2017Fall:Software Security

Lecture 2 : Machine-Level Representation of Programs

Bing Mao

maobing@nju.edu.cn

Department of Computer Science

Outline

Illustrate

Credit:a portion of the slides in this lecture are compiled from book CSAPP.

Introduction

- This chapter is intended for making preparations for subsequent chapters.
- The content contained is about the static structure of program and how CPU executes a program.

Assembly Basics
Register

Suppose we write a C file code,containing the following procedure definition:

```
1   int accum = 0;
2
3   int sum(int x, int y)
4   {
5       int t = x + y;
6       accum += t;
7       return t;
8   }
```

The assembly file contains various declaration including the set of lines:

```
1   00000000 <sum>:
    Offset   Bytes              Equivalent assembly language
2      0:    55                      push   %ebp
3      1:    89 e5                   mov    %esp,%ebp
4      3:    8b 45 0c                mov    0xc(%ebp),%eax
5      6:    03 45 08                add    0x8(%ebp),%eax
6      9:    01 05 00 00 00 00       add    %eax,0x0
7      f:    5d                      pop    %ebp
8     10:    c3                      ret
```

Assembly Basics
Register

Suppose we write a C file code,containing the following procedure definition:

```
1   int accum = 0;
2
3   int sum(int x, int y)
4   {
5       int t = x + y;
6       accum += t;
7       return t;
8   }
```

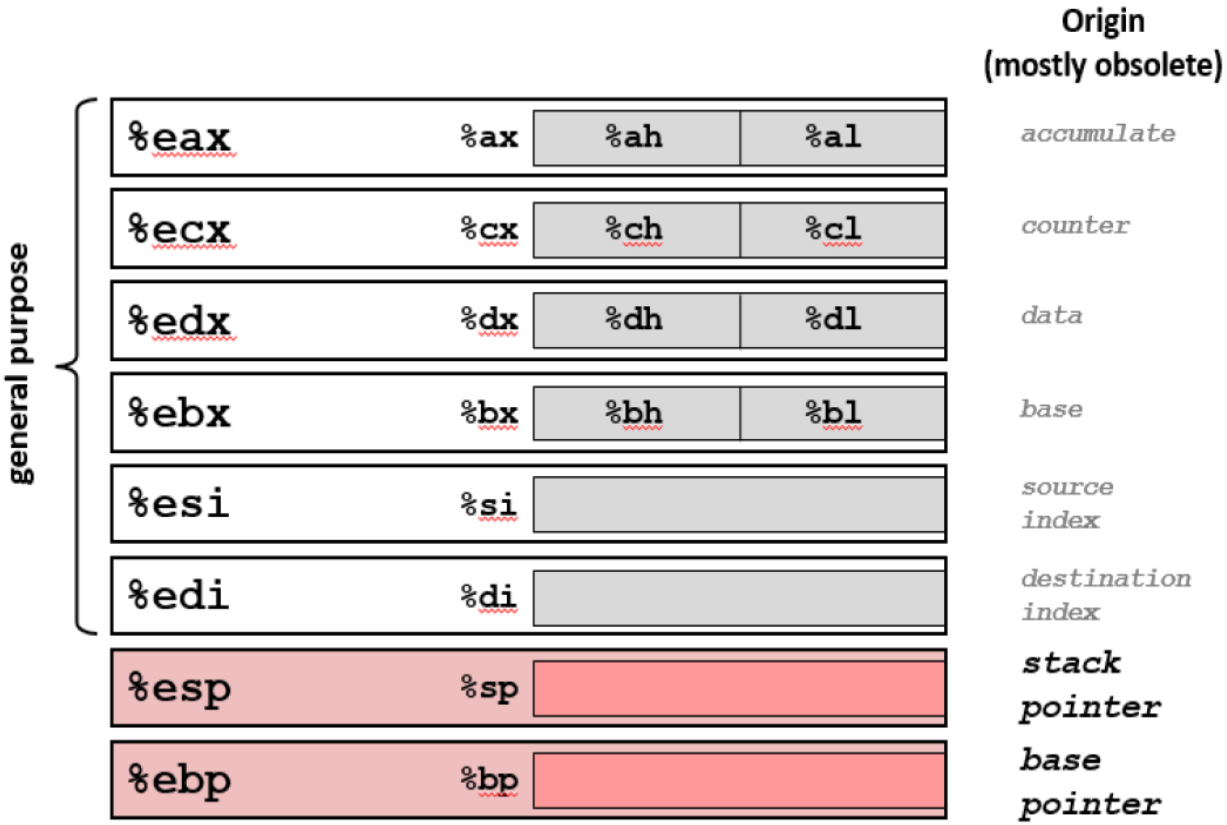
The assembly file contains various declaration including the set of lines:

```
1   00000000 <sum>:
    Offset   Bytes              Equivalent assembly language
2      0:    55                       push   %ebp
3      1:    89 e5                    mov    %esp,%ebp
4      3:    8b 45 0c                 mov    0xc(%ebp),%eax
5      6:    03 45 08                 add    0x8(%ebp),%eax
6      9:    01 05 00 00 00 00        add    %eax,0x0
7      f:    5d                       pop    %ebp
8     10:    c3                       ret
```

Questions

- 1.push? mov? add? pop? ret?
- 2.%esp? %ebp? %eax?

IA32 Registers



Simple Memory Addressing Modes

- **1.Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C
  - Example: movq (%ecx),%eax
- **2.Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Example: movq 8(%ebp),%edx

Complete Memory Addressing Modes

- **1.Most General Form D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]**
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for
  - S: Scale:1, 2, 4, or 8 (why these numbers?)
- **2.Special Cases**
  - (Rb,Ri): Mem[Reg[Rb]+Reg[Ri]]
  - (Rb,Ri): Mem[Reg[Rb]+Reg[Ri]+D]
  - (Rb,Ri,S): Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

- **1.Most General Form D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]**
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for
  - S: Scale:1, 2, 4, or 8 (why these numbers?)
- **2.Special Cases**
  - (Rb,Ri): Mem[Reg[Rb]+Reg[Ri]]
  - (Rb,Ri): Mem[Reg[Rb]+Reg[Ri]+D]
  - (Rb,Ri,S): Mem[Reg[Rb]+S*Reg[Ri]]

Operand forms

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$Imm(E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled indexed
Memory	(E_b, E_i, s)	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

Address Computation Examples

%edx	0xf000
%ecx	0x0100

Expression	Address Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

Practice Problem

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the values for the indicated operands:

Operand	Value
%eax	_____
0x104	_____
$0x108	_____
(%eax)	_____
4(%eax)	_____
9(%eax,%edx)	_____
260(%ecx,%edx)	_____
0xFC(,%ecx,4)	_____
(%eax,%edx,4)	_____

Moving data

- mov Source, Dest:

Operand Types

- 1.Immediate: Constant integer data
  - Example: 0*x*400,-533
  - Example: 0*x*400,-533
  - Encoded with 1, 2, or 4 bytes
- 2.Register: One of 16 integer registers
  - Example: %eax, %r13
  - But %esp reserved for special use
  - Others have special uses for particular instructions
- 3.Memory:address given by register
  - Simplest example:(%eax)
  - Various other "address modes"

MOV Operand Combinations

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb		Move byte	
movw		Move word	
movl		Move double word	
MOVS	S, D	$D \leftarrow$ SignExtend(S)	Move with sign extension
movsbw		Move sign-extended byte to word	
movsbl		Move sign-extended byte to double word	
movswl		Move sign-extended word to double word	
MOVZ	S, D	$D \leftarrow$ ZeroExtend(S)	Move with zero extension
movzbw		Move zero-extended byte to word	
movzbl		Move zero-extended byte to double word	
movzwl		Move zero-extended word to double word	
pushl	S	R[%esp] $\leftarrow$ R[%esp] $-$ 4; M[R[%esp]] $\leftarrow S$	Push double word
popl	D	$D \leftarrow$ M[R[%esp]]; R[%esp] $\leftarrow$ R[%esp] $+$ 4	Pop double word

MOV Operand Combinations

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb		Move byte	
movw		Move word	
movl		Move double word	
MOVS	S, D	$D \leftarrow$ SignExtend(S)	Move with sign extension
movsbw		Move sign-extended byte to word	
movsbl		Move sign-extended byte to double word	
movswl		Move sign-extended word to double word	
MOVZ	S, D	$D \leftarrow$ ZeroExtend(S)	Move with zero extension
movzbw		Move zero-extended byte to word	
movzbl		Move zero-extended byte to double word	
movzwl		Move zero-extended word to double word	
pushl	S	R[%esp] $\leftarrow$ R[%esp] $-$ 4; M[R[%esp]] $\leftarrow S$	Push double word
popl	D	$D \leftarrow$ M[R[%esp]]; R[%esp] $\leftarrow$ R[%esp] $+$ 4	Pop double word

Cannot do memory-memory transfer with a single instruction.

MOV Operand Combinations

- Data Movement Instruction
  - The instructions in the mov class copy their source values to their destinations. The source operand comes first and the destination second:

```
1    movl $0x4050,%eax          Immediate--Register, 4 bytes
2    movw  %bp,%sp              Register--Register,  2 bytes
3    movb (%edi,%ecx),%ah       Memory--Register,    1 byte
4    movb $-17,(%esp)           Immediate--Memory,   1 byte
5    movl %eax,-12(%ebp)        Register--Memory,    4 bytes
```

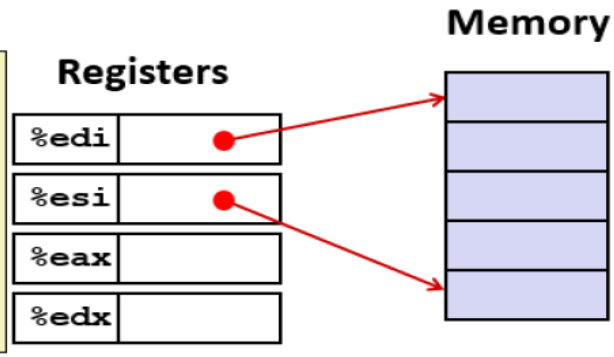
Example of mov operation

```
void swap(long *xp, long *yp){
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq (%edi), %eax
    movq (%esi), %edx
    movq %edx, (%edi)
    movq %eax, (%esi)
    ret
```

Understanding Swap()

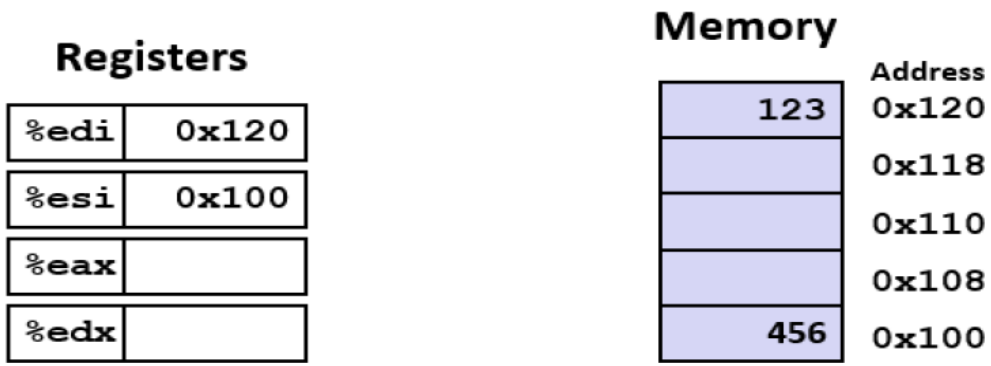
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%edi	xp
%esi	yp
%eax	t0
%edx	t1

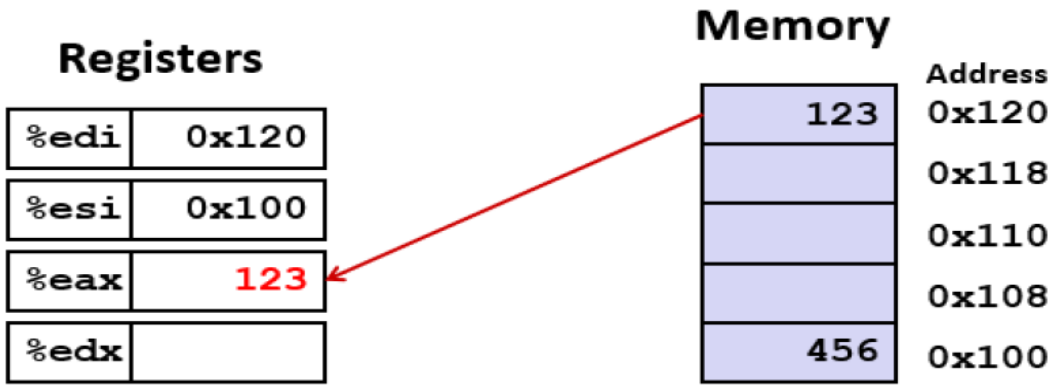
```
swap:
    movq    (%edi), %eax  # t0 = *xp
    movq    (%esi), %edx  # t1 = *yp
    movq    %edx, (%edi)  # *xp = t1
    movq    %eax, (%esi)  # *yp = t0
    ret
```

Understanding Swap()



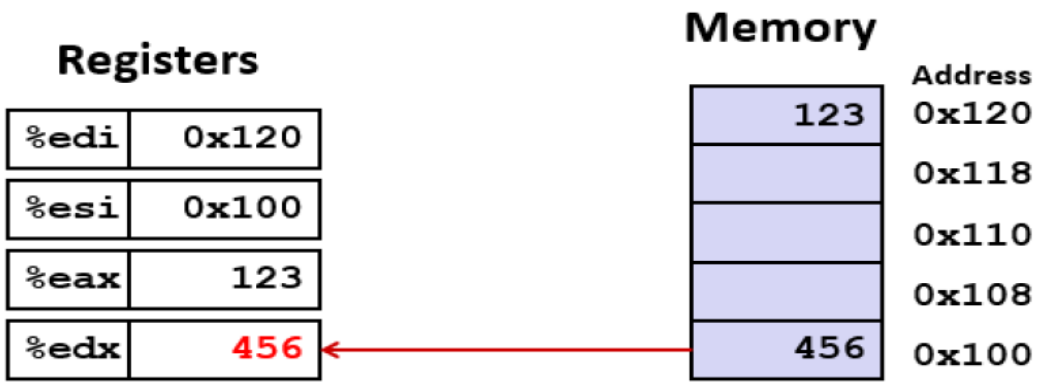

```
swap:
    movq    (%edi), %eax   # t0 = *xp
    movq    (%esi), %edx   # t1 = *yp
    movq    %edx, (%edi)   # *xp = t1
    movq    %eax, (%esi)   # *yp = t0
    ret
```

Understanding Swap()



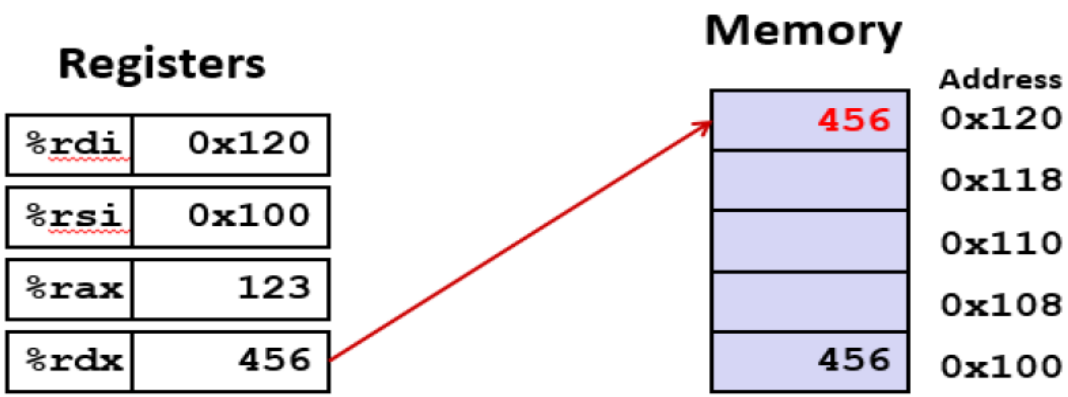

```
swap:
    movq    (%edi), %eax   # t0 = *xp
    movq    (%esi), %edx   # t1 = *yp
    movq    %edx, (%edi)   # *xp = t1
    movq    %eax, (%esi)   # *yp = t0
    ret
```

Understanding Swap()



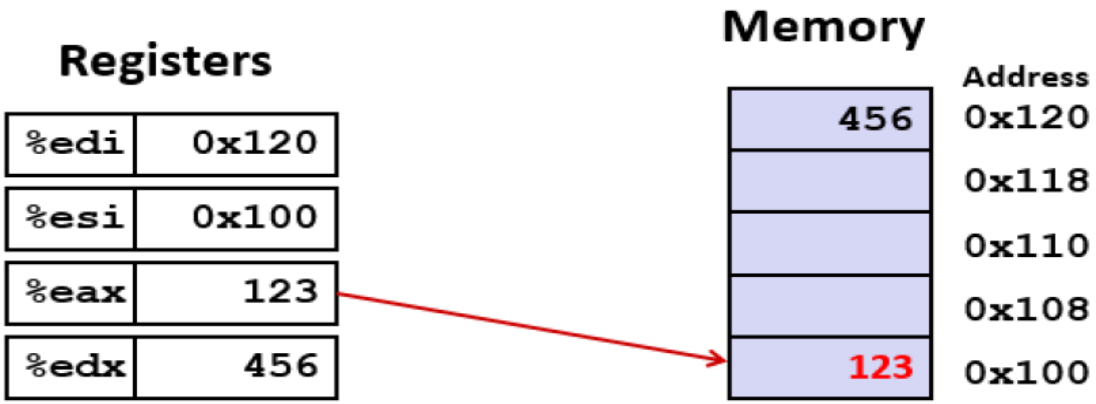

```
swap:
    movq    (%edi), %eax   # t0 = *xp
    movq    (%esi), %edx   # t1 = *yp
    movq    %edx, (%edi)   # *xp = t1
    movq    %eax, (%esi)   # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%edi), %eax   # t0 = *xp
    movq    (%esi), %edx   # t1 = *yp
    movq    %edx, (%edi)   # *xp = t1
    movq    %eax, (%esi)   # *yp = t0
    ret
```

Practice Problem

Practice Problem

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, `mov` can be rewritten as `movb`, `movw`, or `movl`.)

```
1       mov   %eax, (%esp)
2       mov   (%eax), %dx
3       mov   $0xFF, %bl
4       mov   (%esp,%edx,4), %dh
5       push  $0xFF
6       mov   %dx, (%eax)
7       pop   %edi
```

Practice Problem

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

```
1       movb $0xF, (%bl)
2       movl %ax, (%esp)
3       movw (%eax),4(%esp)
4       movb %ah,%sh
5       movl %eax,$0x123
6       movl %eax,%dx
7       movb %si, 8(%ebp)
```

Arithmetic and Logical Operations

Load Effective Address

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Practice Problem

Suppose register %eax holds value x and %ecx holds value y. Fill in the table below with formulas indicating the value that will be stored in register %edx for each of the given assembly code instructions:

Instruction	Result
`leal 6(%eax), %edx`	
`leal (%eax,%ecx), %edx`	
`leal (%eax,%ecx,4), %edx`	
`leal 7(%eax,%eax,8), %edx`	
`leal 0xA(,%ecx,4), %edx`	
`leal 9(%eax,%ecx,2), %edx`	

Practice Problem

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the effects of the following instructions, both in terms of the register or memory location that will be updated and the resulting value:

Instruction	Destination	Value
`addl %ecx,(%eax)`		
`subl %edx,4(%eax)`		
`imull $16,(%eax,%edx,4)`		
`incl 8(%eax)`		
`decl %ecx`		
`subl %edx,%eax`		

The final group consists of shift operations, where the shift amount is given first, and the value to shift is given second.

SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Practice Problem

Suppose we want to generate assembly code for the following C function:

```
int shift_left2_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```

The code that follows is a portion of the assembly code that performs the actual shifts and leaves the final value in register %eax. Two key instructions have been omitted. Parameters x and n are stored at memory locations with offsets 8 and 12, respectively, relative to the address in register %ebp.

```
1    movl    8(%ebp), %eax    Get x
2    _____                    x <<= 2
3    movl    12(%ebp), %ecx   Get n
4    _____                    x >>= n
```

Fill in the missing instructions, following the annotations on the right. The right shift should be performed arithmetically.

Comparison and test instruction

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
`cmpb`		Compare byte	
`cmpw`		Compare word	
`cmpl`		Compare double word	
TEST	S_2, S_1	S_1 & S_2	Test
`testb`		Test byte	
`testw`		Test word	
`testl`		Test double word	

Practice Problem

The following C code

```
int comp(data_t a, data_t b) {
    return a COMP b;
}
```

shows a general comparison between arguments `a` and `b`, where we can set the data type of the arguments by declaring `data_t` with a `typedef` declaration, and we can set the comparison by defining `COMP` with a `#define` declaration.

Suppose `a` is in `%edx` and `b` is in `%eax`. For each of the following instruction sequences, determine which data types `data_t` and which comparisons `COMP` could cause the compiler to generate this code. (There can be multiple correct answers; you should list them all.)

A.
```
cmpl    %eax, %edx
setl    %al
```

B.
```
cmpw    %ax, %dx
setge   %al
```

C.
```
cmpb    %al, %dl
setb    %al
```

D.
```
cmpl    %eax, %edx
setne   %al
```

Practice Problem

The following C code

```
int test(data_t a) {
    return a TEST 0;
}
```

shows a general comparison between argument `a` and 0, where we can set the data type of the argument by declaring `data_t` with a `typedef`, and the nature of the comparison by declaring `TEST` with a `#define` declaration. For each of the following instruction sequences, determine which data types `data_t` and which comparisons `TEST` could cause the compiler to generate this code. (There can be multiple correct answers; list all correct ones.)

A.
```
testl    %eax, %eax
setne    %al
```

B.
```
testw    %ax, %ax
sete     %al
```

C.
```
testb    %al, %al
setg     %al
```

D.
```
testw    %ax, %ax
seta     %al
```

Jump instructions

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Practice Problem

In the following excerpts from a disassembled binary, some of the information has been replaced by Xs. Answer the following questions about these instructions.

A. What is the target of the `je` instruction below? (You don't need to know anything about the `call` instruction here.)

```
804828f:    74 05              je    XXXXXXX
8048291:    e8 1e 00 00 00     call  80482b4
```

B. What is the target of the `jb` instruction below?

```
8048357:    72 e7              jb    XXXXXXX
8048359:    c6 05 10 a0 04 08 01   movb  $0x1,0x804a010
```

C. What is the address of the `mov` instruction?

```
XXXXXXX:    74 12              je    8048391
XXXXXXX:    b8 00 00 00 00     mov   $0x0,%eax
```

D. In the code that follows, the jump target is encoded in PC-relative form as a 4-byte, two's-complement number. The bytes are listed from least significant to most, reflecting the little-endian byte ordering of IA32. What is the address of the jump target?

```
80482bf:    e9 e0 ff ff ff     jmp   XXXXXXX
80482c4:    90                 nop
```

E. Explain the relation between the annotation on the right and the byte coding on the left.

```
80482aa:    ff 25 fc 9f 04 08      jmp   *0x8049ffc
```

Objects Files
Turning Source Code to Object Code

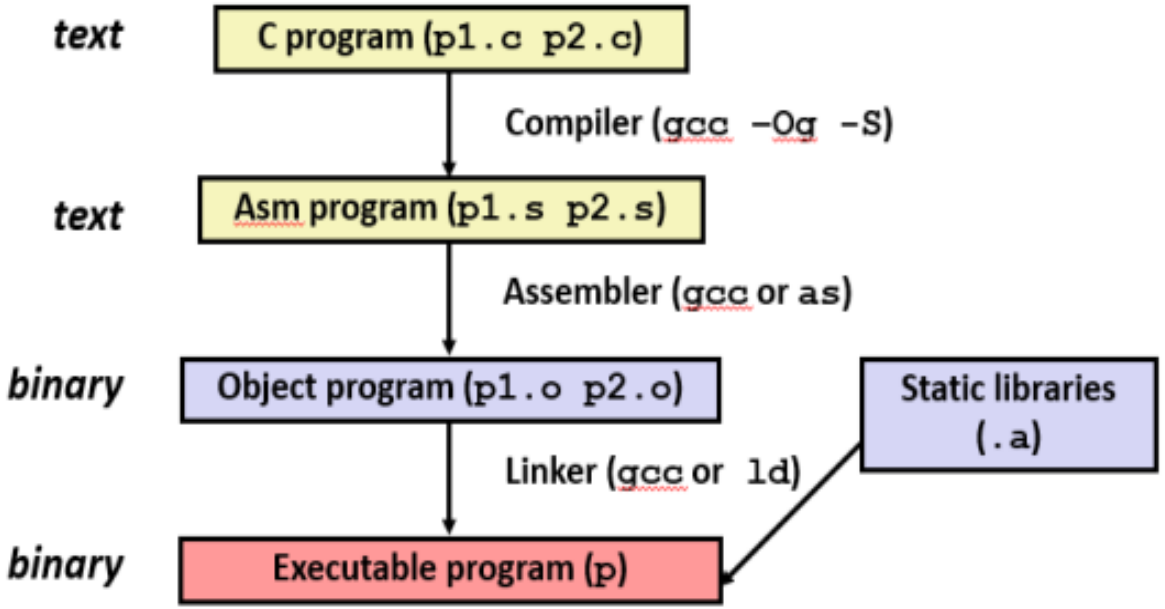
Programmer-Visible State

- Code in files p1.c p2.c
- Compile with command:gcc -Og p1.c p2.c -o p
  - Use basic optimizations (-Og) [New to recent versions of GCC]
  - Put resulting binary in file p

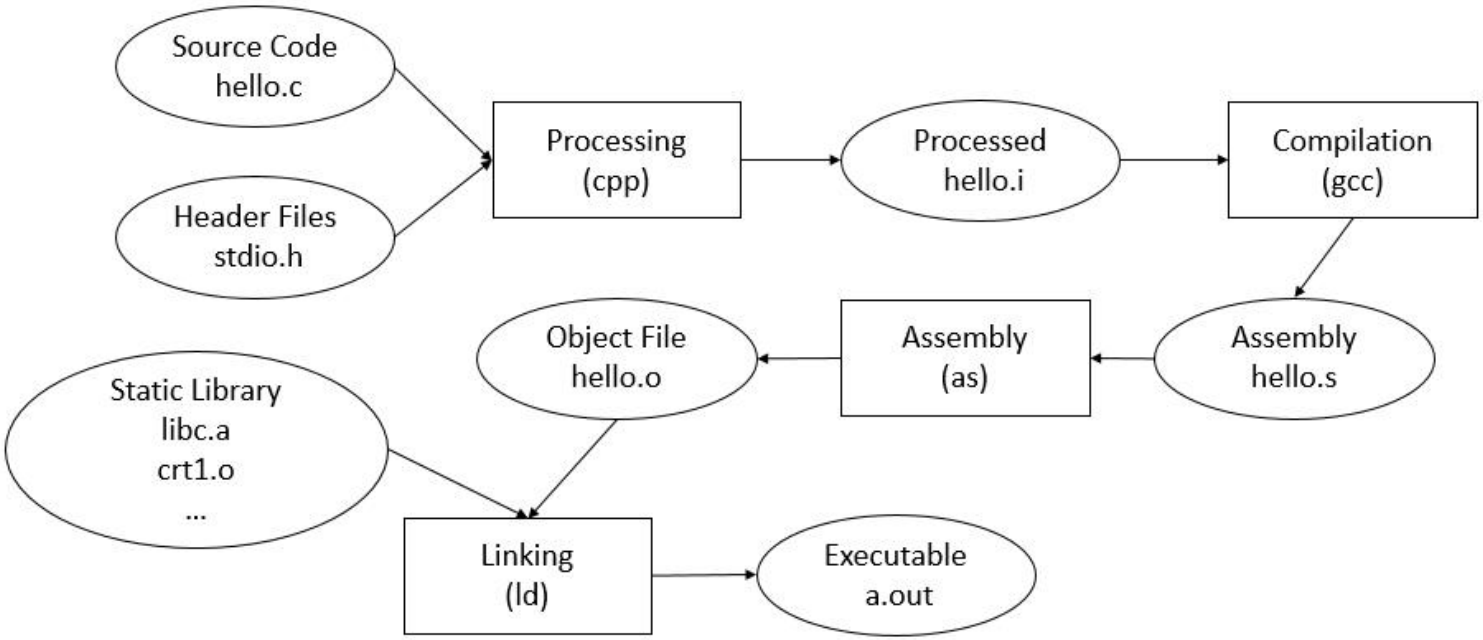
Objects Files
Turning Source Code to Object Code

Programmer-Visible State

- Code in files p1.c p2.c
- Compile with command:gcc -Og p1.c p2.c -o p
  - Use basic optimizations (-Og) [New to recent versions of GCC]
  - Put resulting binary in file p



Objects Files
Turning Source Code to Object Code



Object Files
ELF

Object files are merely collections of blocks of bytes.
Some of these blocks contain program code, others contain program data, and others contain data structures that guide the linker and loader.

Object Files
ELF

Object files come in three forms:

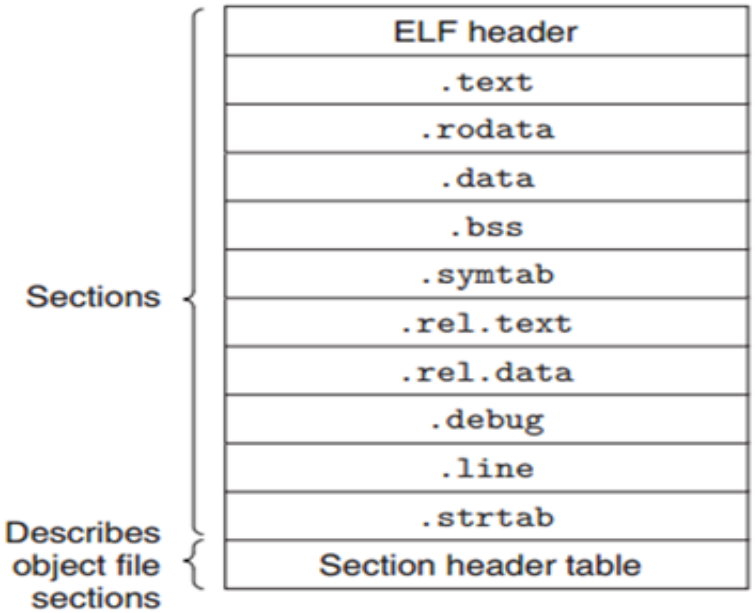
- Relocatable object file
  - Contains binary code and data in a form that can be combined with other relocatable objects files at compile time to create an executable object file.
- Executable object file
  - Contains binary code and data in a form that can be copied directly into memory and executed
- Shared object file
  - A special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time

Objects Files
ELF

Object file formats vary from system to system.

- Early versions of System V Unix used the Common Object File format(COFF)
- Windows NT uses a variant of COFF called the Portable Executable(PE) format
- Modern Unix system–such as Linux, later version of System V Unix, BSD Unix variants, and Sun Solaris–use the Unix Executable and Linkable Format(ELF)

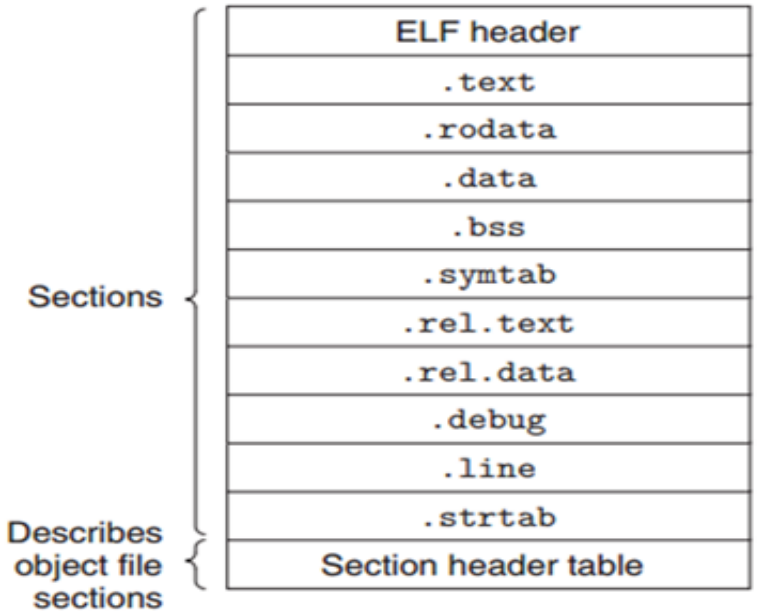
Objects Files
ELF



Typical ELF relocatable object file

The ELF header begins with a 16-byte sequence that describes the word size and byte ordering of the system that generated the file.

Objects Files
ELF



Typical ELF relocatable object file

The section header table describes the locations and sizes of the various sections, and contains a fixed sized entry for each section in the object file.

Objects Files
Symbols and Symbol Tables

A symbol table with information about functions and global variables that are defined and referenced in the program. Each relocatable object module, m, has a symbol table that contains information about the symbols that are defined and referenced by m.

Objects Files
Symbols and Symbol Tables

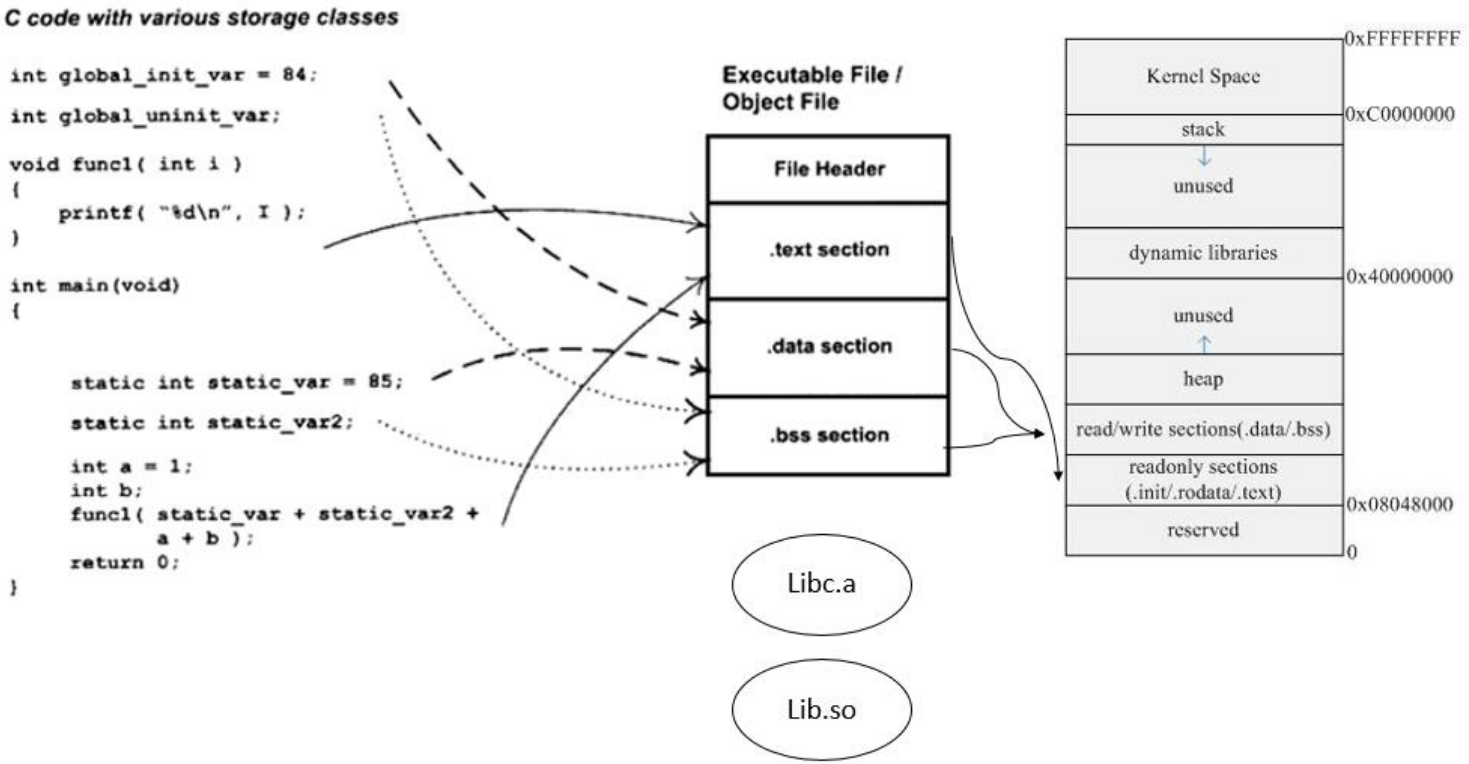
- Global symbols that are defined by module m and that can be referenced by other modules. Global linkers symbols correspond to nonstatic C functions and global variables that are defined without the C static attribute.
- Global symbols that are referenced by module m but defined by some other module. Such symbols are called externals and correspond to C functions and variables that are defined in other modules.
- Local symbols that are defined and referenced exclusively by module m. Some local linker symbols correspond to C functions and global variables that are defined with the static attribute. These symbols are visible anywhere within module m, but cannot be referenced by other modules. The sections in an object file and the name of the source file that correspond to module m also get local symbols.

Objects Files
Symbols and Symbol Tables

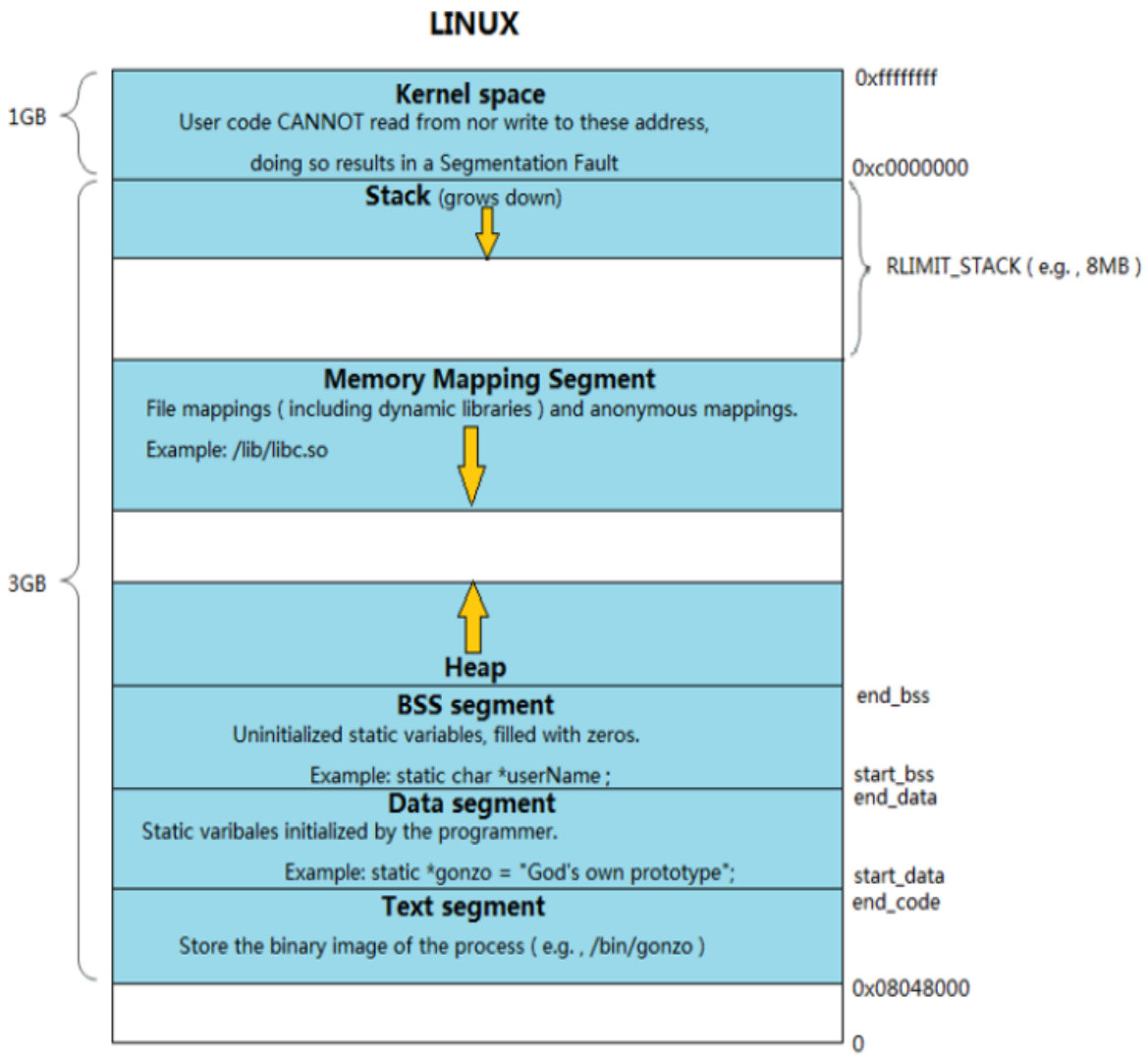

