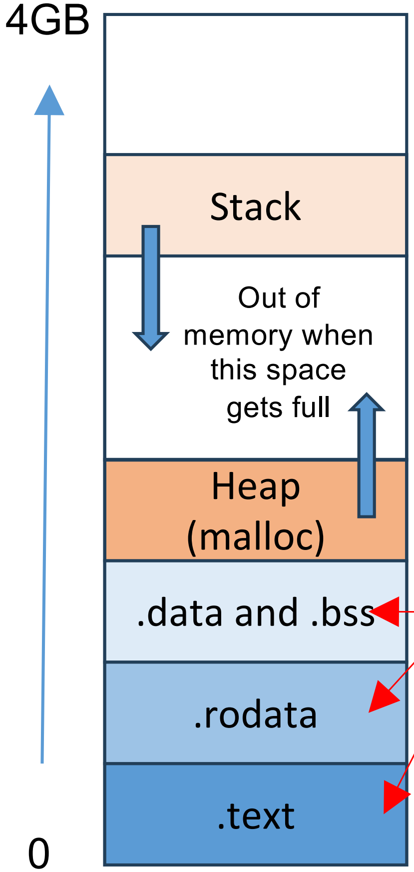
Registers in IA32

Hardware Registers
A small storage on the CPU that can be accessed in a single CPU cycle



- There are total of eight data registers where six are general purpose and two are special purpose (ESP and EBP)
- EIP is a control register to hold the program counter

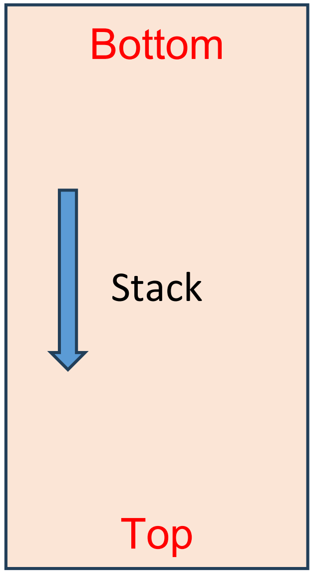
Memory Layout of Program in Execution




```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00770 0x00770 R E 0x1000
  LOAD           0x000ed8 0x00001ed8 0x00001ed8 0x00138 0x400168 RW  0x1000
  DYNAMIC        0x000ee0 0x00001ee0 0x00001ee0 0x000f8 0x000f8 RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x000638 0x00000638 0x00000638 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000ed8 0x00001ed8 0x00001ed8 0x00128 0x00128 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fini
 .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
```

- Total 2^{32} unique memory address on a 32-bit machine
- Stack grows downward whereas heap grows upward
  - Out of memory error generated when there is no more space left. You can easily get this runtime exception while running a Java program. Why?
- Each of the slots shown in the figure consists of several memory pages (4K)
- Stack is the place where the intermediate state of program execution is stored
- Which of the memory slots are Readable? Executable? Writable?

Upside Down Call Stack in IA32



- Stack is just a region of memory allocated by the OS for a running program to save the temporary variables inside a method and to support method invocation
  - Stack grows downward
- Used to hold temporary program state
  - Method call stack (callee and caller information)
    - Local variables
    - Function arguments
    - Return address
    - Temporary space
- This **convention** of saving and restoring program state across method calls is called procedure call linkage. We will now see how it happens on IA32

Callee and Caller

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```

- Call chain
  - main → foo → bar → baz
- Callee and Caller
  - main (caller) → foo (callee)
  - foo (caller) → bar (callee)
  - bar (caller) → baz (callee)