```
1  typedef struct {
2      int name;       /* String table offset */
3      int value;      /* Section offset, or VM address */
4      int size;       /* Object size in bytes */
5      char type:4,    /* Data, func, section, or src file name (4 bits) */
6           binding:4; /* Local or global (4 bits) */
7      char reserved;  /* Unused */
8      char section;   /* Section header index, ABS, UNDEF, */
9                      /* Or COMMON  */
10 } Elf_Symbol;
```

ELF symbol table entry. type and binding are four bits each.

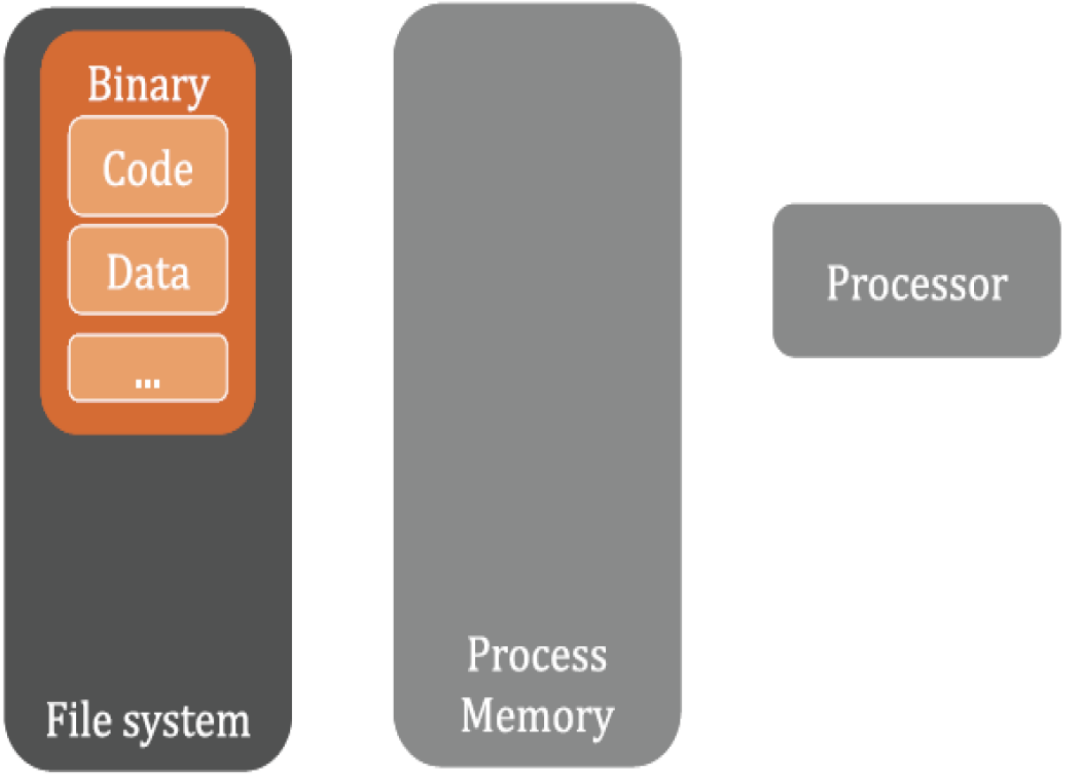
Objects Files
ELF



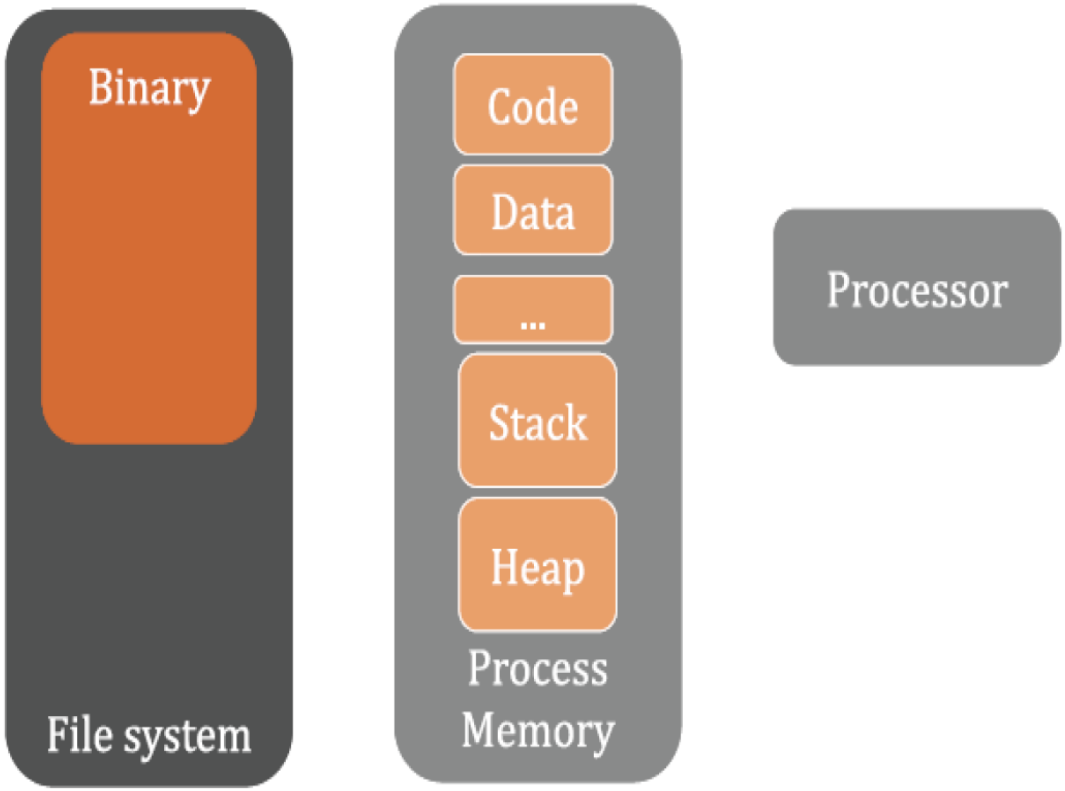
Memory Layout



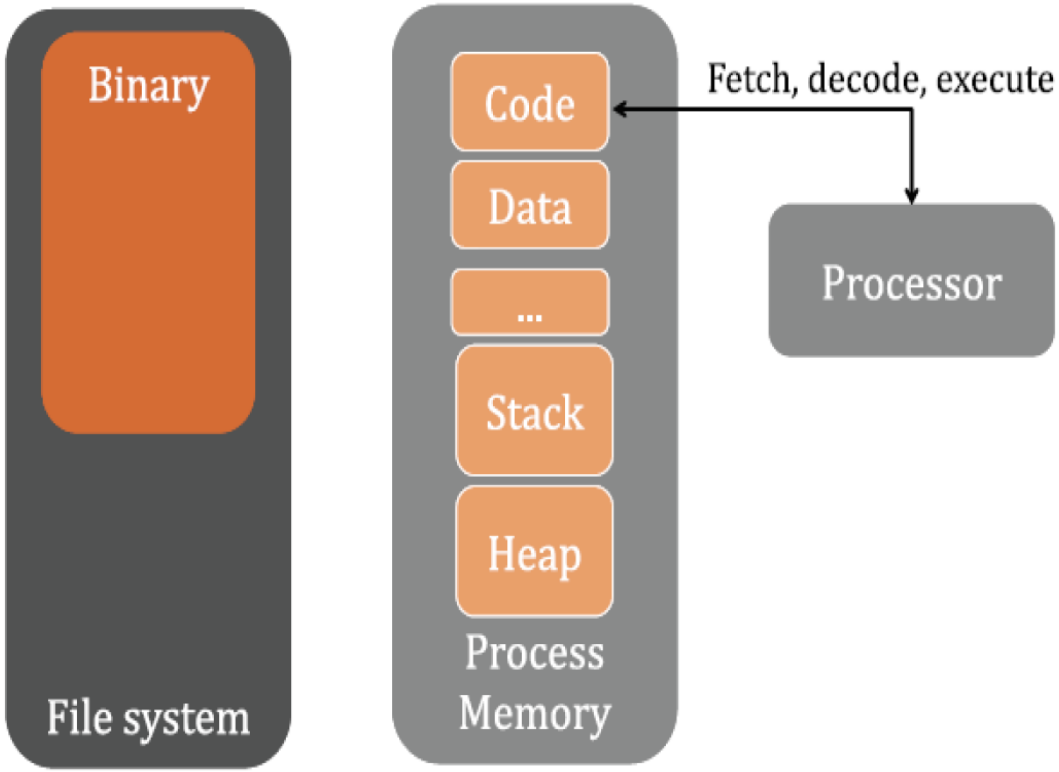
Memory Layout
How CPU Executes Programs?



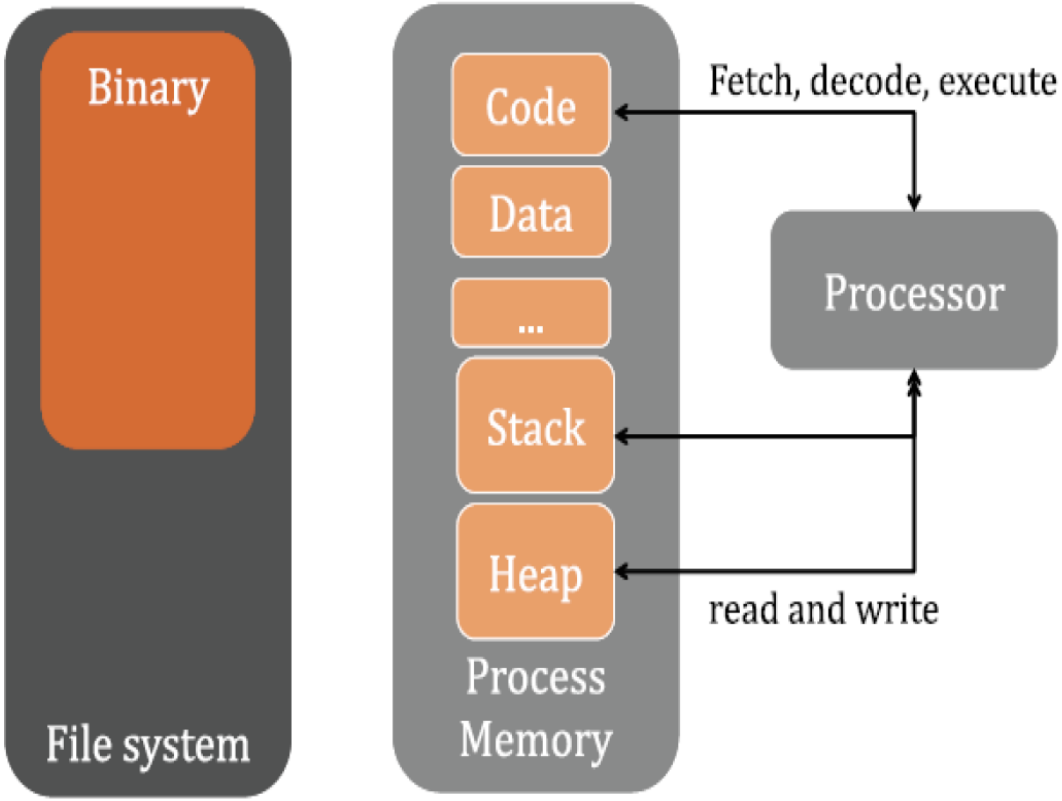
Memory Layout
How CPU Executes Programs?



Memory Layout
How CPU Executes Programs?

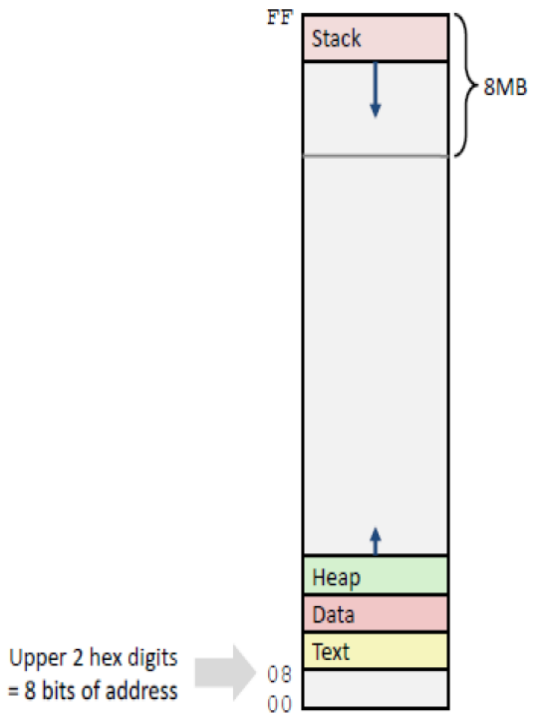


Memory Layout
How CPU Executes Programs?



Memory Layout
IA32 Linux Memory Layout

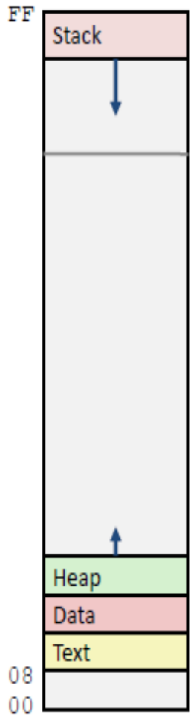
- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only



Memory Layout
IA32 Linux Memory Layout

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB*/

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?

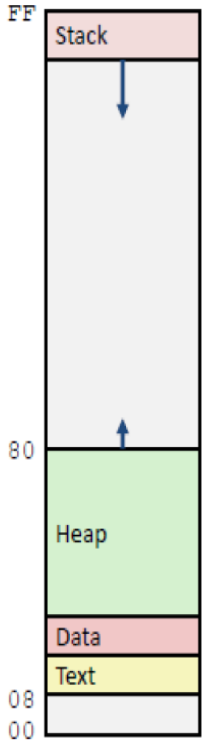


Memory Layout
IA32 Linux Memory Layout

address range ~2^{32}

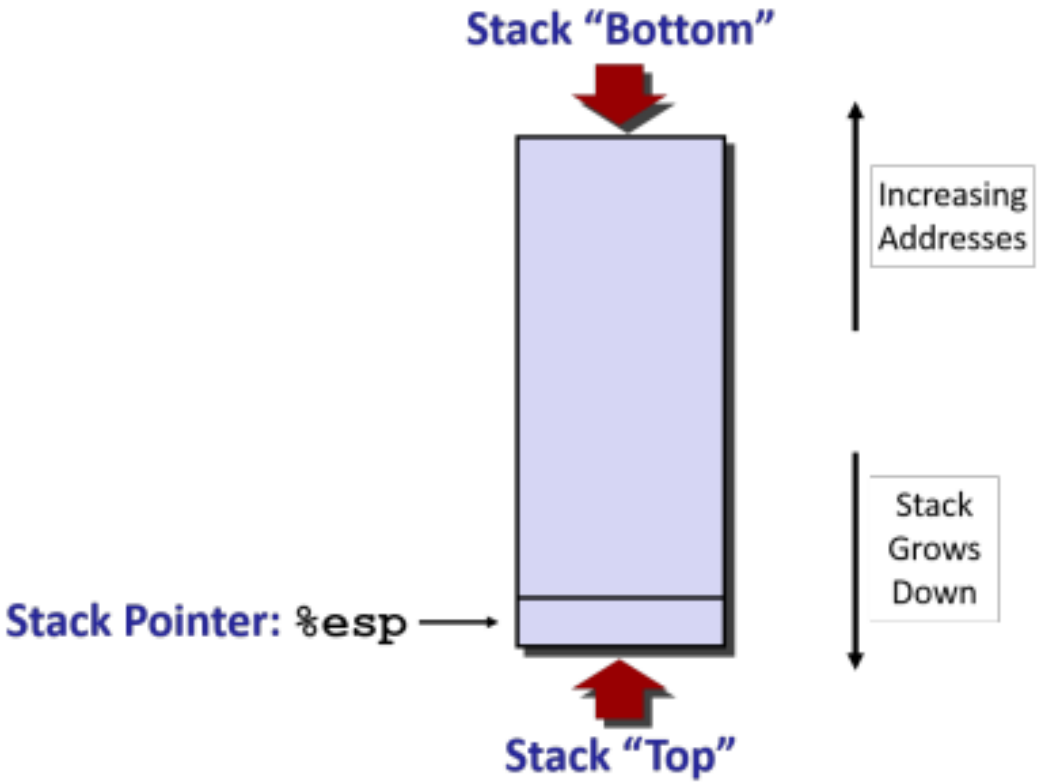
`$esp`	`0xffffbcd0`
`p3`	`0x65586008`
`p1`	`0x55585008`
`p4`	`0x1904a110`
`p2`	`0x1904a008`
`&p2`	`0x18049760`
`&beyond`	`0x08049744`
`big_array`	`0x18049780`
`huge_array`	`0x08049760`
`main()`	`0x080483c6`
`useless()`	`0x08049744`
final `malloc()`	`0x006be166`

`malloc()` is dynamically linked
address determined at runtime



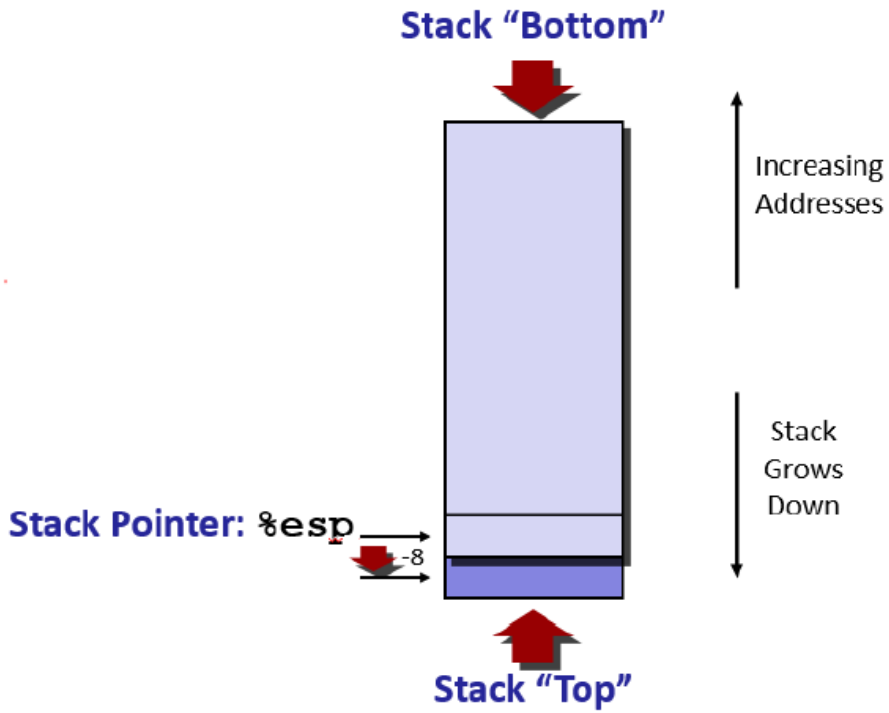
IA32 Stack

- 1.Region of memory managed with stack discipline
- 2.Grows toward lower addresses
- 3.Register %esp contains lowest stack address



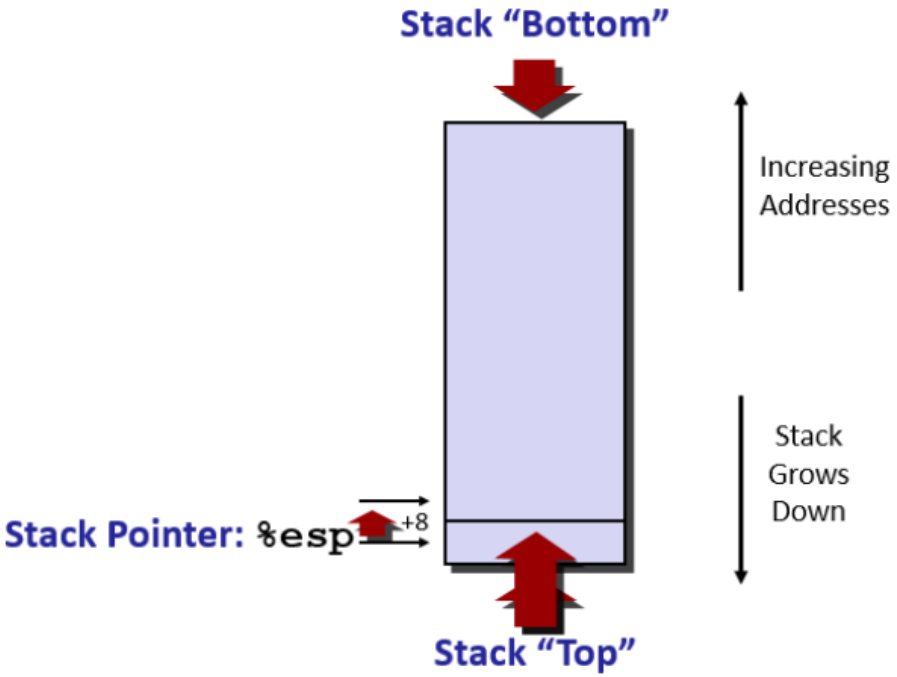
Push Src

- 1.Fetch operand at Src
- 2.Decrement %esp by 4
- 3.Write operand at address given by %esp

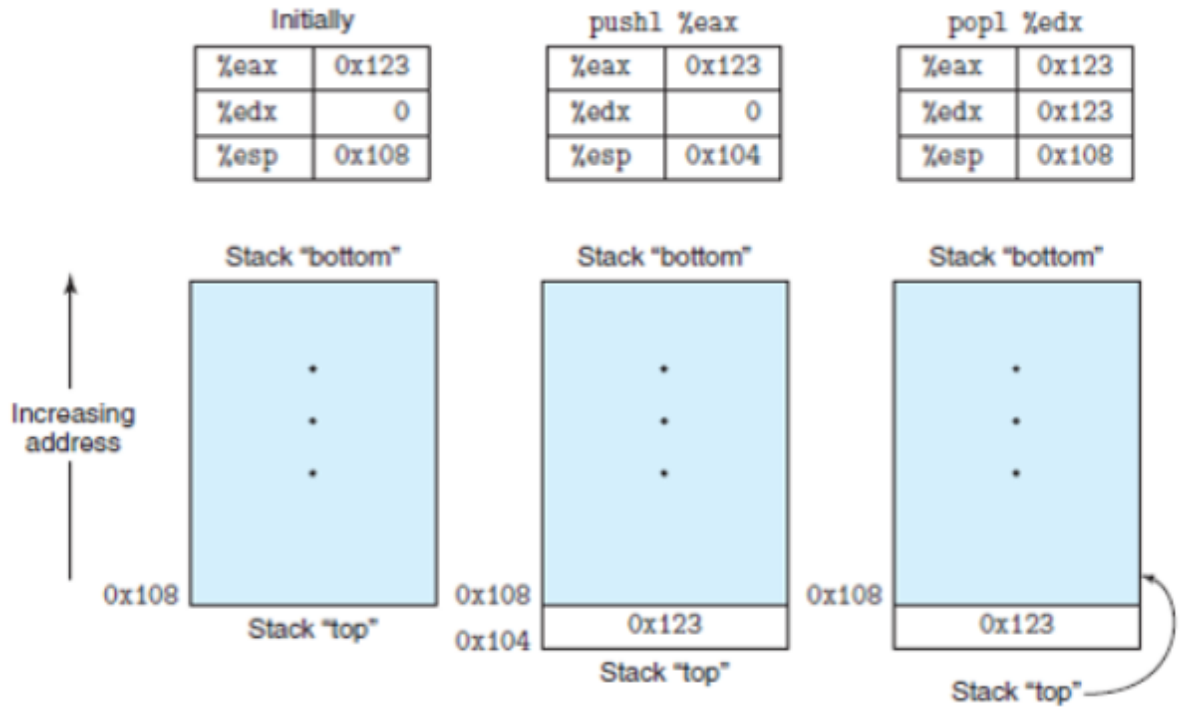


Push Src

- 1.Read value at address given by %esp
- 2.Increment %esp
- 3.Store value at Dest(must be register)



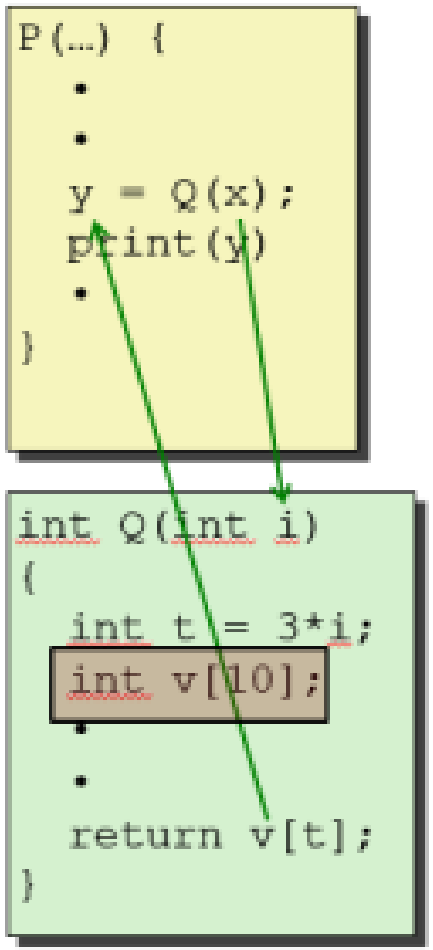
Procedures
Stack Structure



Procedures

- A procedure call involves passing both data(in the form of procedure parameters and return values) and control from one part of a program to another. In addition, it must allocate space for the local variables of the procedures on entry and deallocate them on exit. Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

Procedures
Mechanisms in Procedures



Procedures
Mechanisms in Procedures

Use stack to support procedure call and return.
Procedure call: call label

- Push return address on stack
- Jump to label

Return address

- Address of the next instruction right after call
- Example from disassembly

Procedure return: ret

- Pop address from stack
- Jump to address

Practice Problem:

A C function `fun` has the following code body:

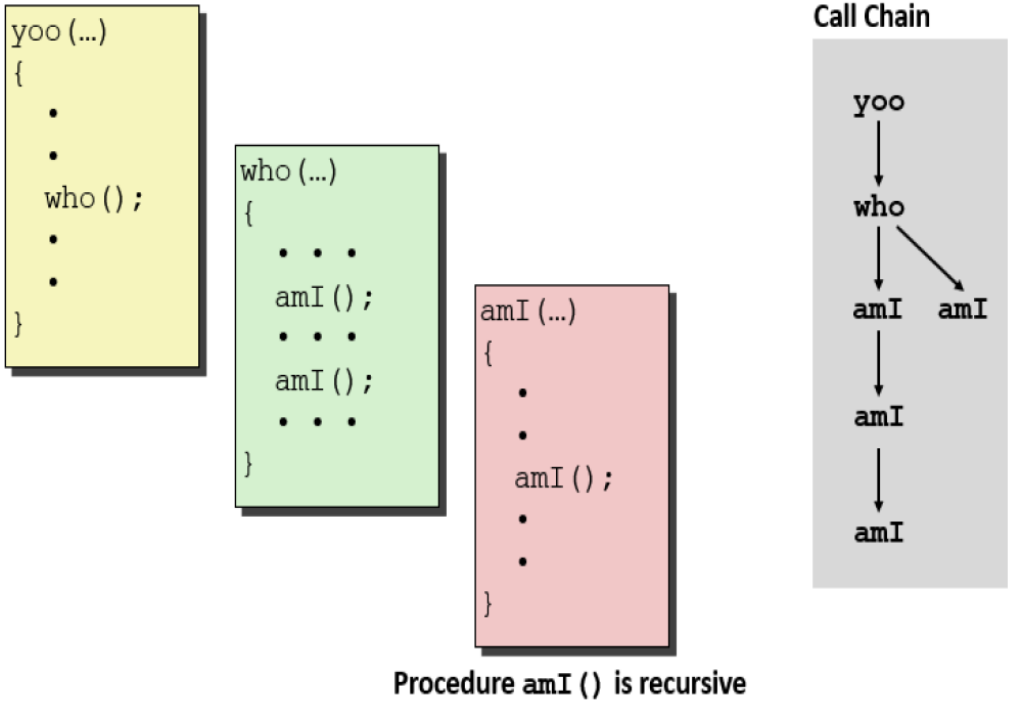
```
*p = d;
return x-c;
```

The IA32 code implementing this body is as follows:

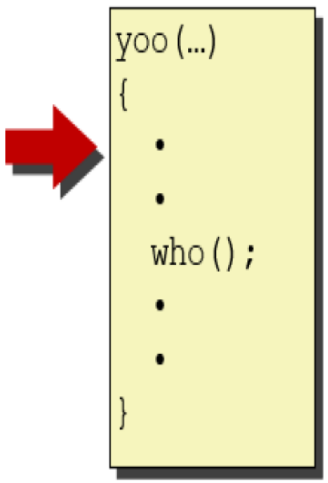
```
1     movsbl  12(%ebp),%edx
2     movl    16(%ebp), %eax
3     movl    %edx, (%eax)
4     movswl  8(%ebp),%eax
5     movl    20(%ebp), %edx
6     subl    %eax, %edx
7     movl    %edx, %eax
```

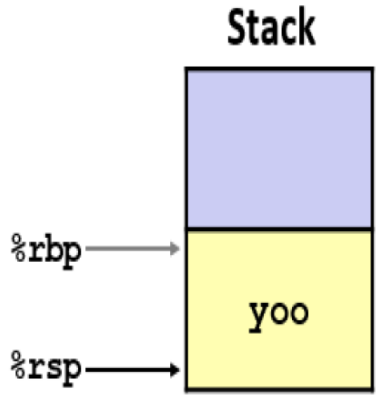
Write a prototype for function `fun`, showing the types and ordering of the arguments `p`, `d`, `x`, and `c`.

Procedures
Call Chain

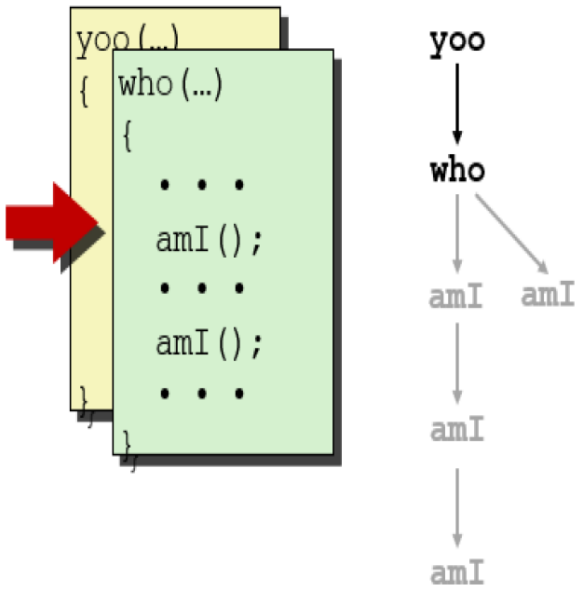


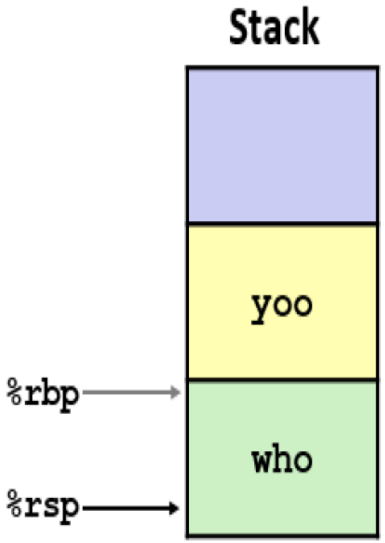
Procedures
Call Chain



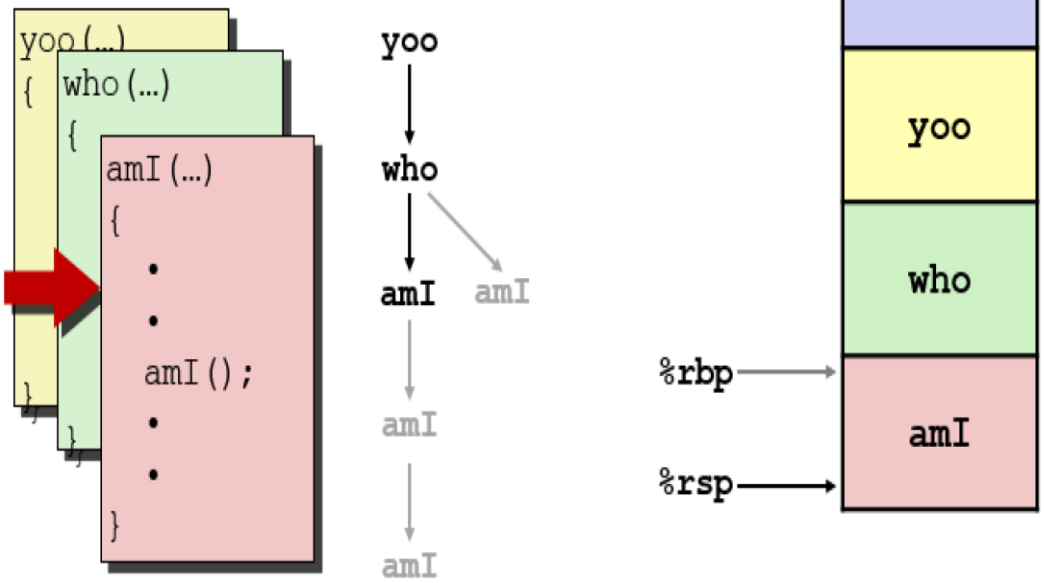


Procedures
Call Chain

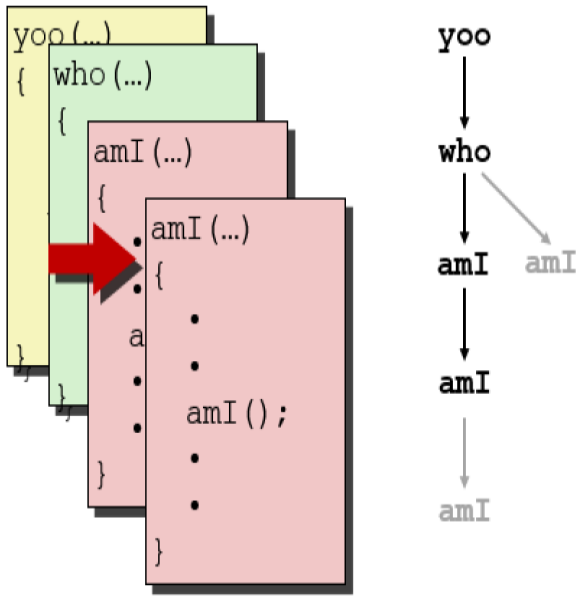


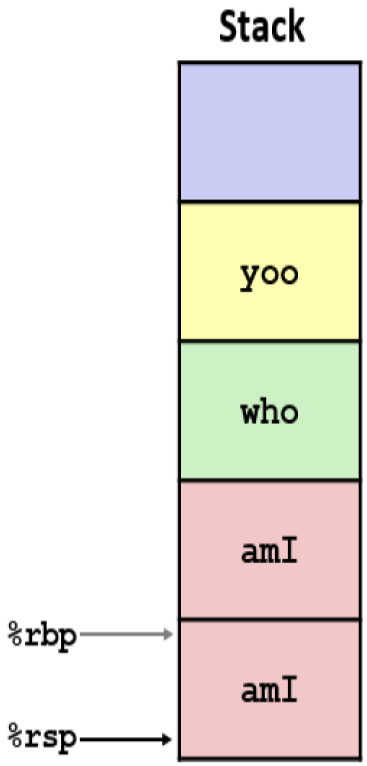


Procedures
Call Chain

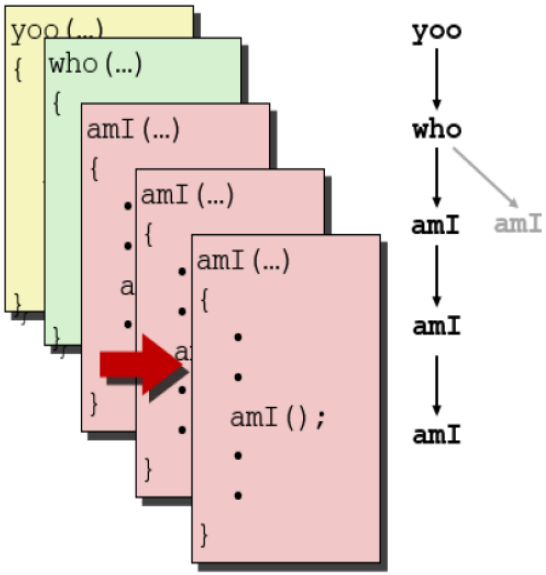


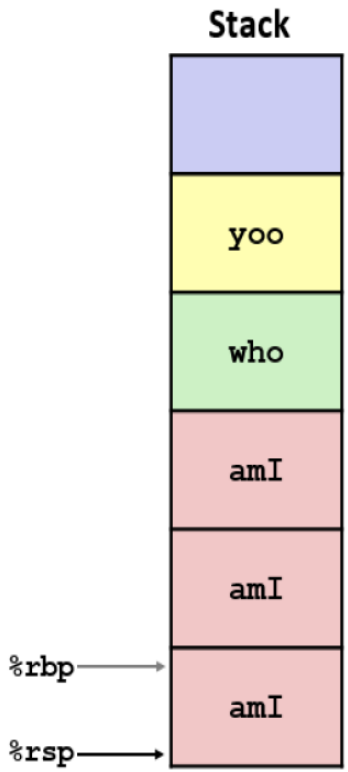
Procedures