Today's Lecture

- Loader
- Call stack
- **Procedure calling convention**

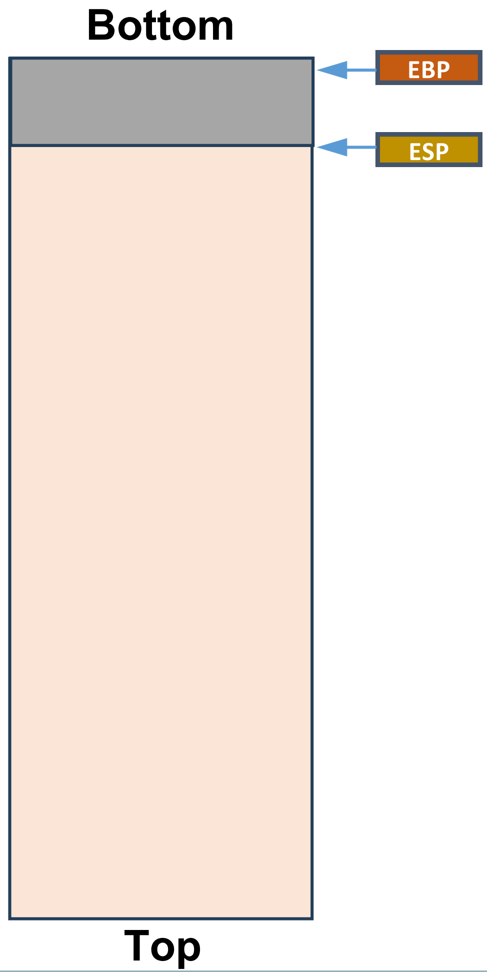
Stack Frames in IA32 (1/7)

```
L1: main () {
L2:   foo();
L3: }
```

```
L4: foo () {
L5:   bar(100, 200);
L6: }
```

```
L7: bar(int x1, int x2) {
L8:   int b1=x1,b2=x2, b3;
L9:   b3 = baz();
L10: }
```

```
L11: int baz () {
L12:   int r = 10;
L13:   return r;
L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

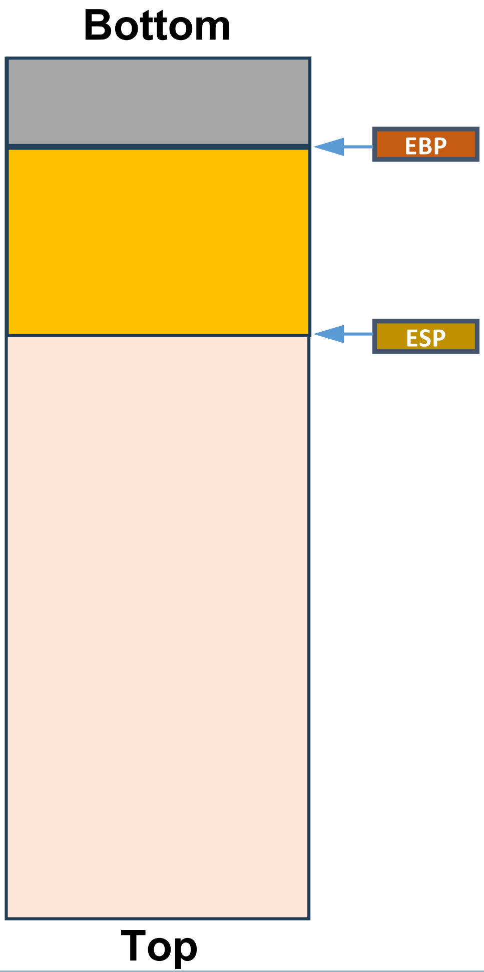
Stack Frames in IA32 (2/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

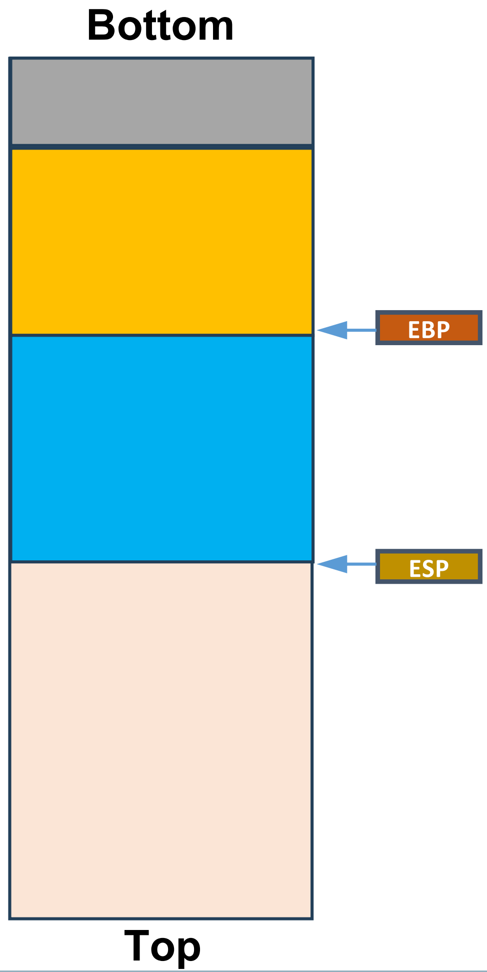
Stack Frames in IA32 (3/7)

```
L1: main () {
L2:    foo();
L3: }
```

```
L4: foo () {
L5:    bar(100, 200);
L6: }
```

```
L7: bar(int x1, int x2) {
L8:    int b1=x1,b2=x2, b3;
L9:    b3 = baz();
L10: }
```

```
L11: int baz () {
L12:    int r = 10;
L13:    return r;
L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

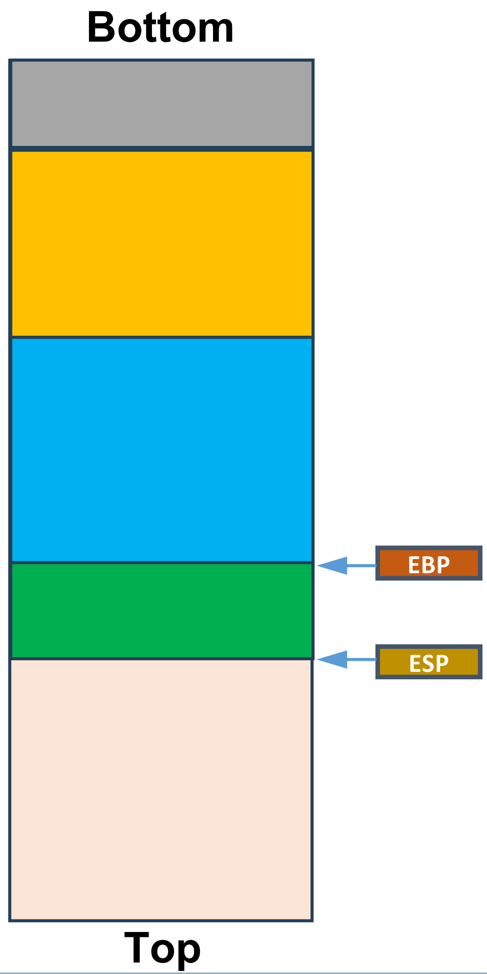
Stack Frames in IA32 (4/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