Call Chain



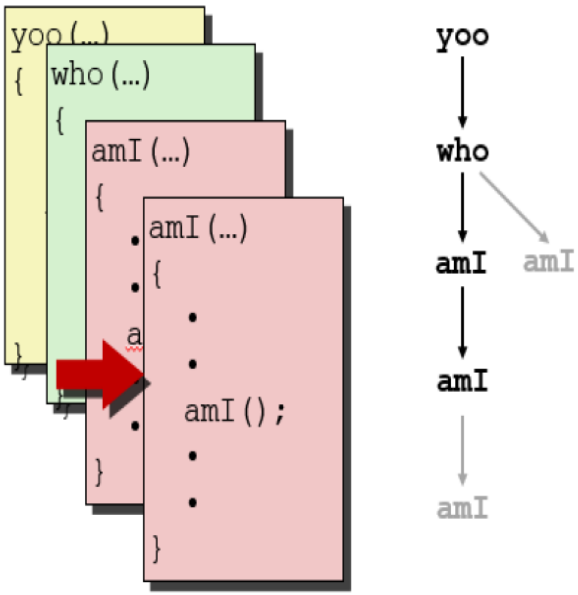


Procedures
Call Chain

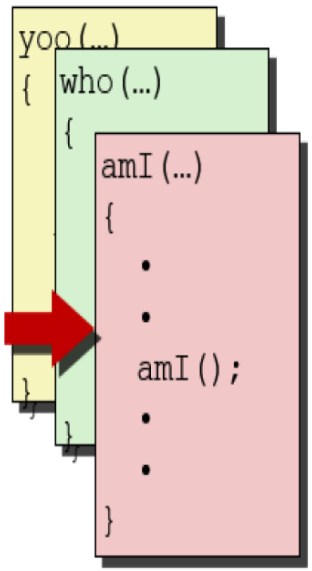


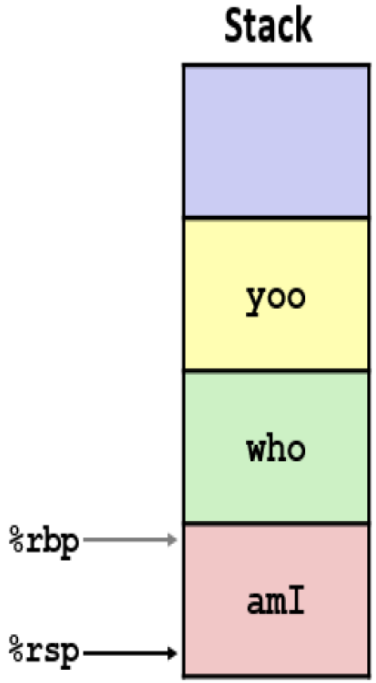


Procedures
Call Chain

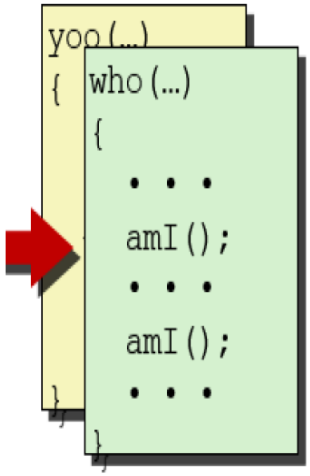


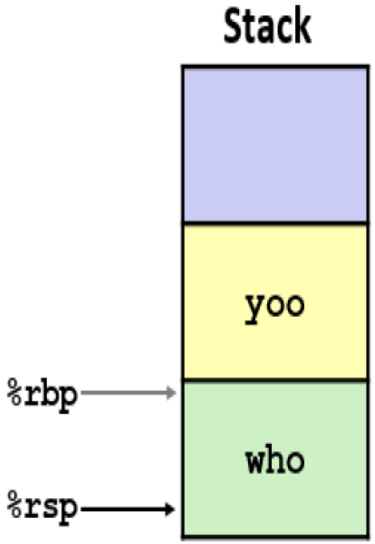
Procedures
Call Chain



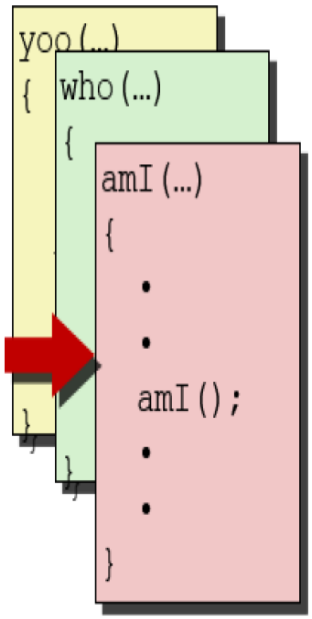


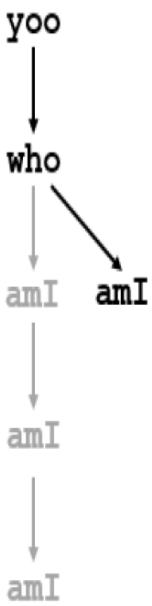
Procedures
Call Chain

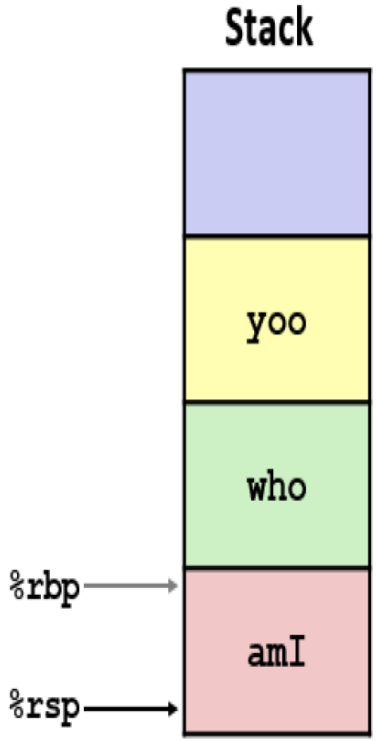




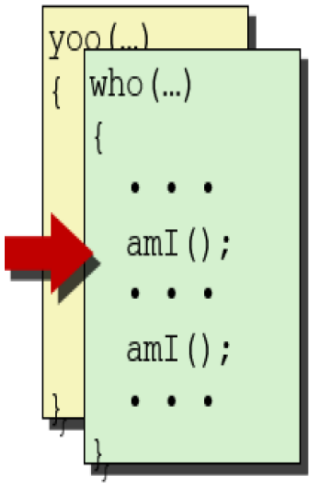
Procedures
Call Chain

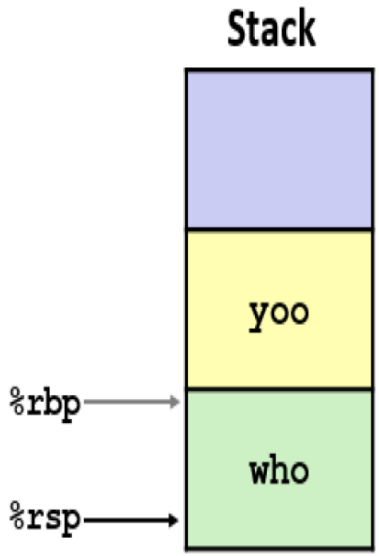




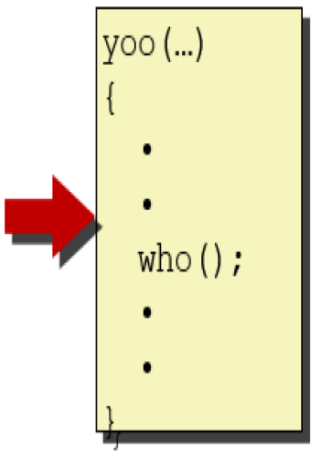


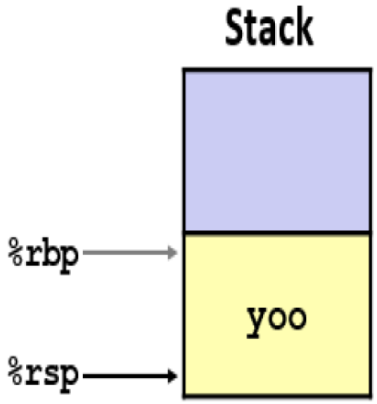
Procedures
Call Chain





Procedures
Call Chain





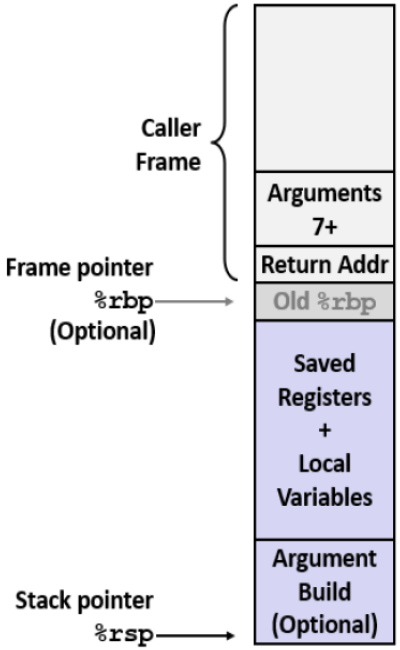
Procedures
Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- "Argument build:" Parameters for function about to call
- Local variables (If can¡¯t keep in registers)
- Saved register context
- Old frame pointer (optional)

Caller Stack Frame

- Return Address
  - Pushed by call instruction
- Arguments for this call



Example: incr

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

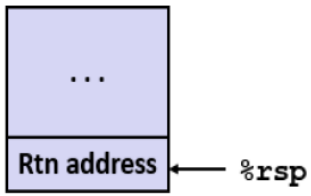
Register	Use(s)
%rdi	Argument p
%rsi	Argument val, y
%rax	x, Return value

Example: Calling incr 1

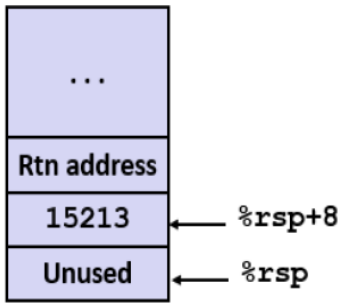
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure

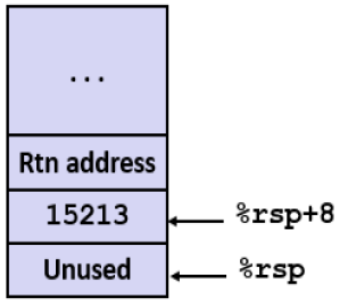


Example: Calling incr 2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



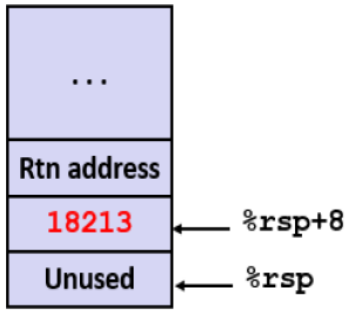
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr 3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



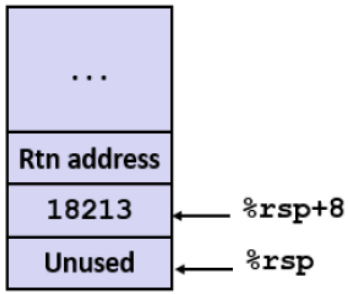
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr 4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

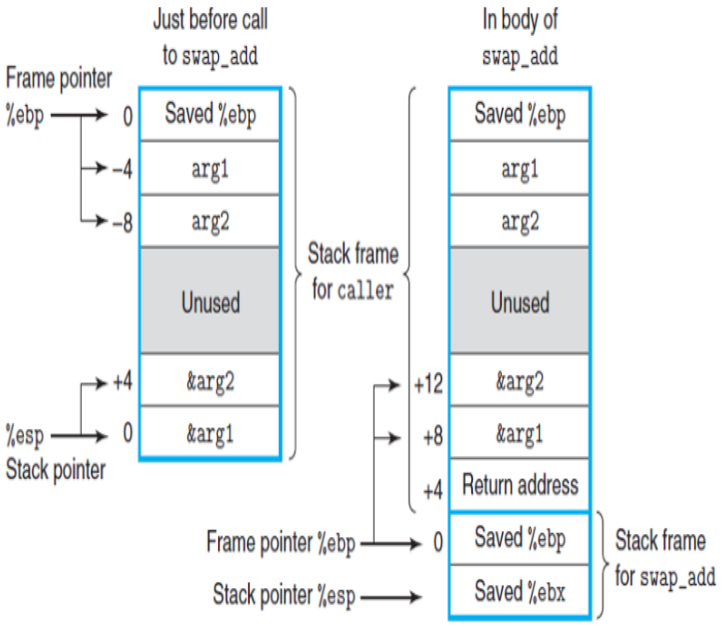
Stack Structure



Register	Use(s)
%rax	Return value

Updated Stack Structure

Example: Calling swap_add



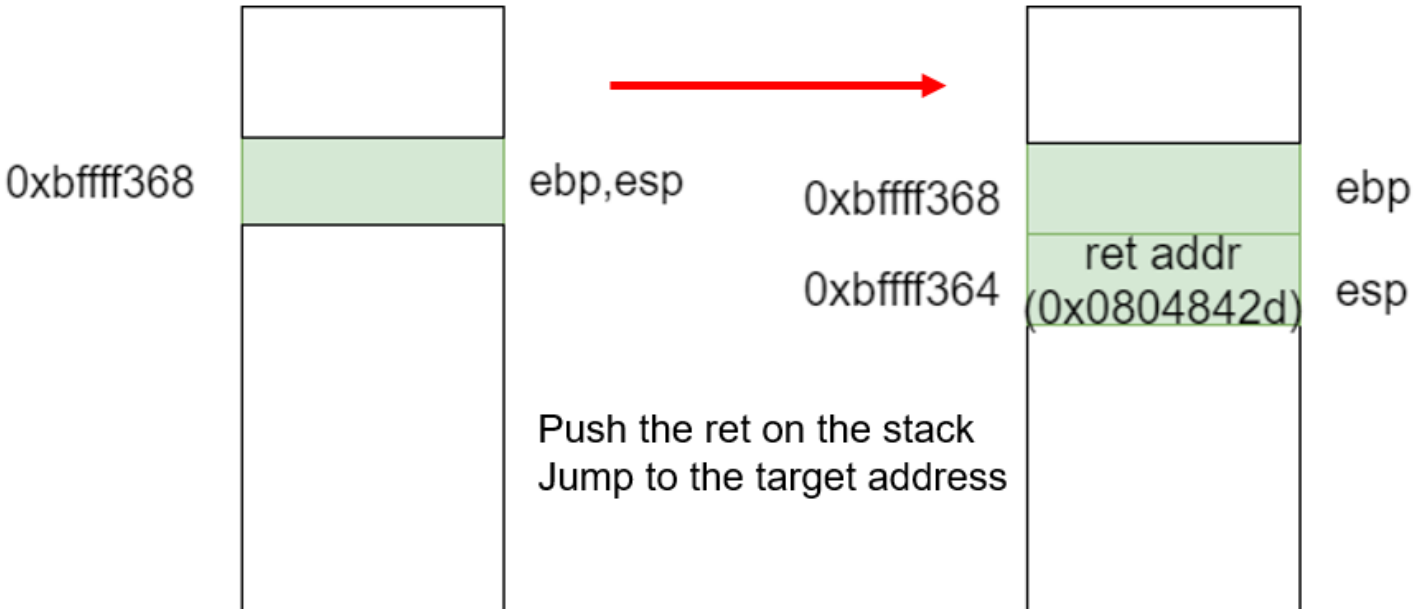

```
1  int swap_add(int *xp, int *yp)
2  {
3      int x = *xp;
4      int y = *yp;
5
6      *xp = y;
7      *yp = x;
8      return x + y;
9  }
10
11 int caller()
12 {
13     int arg1 = 534;
14     int arg2 = 1057;
15     int sum = swap_add(&arg1, &arg2);
16     int diff = arg1 - arg2;
17
18     return sum * diff;
19 }
```

Procedures
Linux Stack Frame

```c
#include <stdio.h>
int swap_add(int *xp, int *yp)
{
        int x=*xp;
        int y=*yp;
        *xp=y;
        *yp=x;
        return x+y;
}

int caller()
{
        int arg1=534;
        int arg2=1057;
        int sum=swap_add(&arg1,&arg2);
        int diff=arg1-arg2;
        return sum*diff;
}

int main()
{
        caller();
}
```

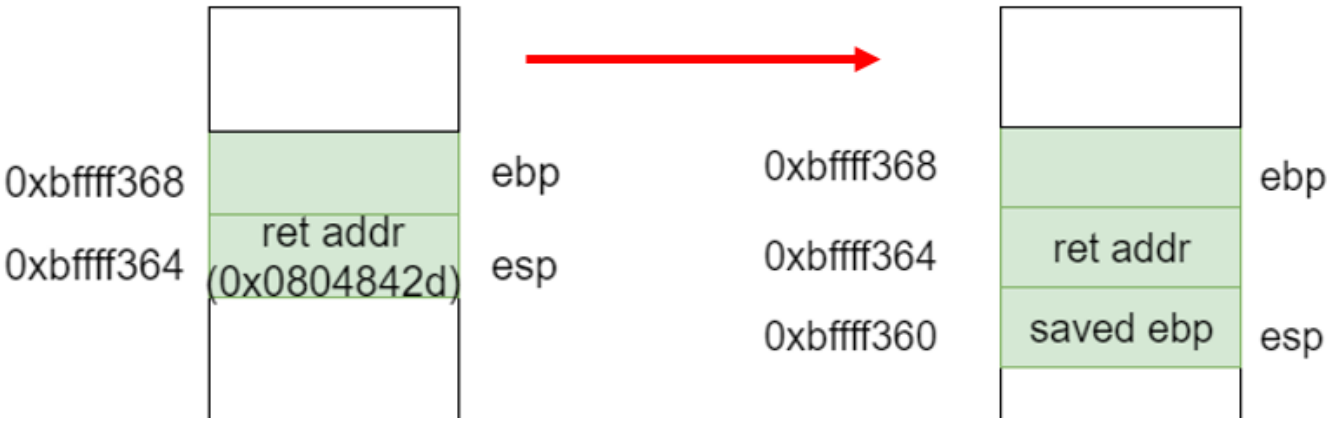
Procedures
Linux Stack Frame


```
Dump of assembler code for function main:
   0x08048425 <+0>:     push   %ebp
   0x08048426 <+1>:     mov    %esp,%ebp
=> 0x08048428 <+3>:     call   0x80483e4 <caller>
   0x0804842d <+8>:     pop    %ebp
   0x0804842e <+9>:     ret
End of assembler dump.
```



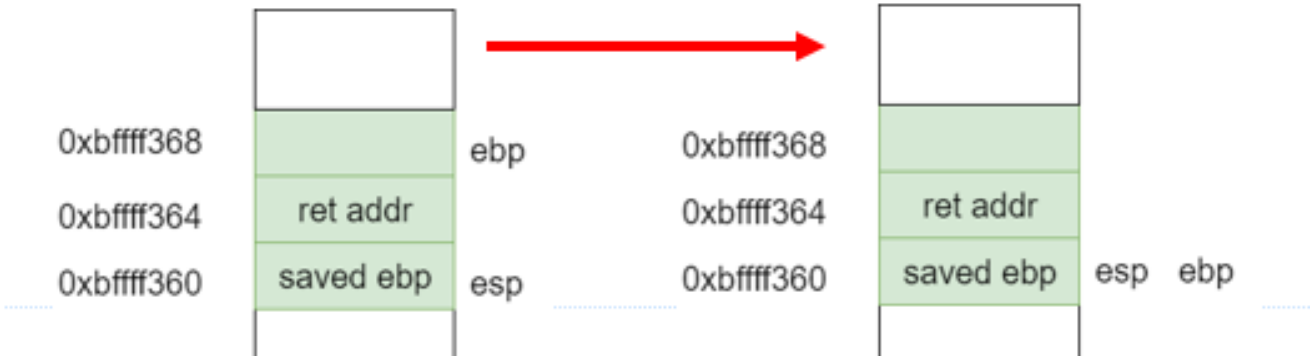
Procedures
Linux Stack Frame

```
Dump of assembler code for function caller:
=> 0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     movl   $0x216,-0x10(%ebp)
   0x080483f1 <+13>:    movl   $0x421,-0xc(%ebp)
   0x080483f8 <+20>:    lea    -0xc(%ebp),%eax
   0x080483fb <+23>:    mov    %eax,0x4(%esp)
   0x080483ff <+27>:    lea    -0x10(%ebp),%eax
   0x08048402 <+30>:    mov    %eax,(%esp)
   0x08048405 <+33>:    call   0x80483b4 <swap_add>
   0x0804840a <+38>:    mov    %eax,-0x8(%ebp)
   0x0804840d <+41>:    mov    -0x10(%ebp),%edx
   0x08048410 <+44>:    mov    -0xc(%ebp),%eax
   0x08048413 <+47>:    mov    %edx,%ecx
   0x08048415 <+49>:    sub    %eax,%ecx
   0x08048417 <+51>:    mov    %ecx,%eax
   0x08048419 <+53>:    mov    %eax,-0x4(%ebp)
   0x0804841c <+56>:    mov    -0x8(%ebp),%eax
   0x0804841f <+59>:    imul   -0x4(%ebp),%eax
   0x08048423 <+63>:    leave
   0x08048424 <+64>:    ret
End of assembler dump.
```



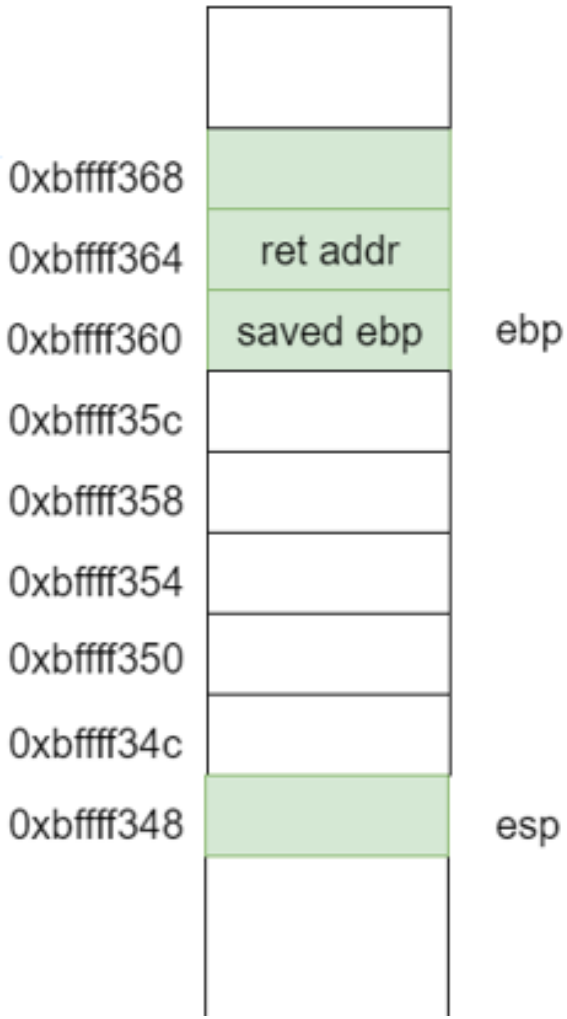
Procedures
Linux Stack Frame

```
Dump of assembler code for function caller:
   0x080483e4 <+0>:     push   %ebp
=> 0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     movl   $0x216,-0x10(%ebp)
   0x080483f1 <+13>:    movl   $0x421,-0xc(%ebp)
   0x080483f8 <+20>:    lea    -0xc(%ebp),%eax
   0x080483fb <+23>:    mov    %eax,0x4(%esp)
   0x080483ff <+27>:    lea    -0x10(%ebp),%eax
   0x08048402 <+30>:    mov    %eax,(%esp)
   0x08048405 <+33>:    call   0x80483b4 <swap_add>
   0x0804840a <+38>:    mov    %eax,-0x8(%ebp)
   0x0804840d <+41>:    mov    -0x10(%ebp),%edx
   0x08048410 <+44>:    mov    -0xc(%ebp),%eax
   0x08048413 <+47>:    mov    %edx,%ecx
   0x08048415 <+49>:    sub    %eax,%ecx
   0x08048417 <+51>:    mov    %ecx,%eax
   0x08048419 <+53>:    mov    %eax,-0x4(%ebp)
   0x0804841c <+56>:    mov    -0x8(%ebp),%eax
   0x0804841f <+59>:    imul   -0x4(%ebp),%eax
   0x08048423 <+63>:    leave
   0x08048424 <+64>:    ret
End of assembler dump.
```



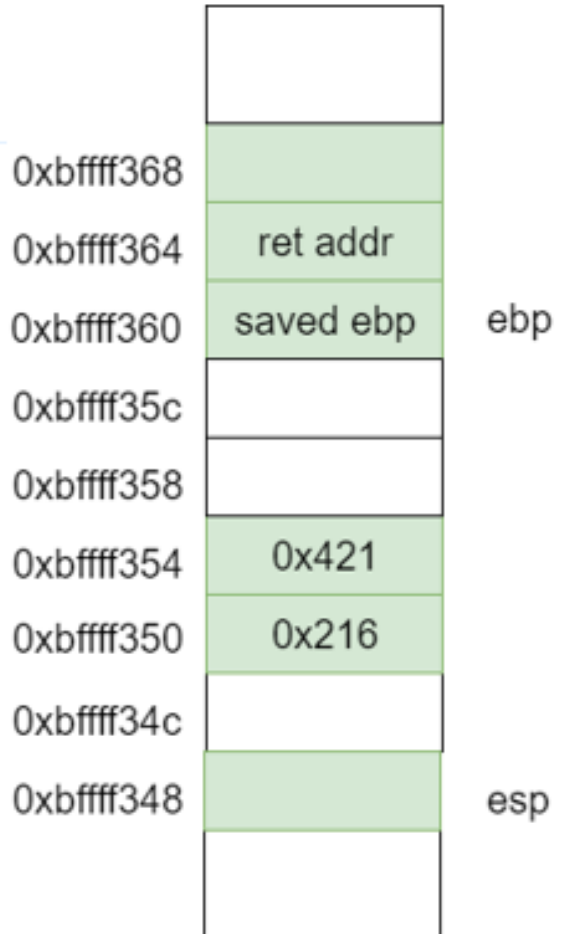
Procedures
Linux Stack Frame

```
Dump of assembler code for function caller:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
=> 0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     movl   $0x216,-0x10(%ebp)
   0x080483f1 <+13>:    movl   $0x421,-0xc(%ebp)
   0x080483f8 <+20>:    lea    -0xc(%ebp),%eax
   0x080483fb <+23>:    mov    %eax,0x4(%esp)
   0x080483ff <+27>:    lea    -0x10(%ebp),%eax
   0x08048402 <+30>:    mov    %eax,(%esp)
   0x08048405 <+33>:    call   0x80483b4 <swap_add>
   0x0804840a <+38>:    mov    %eax,-0x8(%ebp)
   0x0804840d <+41>:    mov    -0x10(%ebp),%edx
   0x08048410 <+44>:    mov    -0xc(%ebp),%eax
   0x08048413 <+47>:    mov    %edx,%ecx
   0x08048415 <+49>:    sub    %eax,%ecx
   0x08048417 <+51>:    mov    %ecx,%eax
   0x08048419 <+53>:    mov    %eax,-0x4(%ebp)
   0x0804841c <+56>:    mov    -0x8(%ebp),%eax
   0x0804841f <+59>:    imul   -0x4(%ebp),%eax
   0x08048423 <+63>:    leave
   0x08048424 <+64>:    ret
End of assembler dump.
```



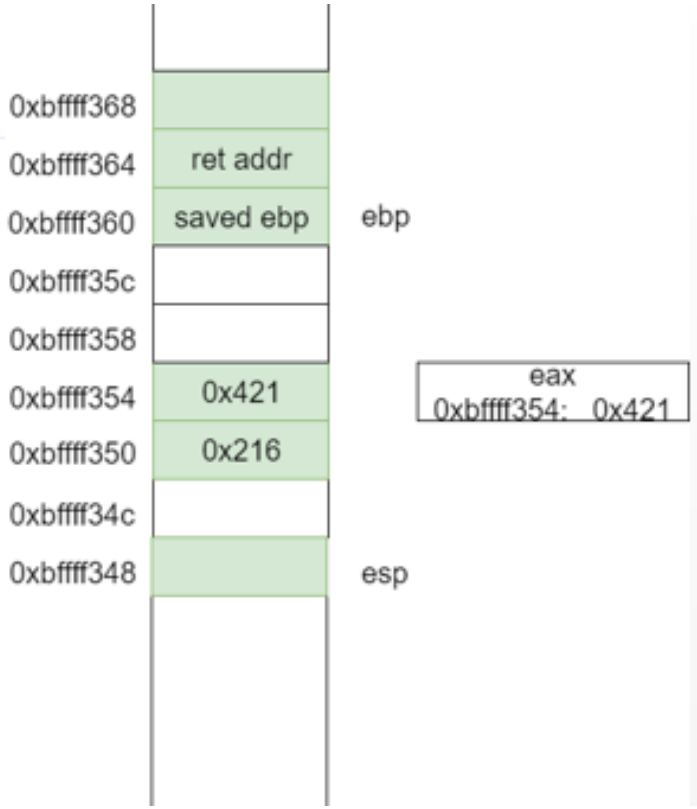
Procedures
Linux Stack Frame

```
Dump of assembler code for function caller:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     movl   $0x216,-0x10(%ebp)
=> 0x080483f1 <+13>:    movl   $0x421,-0xc(%ebp)
   0x080483f8 <+20>:    lea    -0xc(%ebp),%eax
   0x080483fb <+23>:    mov    %eax,0x4(%esp)
   0x080483ff <+27>:    lea    -0x10(%ebp),%eax
   0x08048402 <+30>:    mov    %eax,(%esp)
   0x08048405 <+33>:    call   0x80483b4 <swap_add>
   0x0804840a <+38>:    mov    %eax,-0x8(%ebp)
   0x0804840d <+41>:    mov    -0x10(%ebp),%edx
   0x08048410 <+44>:    mov    -0xc(%ebp),%eax
   0x08048413 <+47>:    mov    %edx,%ecx
   0x08048415 <+49>:    sub    %eax,%ecx
   0x08048417 <+51>:    mov    %ecx,%eax
   0x08048419 <+53>:    mov    %eax,-0x4(%ebp)
   0x0804841c <+56>:    mov    -0x8(%ebp),%eax
   0x0804841f <+59>:    imul   -0x4(%ebp),%eax
   0x08048423 <+63>:    leave
   0x08048424 <+64>:    ret
End of assembler dump.
```



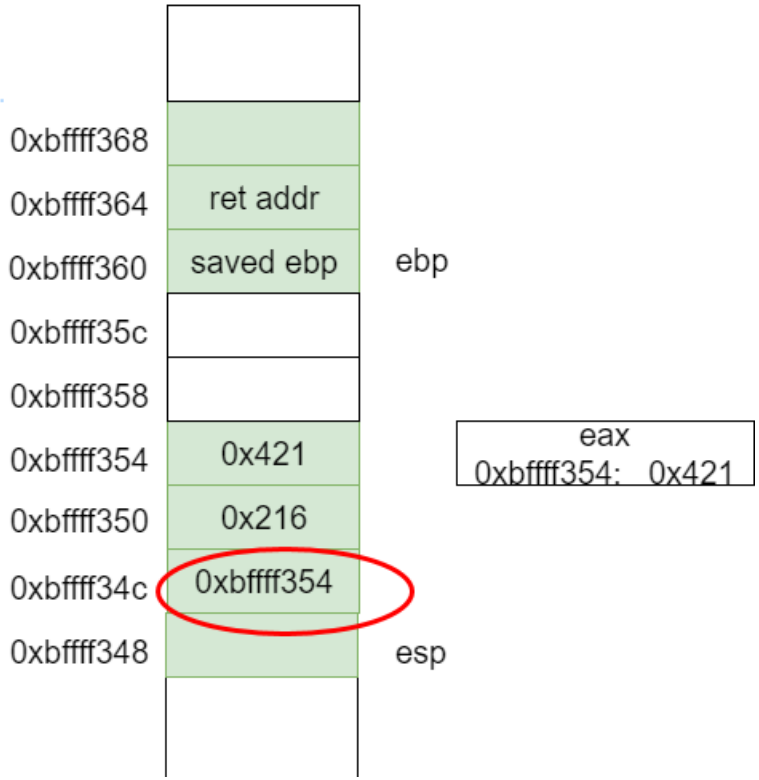
Procedures
Linux Stack Frame