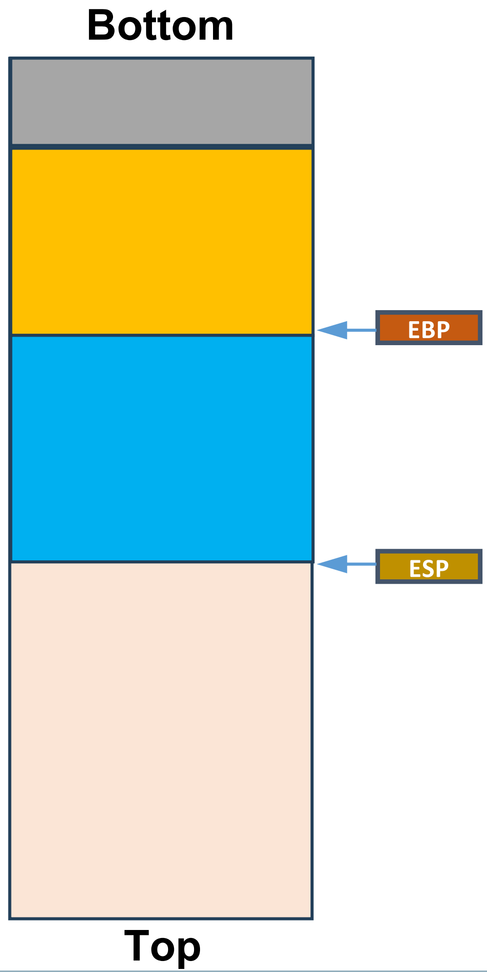
Stack Frames in IA32 (5/7)

```
L1: main () {
L2:   foo();
L3: }
```

```
L4: foo () {
L5:   bar(100, 200);
L6: }
```

```
L7: bar(int x1, int x2) {
L8:   int b1=x1,b2=x2, b3;
L9:   b3 = baz();
L10: }
```

```
L11: int baz () {
L12:   int r = 10;
L13:   return r;
L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

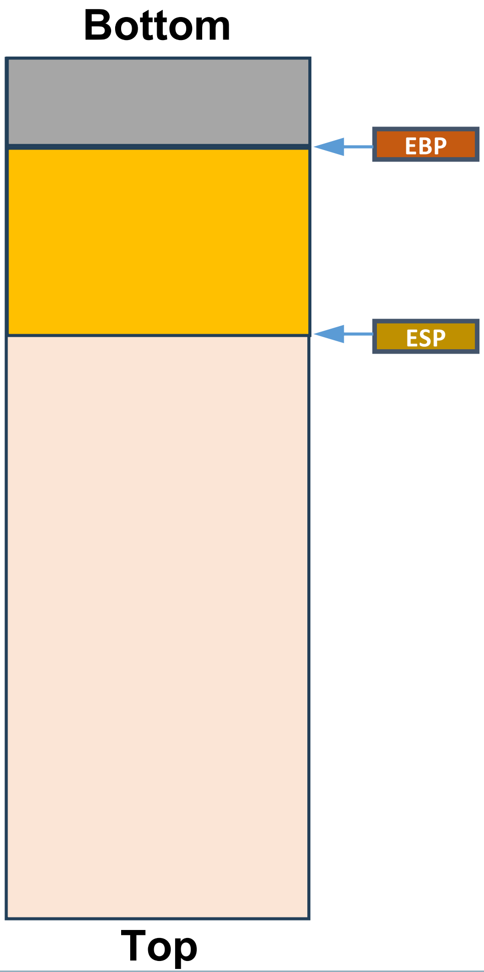
Stack Frames in IA32 (6/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

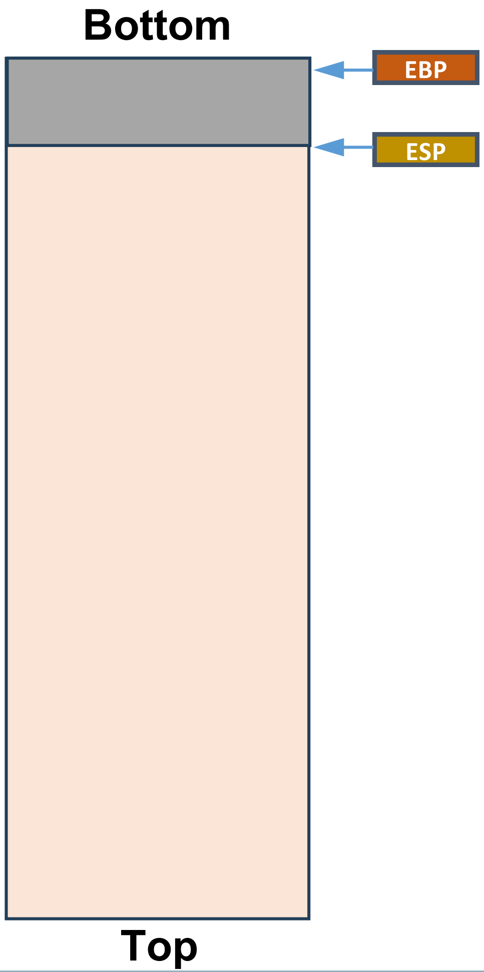
Stack Frames in IA32 (7/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

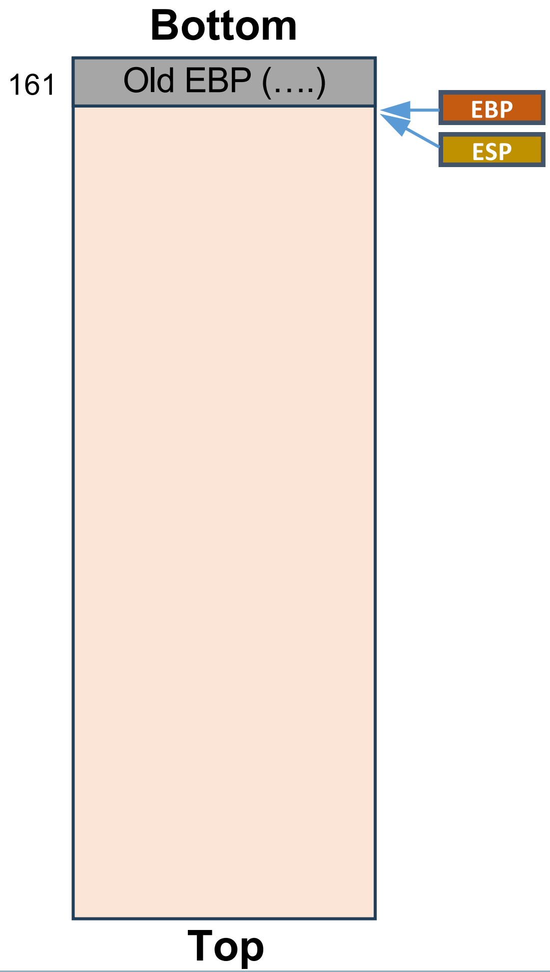
Calling Convention in X86 (1/14)

```
L1: main () {
       <save_regs>

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <save regs>
  - Current content of EBP register is saved in stack
    - Which method it corresponds to?
    - Carried out before every method call
    - Also known as function prologue
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot

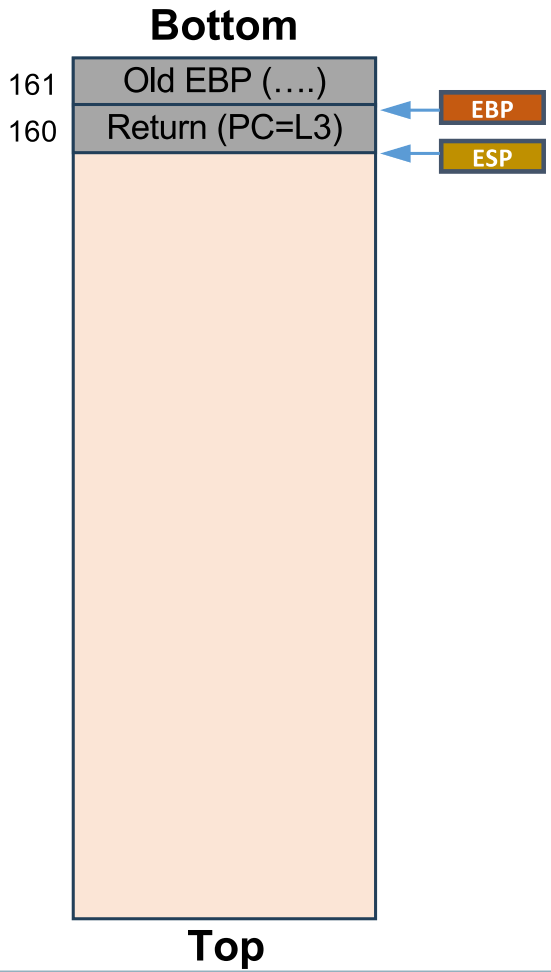
Calling Convention in X86 (2/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <save (PC)>
  - Current content of EIP register is saved in stack
  - ESP is now pointing to newly added slot

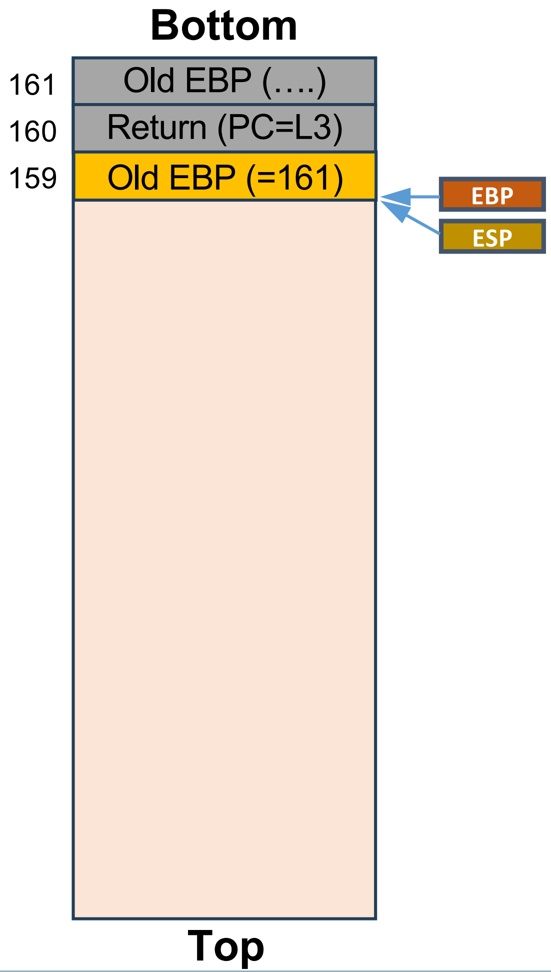
Calling Convention in X86 (3/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <save regs>
  - Current content of EBP register is saved in stack
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot

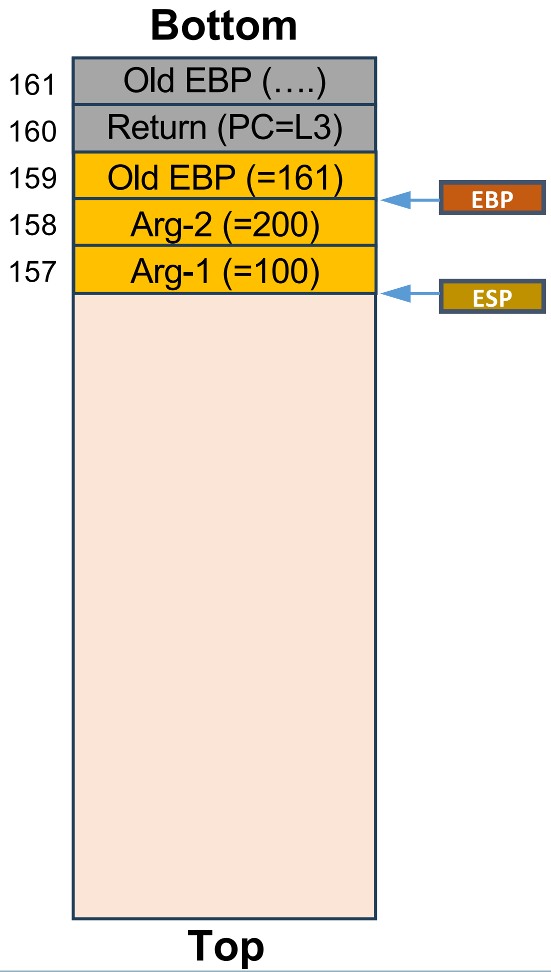
Calling Convention in X86 (4/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <alloc(8)>
  - Two slots are added on the stack in the current stack frame for two arguments to the callee method
  - Arguments are stored in the reverse ordered on the stack
  - ESP is updated