```
Dump of assembler code for function caller:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     movl   $0x216,-0x10(%ebp)
   0x080483f1 <+13>:    movl   $0x421,-0xc(%ebp)
=> 0x080483f8 <+20>:    lea    -0xc(%ebp),%eax
   0x080483fb <+23>:    mov    %eax,0x4(%esp)
   0x080483ff <+27>:    lea    -0x10(%ebp),%eax
   0x08048402 <+30>:    mov    %eax,(%esp)
   0x08048405 <+33>:    call   0x80483b4 <swap_add>
   0x0804840a <+38>:    mov    %eax,-0x8(%ebp)
   0x0804840d <+41>:    mov    -0x10(%ebp),%edx
   0x08048410 <+44>:    mov    -0xc(%ebp),%eax
   0x08048413 <+47>:    mov    %edx,%ecx
   0x08048415 <+49>:    sub    %eax,%ecx
   0x08048417 <+51>:    mov    %ecx,%eax
   0x08048419 <+53>:    mov    %eax,-0x4(%ebp)
   0x0804841c <+56>:    mov    -0x8(%ebp),%eax
   0x0804841f <+59>:    imul   -0x4(%ebp),%eax
   0x08048423 <+63>:    leave
   0x08048424 <+64>:    ret
End of assembler dump.
```



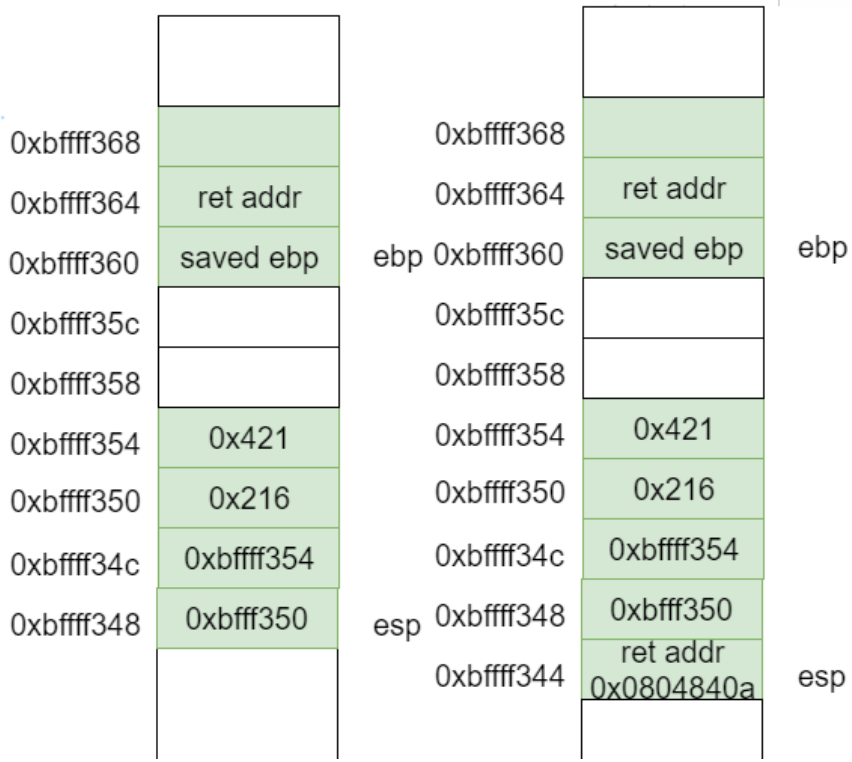
Procedures
Linux Stack Frame

```
Dump of assembler code for function caller:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     movl   $0x216,-0x10(%ebp)
   0x080483f1 <+13>:    movl   $0x421,-0xc(%ebp)
   0x080483f8 <+20>:    lea    -0xc(%ebp),%eax
=> 0x080483fb <+23>:    mov    %eax,0x4(%esp)
   0x080483ff <+27>:    lea    -0x10(%ebp),%eax
   0x08048402 <+30>:    mov    %eax,(%esp)
   0x08048405 <+33>:    call   0x80483b4 <swap_add>
   0x0804840a <+38>:    mov    %eax,-0x8(%ebp)
   0x0804840d <+41>:    mov    -0x10(%ebp),%edx
   0x08048410 <+44>:    mov    -0xc(%ebp),%eax
   0x08048413 <+47>:    mov    %edx,%ecx
   0x08048415 <+49>:    sub    %eax,%ecx
   0x08048417 <+51>:    mov    %ecx,%eax
   0x08048419 <+53>:    mov    %eax,-0x4(%ebp)
   0x0804841c <+56>:    mov    -0x8(%ebp),%eax
   0x0804841f <+59>:    imul   -0x4(%ebp),%eax
   0x08048423 <+63>:    leave
   0x08048424 <+64>:    ret
End of assembler dump.
```



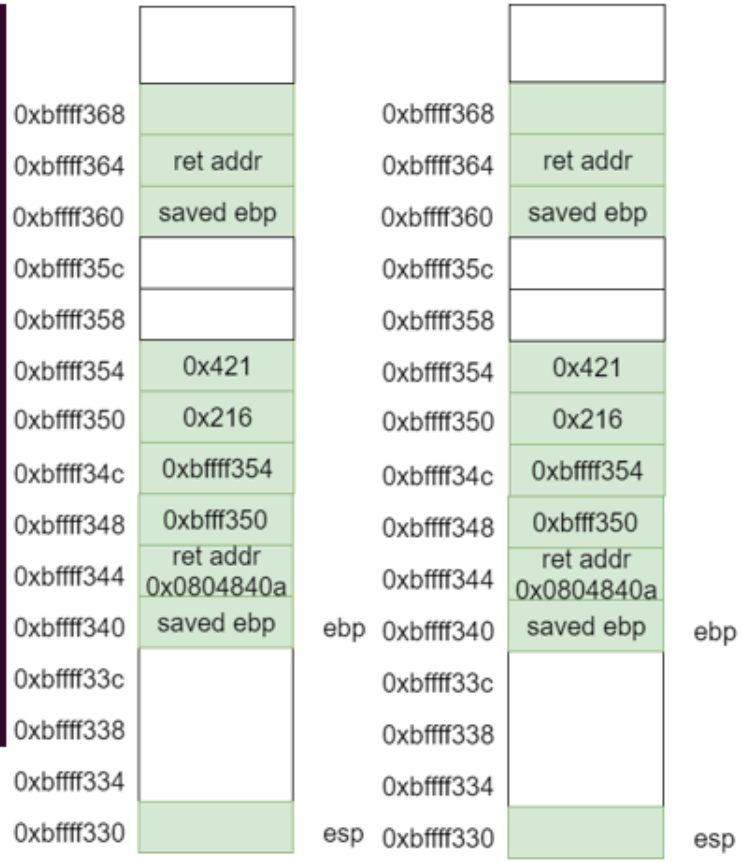
Procedures
Linux Stack Frame

```
Dump of assembler code for function caller:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     movl   $0x216,-0x10(%ebp)
   0x080483f1 <+13>:    movl   $0x421,-0xc(%ebp)
   0x080483f8 <+20>:    lea    -0xc(%ebp),%eax
   0x080483fb <+23>:    mov    %eax,0x4(%esp)
   0x080483ff <+27>:    lea    -0x10(%ebp),%eax
   0x08048402 <+30>:    mov    %eax,(%esp)
=> 0x08048405 <+33>:    call   0x80483b4 <swap_add>
   0x0804840a <+38>:    mov    %eax,-0x8(%ebp)
   0x0804840d <+41>:    mov    -0x10(%ebp),%edx
   0x08048410 <+44>:    mov    -0xc(%ebp),%eax
   0x08048413 <+47>:    mov    %edx,%ecx
   0x08048415 <+49>:    sub    %eax,%ecx
   0x08048417 <+51>:    mov    %ecx,%eax
   0x08048419 <+53>:    mov    %eax,-0x4(%ebp)
   0x0804841c <+56>:    mov    -0x8(%ebp),%eax
   0x0804841f <+59>:    imul   -0x4(%ebp),%eax
   0x08048423 <+63>:    leave
   0x08048424 <+64>:    ret
End of assembler dump.
```



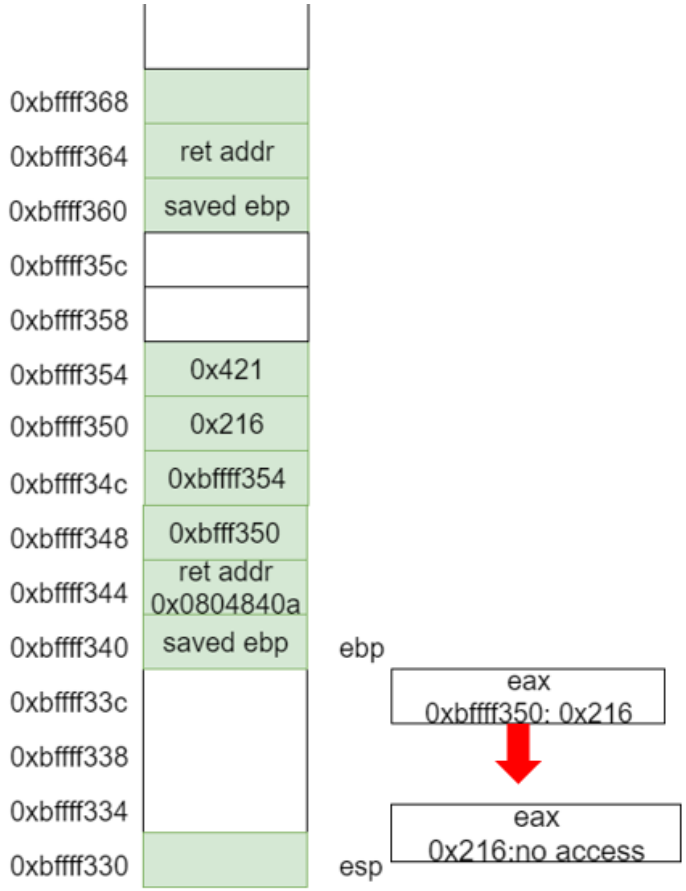
Procedures
Linux Stack Frame

```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
=> 0x080483ba <+6>:     mov    0x8(%ebp),%eax
   0x080483bd <+9>:     mov    (%eax),%eax
   0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
   0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
   0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
   0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
   0x080483e0 <+44>:    add    %edx,%eax
   0x080483e2 <+46>:    leave
   0x080483e3 <+47>:    ret
End of assembler dump.
```



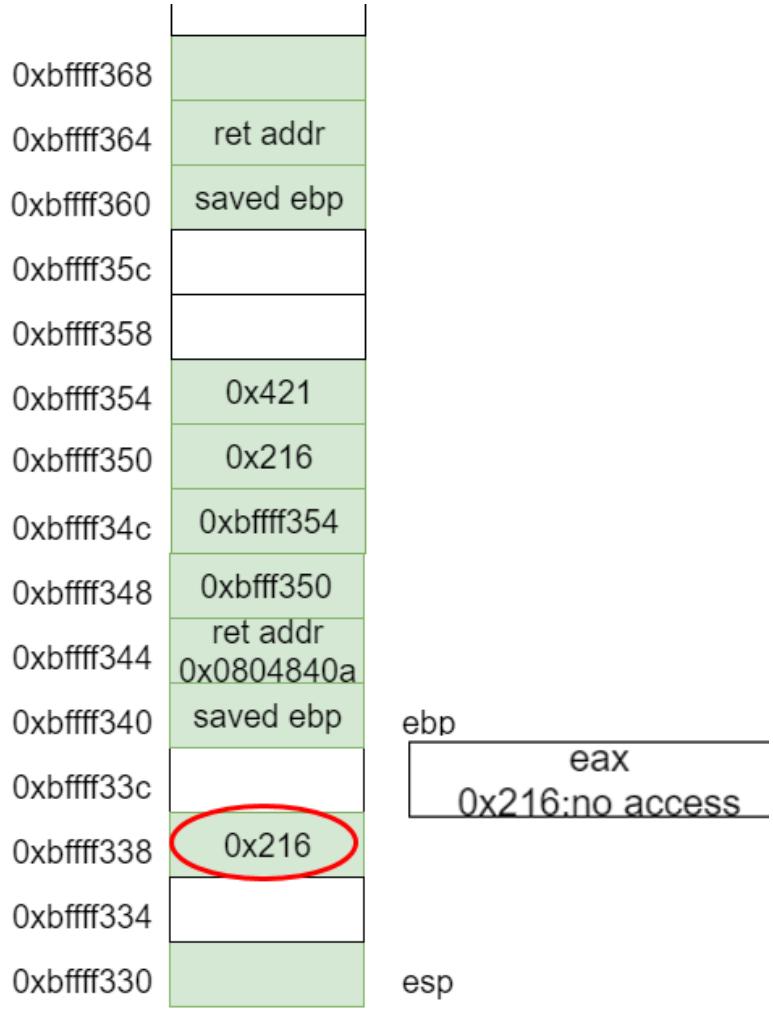
Procedures
Linux Stack Frame

```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0x8(%ebp),%eax
=> 0x080483bd <+9>:     mov    (%eax),%eax
   0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
   0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
   0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
   0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
   0x080483e0 <+44>:    add    %edx,%eax
   0x080483e2 <+46>:    leave
   0x080483e3 <+47>:    ret
End of assembler dump.
```



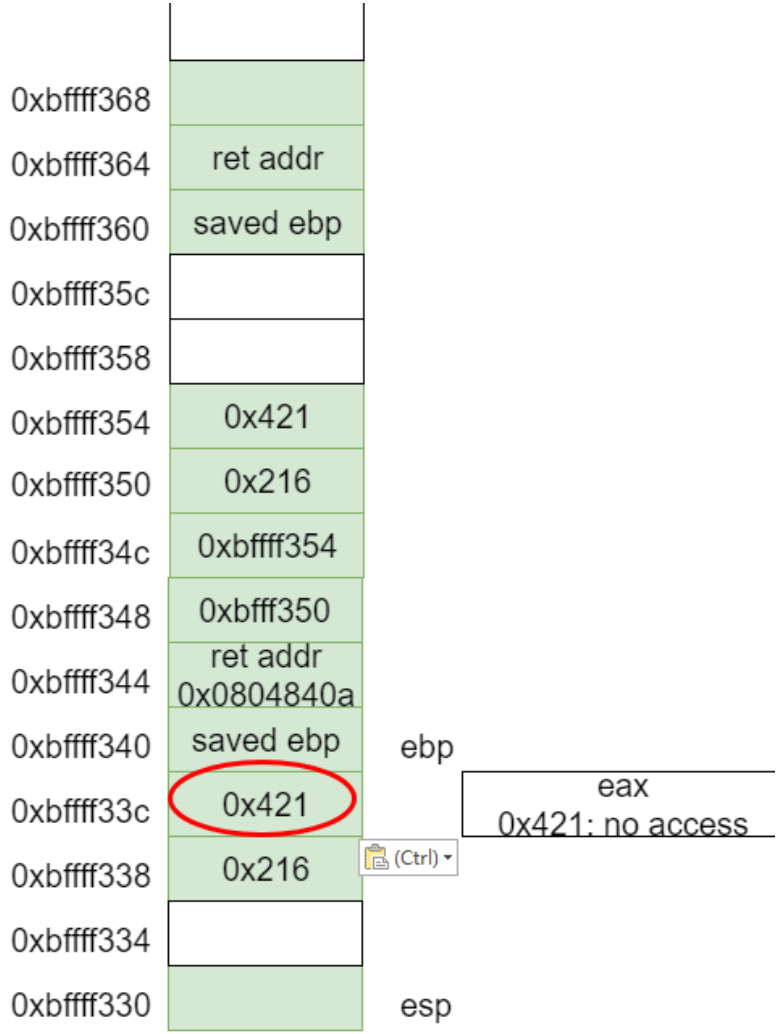
Procedures
Linux Stack Frame