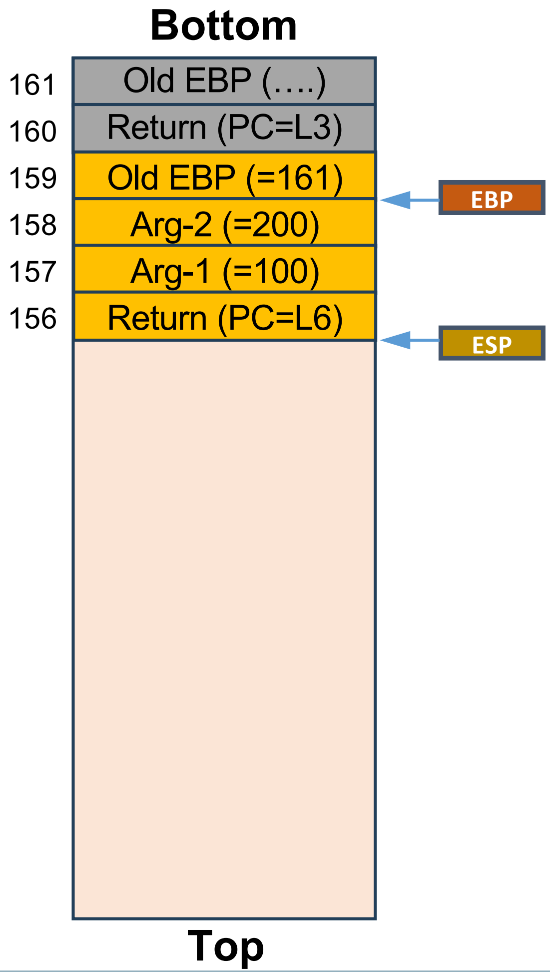
Calling Convention in X86 (5/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);

L6: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <save (PC)>
  - Current content of EIP register is saved in stack
  - ESP is now pointing to newly added slot

Calling Convention in X86 (6/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:   int r = 10;
L13:   return r;

L14: }
```

Bottom

161	Old EBP (….)	
160	Return (PC=L3)	
159	Old EBP (=161)	
158	Arg-2 (=200)	
157	Arg-1 (=100)	
156	Return (PC=L6)	
155	Old EBP (=159)	← EBP, ESP

Top

- <save regs>
  - Current content of EBP register is saved in stack
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot

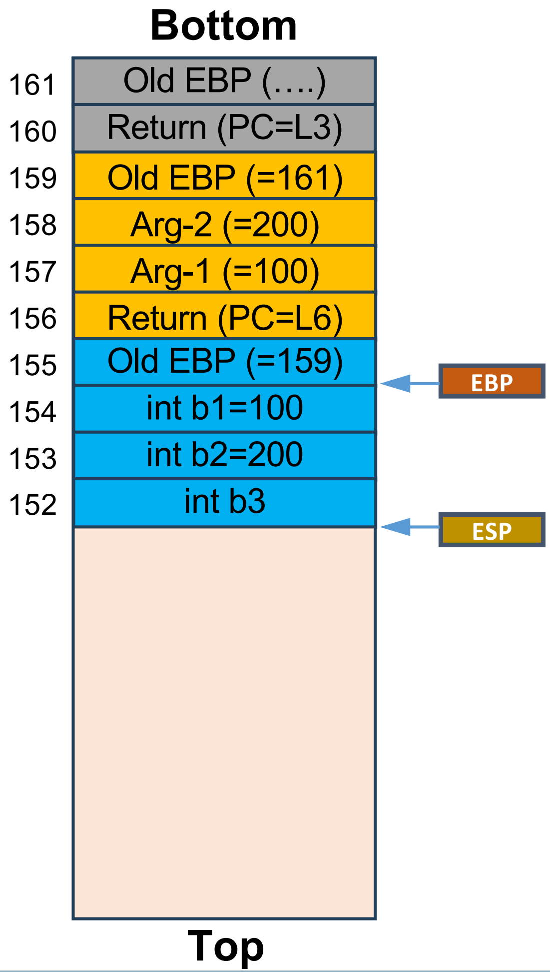
Calling Convention in X86 (7/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <alloc(12)>
  - Three slots are added on the stack in the current stack frame for the three local variables
  - Variables are stored in the same order on the stack as encountered
  - ESP is updated

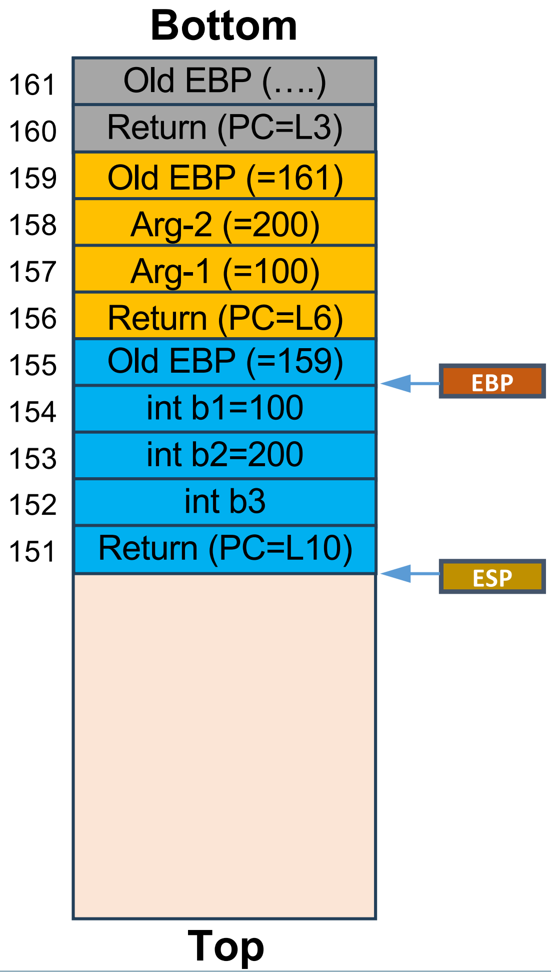
Calling Convention in X86 (8/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:   int b1=x1,b2=x2, b3;
        <save(PC)>
L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:   int r = 10;
L13:   return r;

L14: }
```



- <save (PC)>
  - Current content of EIP register is saved in stack
  - ESP is now pointing to newly added slot

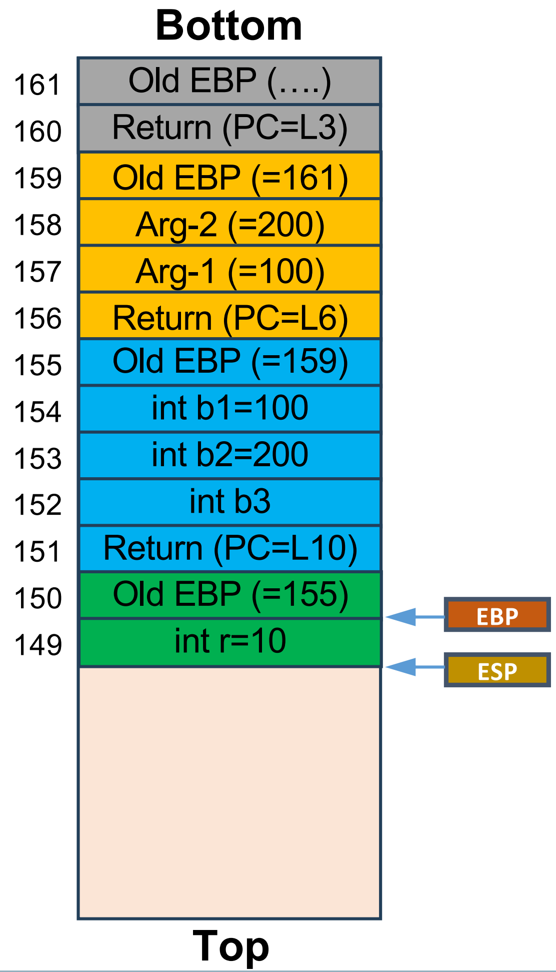
Calling Convention in X86 (9/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();


L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);


L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:   int b1=x1,b2=x2, b3;
        <save(PC)>
L9:   b3 = baz();


L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:   int r = 10;
L13:   return r;