```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0x8(%ebp),%eax
   0x080483bd <+9>:     mov    (%eax),%eax
=> 0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
   0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
   0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
   0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
   0x080483e0 <+44>:    add    %edx,%eax
   0x080483e2 <+46>:    leave
   0x080483e3 <+47>:    ret
End of assembler dump.
```



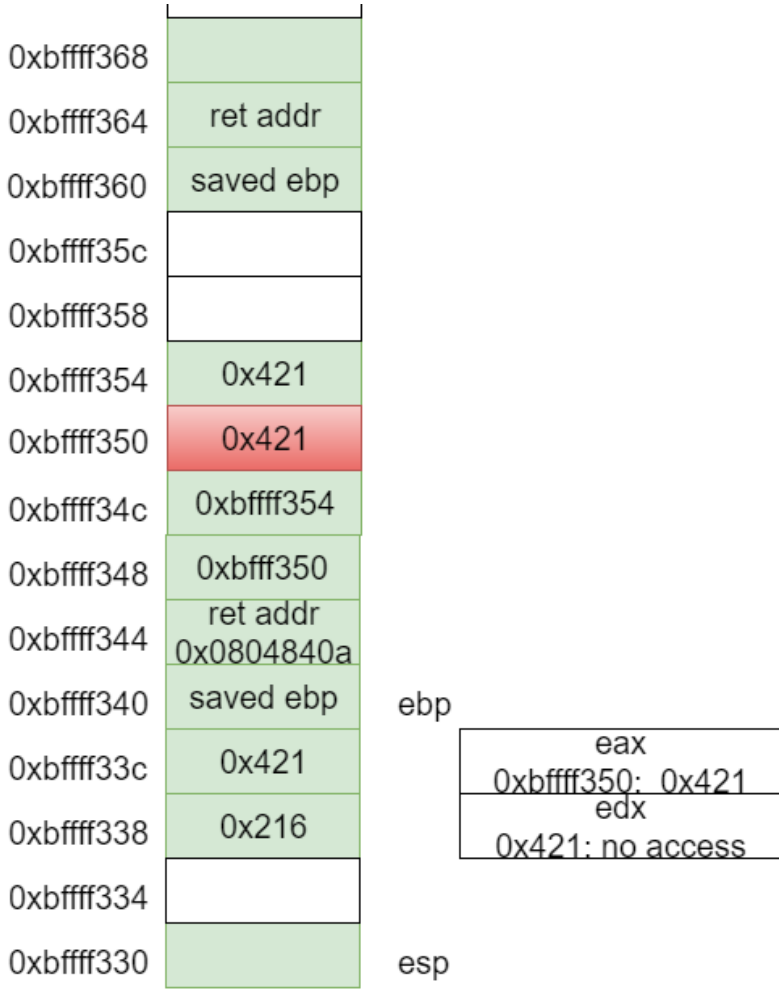
Procedures
Linux Stack Frame

```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0x8(%ebp),%eax
   0x080483bd <+9>:     mov    (%eax),%eax
   0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
=> 0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
   0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
   0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
   0x080483e0 <+44>:    add    %edx,%eax
   0x080483e2 <+46>:    leave
   0x080483e3 <+47>:    ret
End of assembler dump.
```



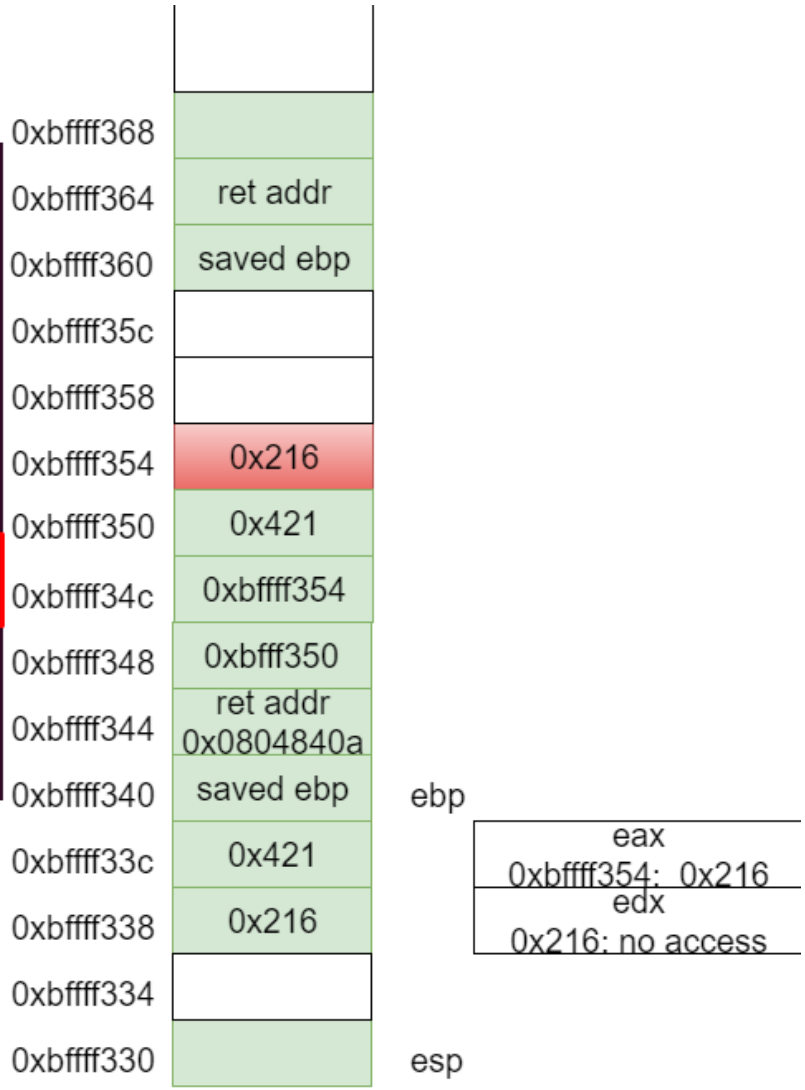
Procedures
Linux Stack Frame

```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0x8(%ebp),%eax
   0x080483bd <+9>:     mov    (%eax),%eax
   0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
   0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
=> 0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
   0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
   0x080483e0 <+44>:    add    %edx,%eax
   0x080483e2 <+46>:    leave
   0x080483e3 <+47>:    ret
End of assembler dump.
```



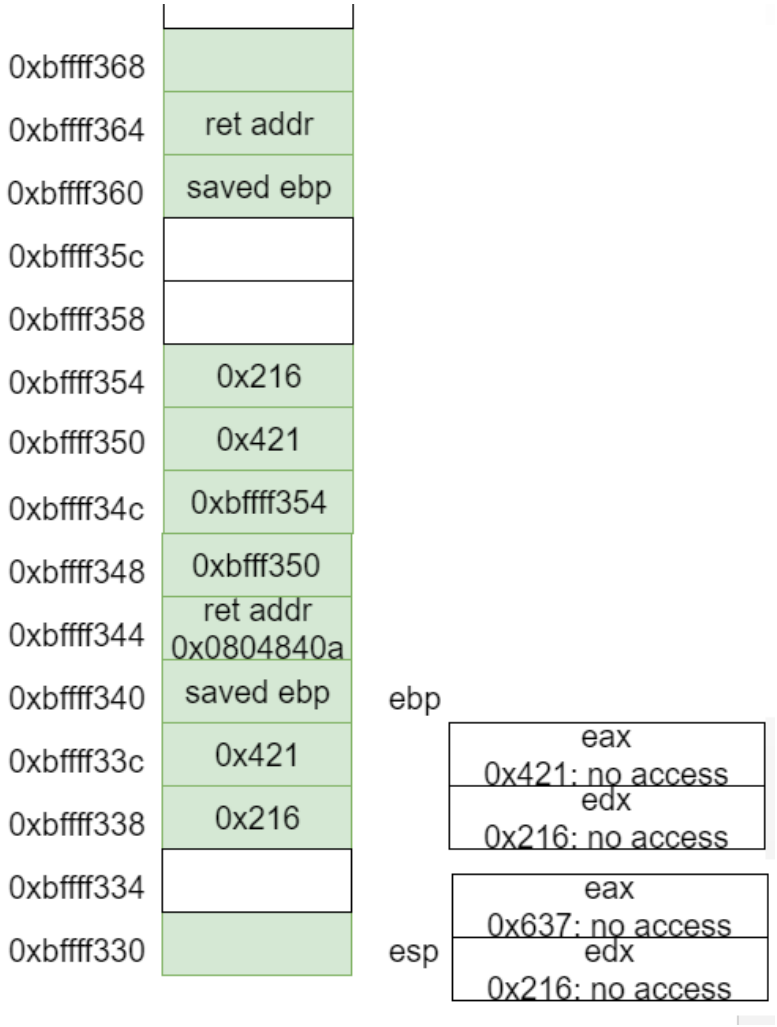
Procedures
Linux Stack Frame


```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0x8(%ebp),%eax
   0x080483bd <+9>:     mov    (%eax),%eax
   0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
   0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
   0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
=> 0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
   0x080483e0 <+44>:    add    %edx,%eax
   0x080483e2 <+46>:    leave
   0x080483e3 <+47>:    ret
End of assembler dump.
```



Procedures
Linux Stack Frame

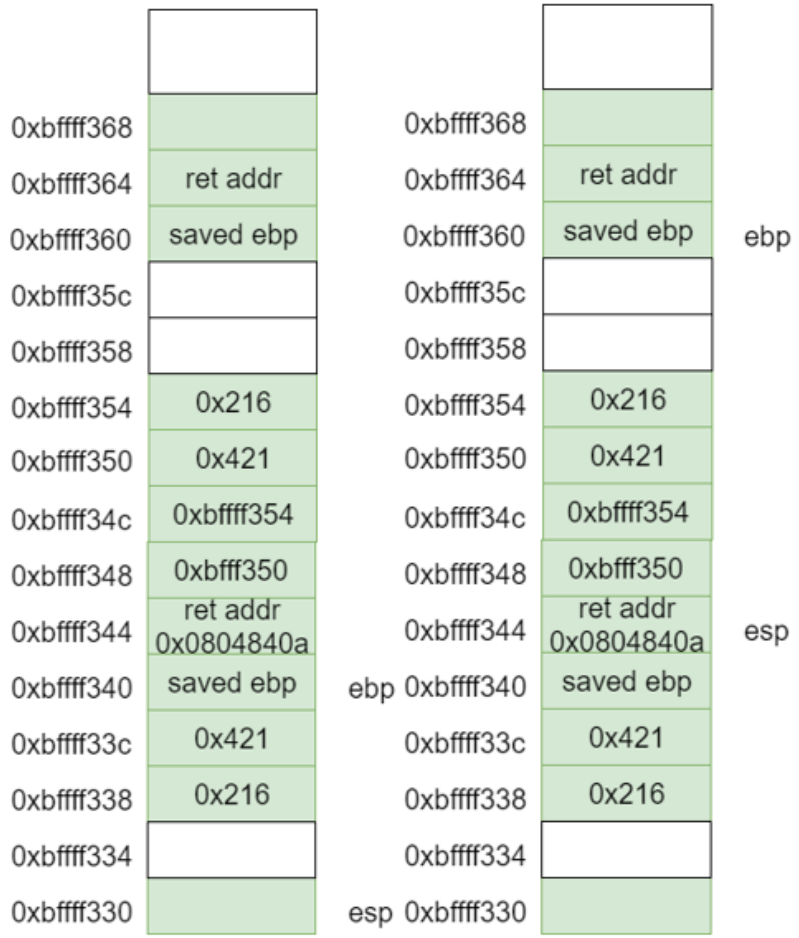
```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0x8(%ebp),%eax
   0x080483bd <+9>:     mov    (%eax),%eax
   0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
   0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
   0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
   0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
=> 0x080483e0 <+44>:    add    %edx,%eax
   0x080483e2 <+46>:    leave
   0x080483e3 <+47>:    ret
End of assembler dump.
```



Procedures
Linux Stack Frame


```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0x8(%ebp),%eax
   0x080483bd <+9>:     mov    (%eax),%eax
   0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
   0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
   0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
   0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
   0x080483e0 <+44>:    add    %edx,%eax
=> 0x080483e2 <+46>:    leave
   0x080483e3 <+47>:    ret
End of assembler dump.
```

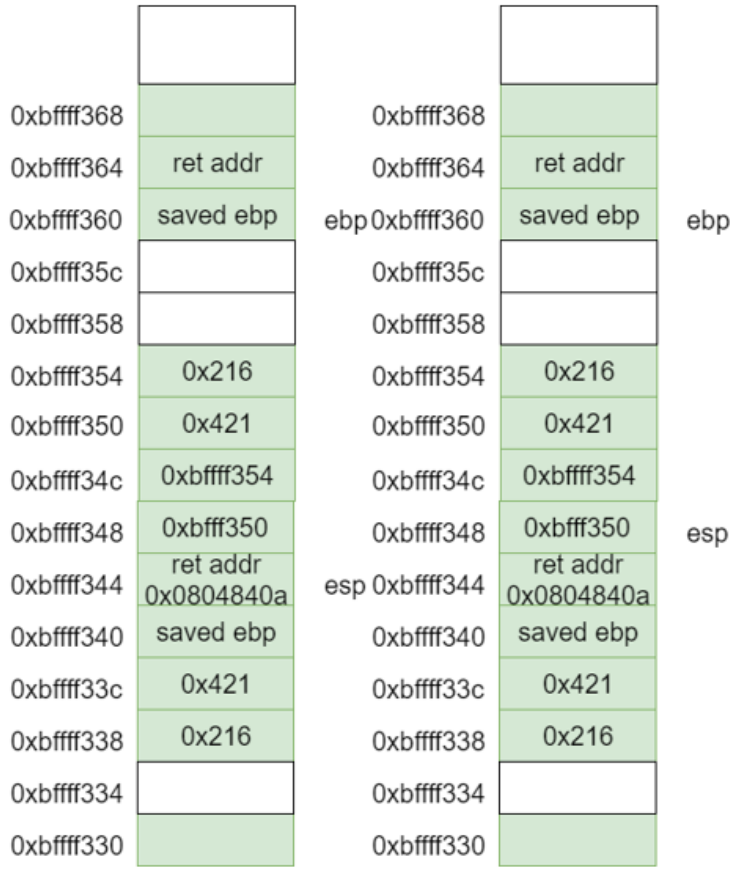
leave means
mov %ebp， %esp
pop %ebp



Procedures
Linux Stack Frame

```
Dump of assembler code for function swap_add:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0x8(%ebp),%eax
   0x080483bd <+9>:     mov    (%eax),%eax
   0x080483bf <+11>:    mov    %eax,-0x8(%ebp)
   0x080483c2 <+14>:    mov    0xc(%ebp),%eax
   0x080483c5 <+17>:    mov    (%eax),%eax
   0x080483c7 <+19>:    mov    %eax,-0x4(%ebp)
   0x080483ca <+22>:    mov    0x8(%ebp),%eax
   0x080483cd <+25>:    mov    -0x4(%ebp),%edx
   0x080483d0 <+28>:    mov    %edx,(%eax)
   0x080483d2 <+30>:    mov    0xc(%ebp),%eax
   0x080483d5 <+33>:    mov    -0x8(%ebp),%edx
   0x080483d8 <+36>:    mov    %edx,(%eax)
   0x080483da <+38>:    mov    -0x4(%ebp),%eax
   0x080483dd <+41>:    mov    -0x8(%ebp),%edx
   0x080483e0 <+44>:    add    %edx,%eax
   0x080483e2 <+46>:    leave
=> 0x080483e3 <+47>:    ret
End of assembler dump.
```

```
Dump of assembler code for function caller:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     movl   $0x216,-0x10(%ebp)
   0x080483f1 <+13>:    movl   $0x421,-0xc(%ebp)
   0x080483f8 <+20>:    lea    -0xc(%ebp),%eax
   0x080483fb <+23>:    mov    %eax,0x4(%esp)
   0x080483ff <+27>:    lea    -0x10(%ebp),%eax
   0x08048402 <+30>:    mov    %eax,(%esp)
   0x08048405 <+33>:    call   0x80483b4 <swap_add>
=> 0x0804840a <+38>:    mov    %eax,-0x8(%ebp)
   0x0804840d <+41>:    mov    -0x10(%ebp),%edx
   0x08048410 <+44>:    mov    -0xc(%ebp),%eax
   0x08048413 <+47>:    mov    %edx,%ecx
   0x08048415 <+49>:    sub    %eax,%ecx
   0x08048417 <+51>:    mov    %ecx,%eax
   0x08048419 <+53>:    mov    %eax,-0x4(%ebp)
   0x0804841c <+56>:    mov    -0x8(%ebp),%eax
   0x0804841f <+59>:    imul   -0x4(%ebp),%eax
   0x08048423 <+63>:    leave
   0x08048424 <+64>:    ret
End of assembler dump.
```



ret
pop the %esp to %eip
jump to the target

Procedures
Linux Stack Frame

CALL，RET和LEAVE

CALL指令的步骤：首先是将返回地址（也就是call指令要执行时EIP的值）压入栈顶，然后是将程序跳转到当前调用的方法的起始地址。执行push和jump指令。

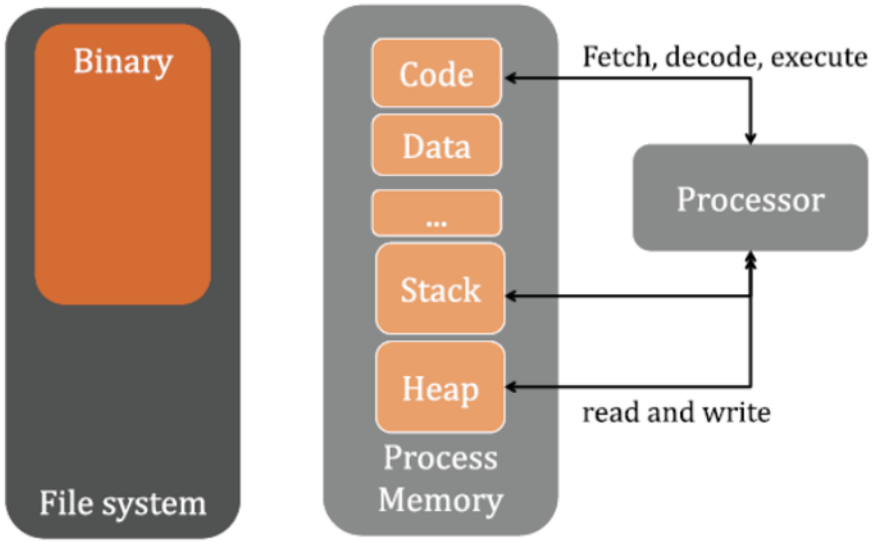
RET指令则是将栈顶的返回地址弹出到EIP，然后按照EIP此时指示的指令地址继续执行程序。

LEAVE指令是将栈指针指向帧指针，然后POP备份的原帧指针到%EBP。

Procedures
Linux Stack Frame


```
push %ebp:
    sub $4, %esp
    mov %ebp, (%esp)

pop %ebp:
    mov (%esp), %ebp
    add $4, %esp

leave:
    mov %ebp, %esp
    pop %ebp

call:
    Push the ret on the stack
    Jump to the target address(the start address of function called)

ret:
    pop the ret(stored on stack) to %eip
    back to the address
```

Procedures
Control Flow Hijacking



Using an exploit (a special input) to hijack the original program control flow

computation + ***control***