L14: }
```



- <save regs>
  - Current content of EBP register is saved in stack
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot
- <alloc(4)>
  - A slot is added on the stack in the current stack frame for the local variable

Calling Convention in X86 (10/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:   int b1=x1,b2=x2, b3;
        <save(PC)>
L9:   b3 = baz();

L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:   int r = 10;
L13:   return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```

Bottom

161	Old EBP (….)	
160	Return (PC=L3)	
159	Old EBP (=161)	
158	Arg-2 (=200)	
157	Arg-1 (=100)	
156	Return (PC=L6)	
155	Old EBP (=159)	← EBP
154	int b1=100	
153	int b2=200	
152	int b3	
151	Return (PC=L10)	← ESP

Top

- Value returned is stored inside the slot 152 for "b3"
- <dealloc(4)>
  - ESP is updated to point to the slot just before the slot(s) added for storing local variable(s)
- <restore_regs>
  - Old value of EBP is popped and stored in EBP
  - ESP adjusted to point to the last slot in the current stack frame
  - Carried out before exiting any method call
  - Also known as function epilogue

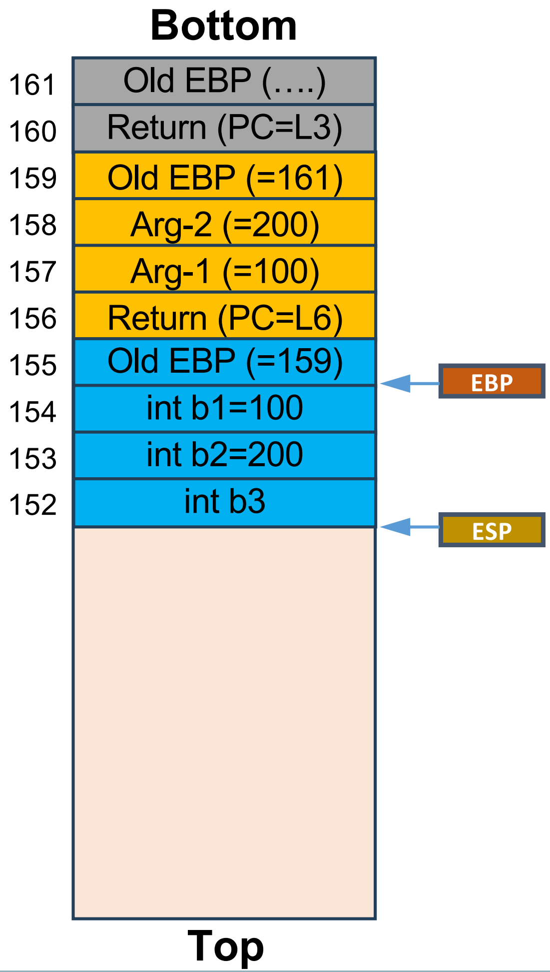
Calling Convention in X86 (11/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:   int b1=x1,b2=x2, b3;
        <save(PC)>
L9:   b3 = baz();
        <restore(PC)>

L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:    int r = 10;
L13:    return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```



- <restore(PC)>
  - EIP register is now restored with the old value of EIP before the call went inside the callee

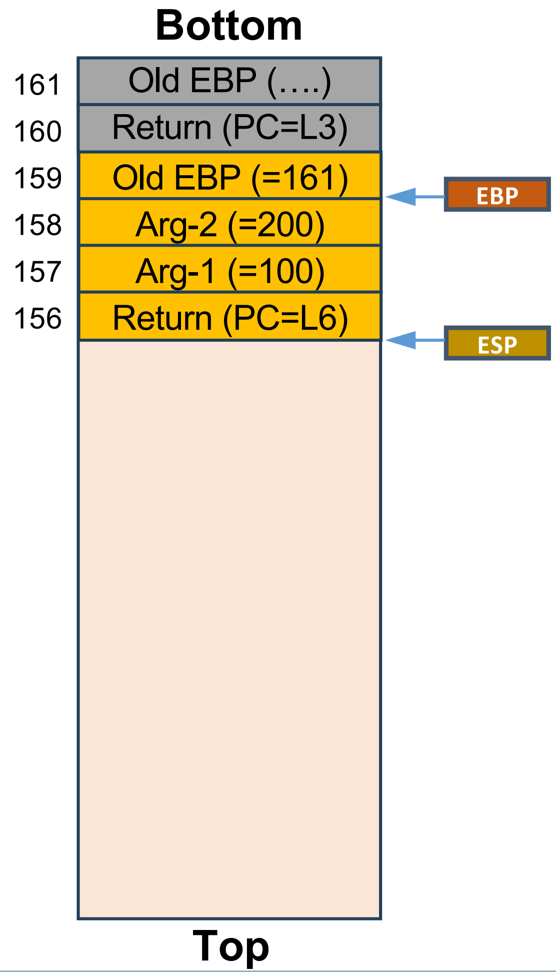
Calling Convention in X86 (12/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:   int b1=x1,b2=x2, b3;
        <save(PC)>
L9:   b3 = baz();
        <restore(PC)>
        <dealloc(12)>
        <restore_regs>
L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:   int r = 10;
L13:   return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```



- <dealloc(12)>
  - ESP is updated to point to the slot just before the slot(s) added for storing local variable(s)
- <restore_regs>
  - Old value of EBP is popped and stored in EBP
  - ESP adjusted to point to the last slot in the current stack frame

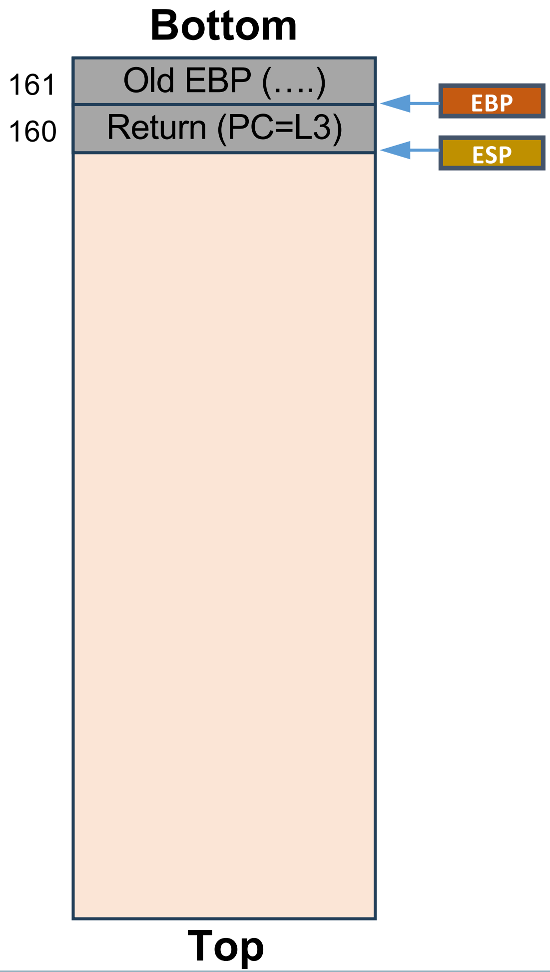
Calling Convention in X86 (13/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);
        <restore(PC)>
        <dealloc(8)>
        <restore_regs>
L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:   int b1=x1,b2=x2, b3;
        <save(PC)>
L9:   b3 = baz();
        <restore(PC)>
        <dealloc(12)>
        <restore_regs>
L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:   int r = 10;
L13:   return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```



- <restore(PC)>
  - EIP register is now restored with the old value of EIP before the call went inside the callee
- <dealloc(8)>
  - ESP is updated to point to the slot just before the slot(s) added for storing local variable(s)
- <restore_regs>
  - Old value of EBP is popped and stored in EBP
  - ESP adjusted to point to the last slot in the current stack frame

Calling Convention in X86 (14/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:   foo();
        <restore(PC)>
        <restore_regs>
L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:   bar(100, 200);
        <restore(PC)>
        <dealloc(8)>
        <restore_regs>
L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:   int b1=x1,b2=x2, b3;
        <save(PC)>
L9:   b3 = baz();
        <restore(PC)>
        <dealloc(12)>
        <restore_regs>
L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:   int r = 10;
L13:   return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```

Bottom

Top

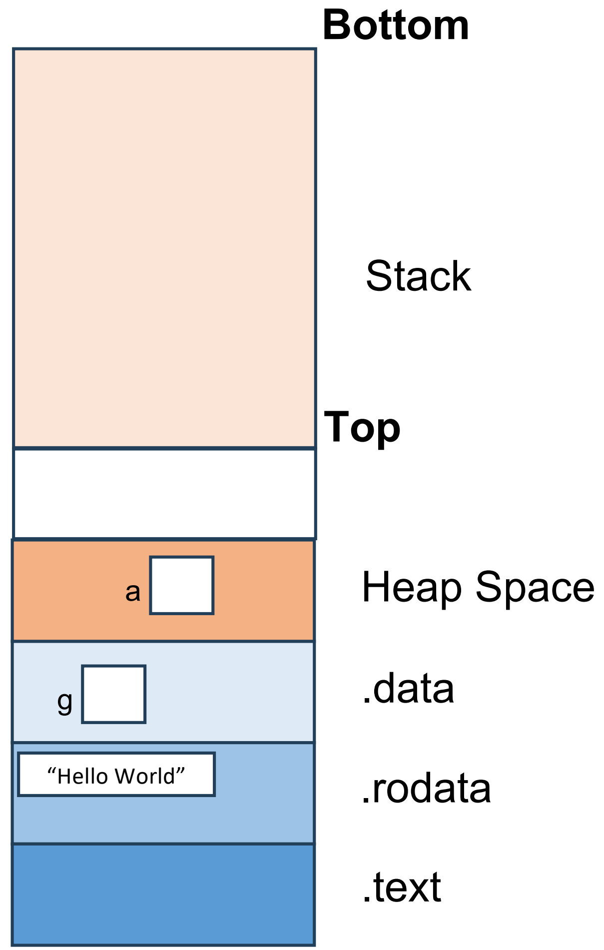
- As the main method terminates, call is transferred to the caller that happens to be the libc main and then finally to the _start method

Program Execution: Class Exercise

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = “Hello World”;
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* a) {
L9:     *a = 20;
L10:}
```

Draw the stack frames when the execution is at L9

I will discuss it in next lecture recap



Caller and Callee Save Registers

- Recall, there are six general purpose registers that can be used to store temporary data during a method execution
- If the caller is using these registers, it must save each of these 6 registers before transferring the call to the callee
- However, it could be possible that callee doesn't use all these registers. Hence, the saving and restoring of these 6 registers is divided across both caller and the callee
- If the caller method is using these data registers then it saves/restore only these 3 on its call stack before/after method call: **EAX**, **EDX**, and **ECX**
- If the callee method wants to use these 3 registers then it saves/restore the current value on its call stack at entry/exit: **EBX**, **ESI**, and **EDI**

x86-64 Calling Convention

- Different than IA32 (x86)
- There are 16 general purpose 64-bit registers compared to 8, 32-bit registers on x86. Hence, less dependence on stack
  - As example, the arguments to callee and temporary variables can be stored on registers
  - If more data than registers then fall back to x86 way of storing on stack
- Execution is much faster due to less dependence on stack

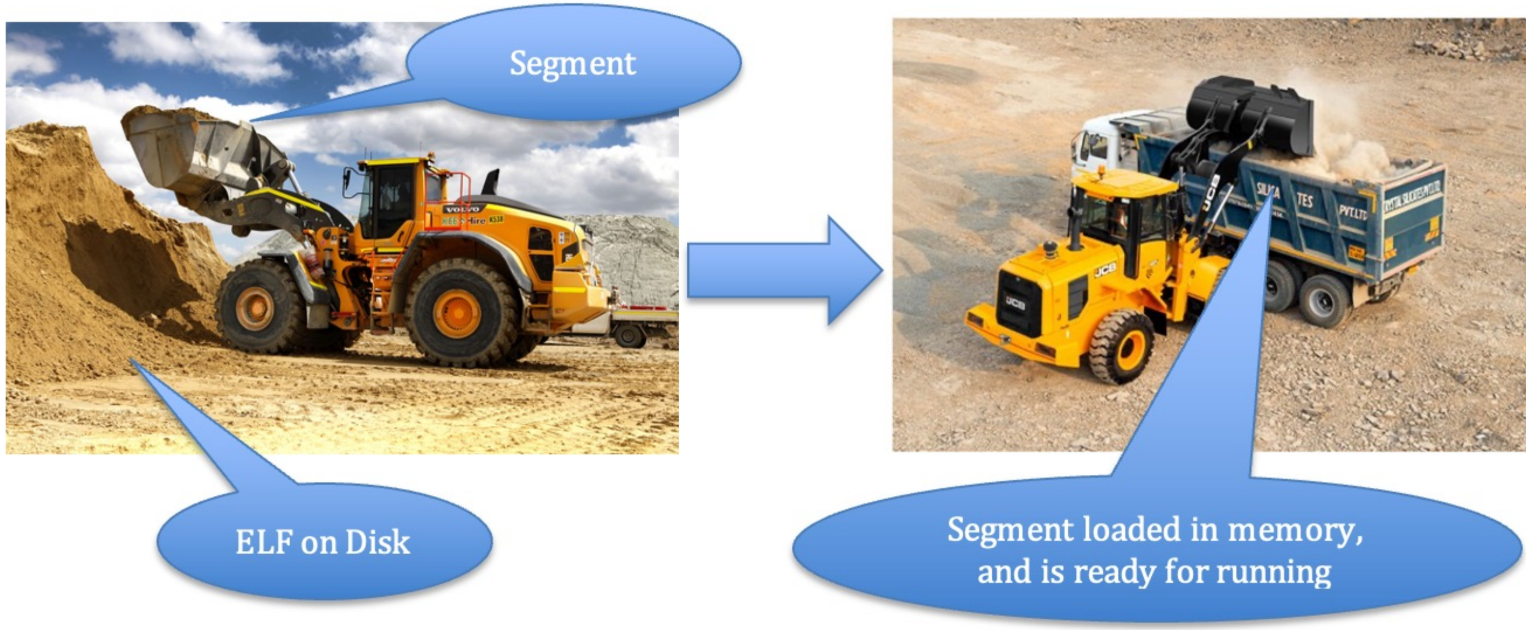
Assignment-1 (Section-A)

SimpleLoader: An ELF Loader in C from Scratch

Due by 11:59pm on 1st September 2024

(Total **7%** weightage)

Instructor: Vivek Kumar



Next Lecture

- The process abstraction
- Quiz-1 during lecture hours