计算机系统基础

程序的机器级表示（5）

王晶

jwang@ruc.edu.cn，信息楼124

2024年11月

过程机制

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

过程机制

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

过程机制

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

过程机制

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Today

- Procedures
  - Mechanisms
  - Stack Structure
  - **Calling Conventions**
    - **Passing control**
    - Passing data
    - Managing local data
  - Illustration of Recursion

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **`call label`**
  - Push return address on stack
  - Jump to ***label*** **`(direct)`**
  - **`*operand (indirect)`**

```
push retaddr
jmp callee
```

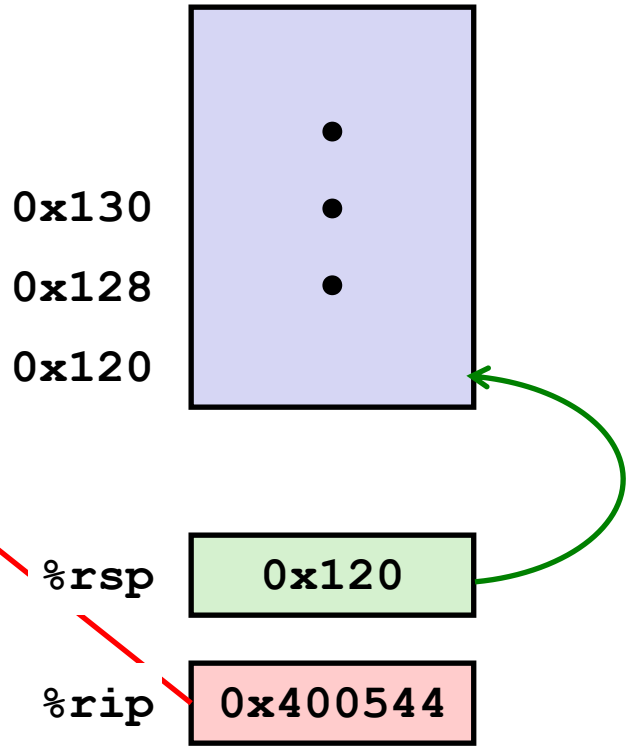
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly
- Procedure return: **`ret`**
  - Pop address from stack
  - Jump to address

```
pop retaddr
jmp retaddr
```

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

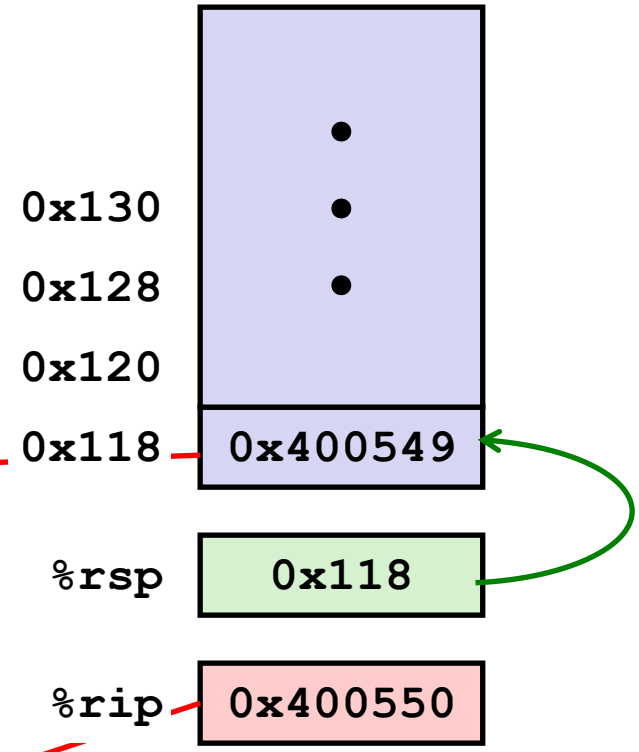
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

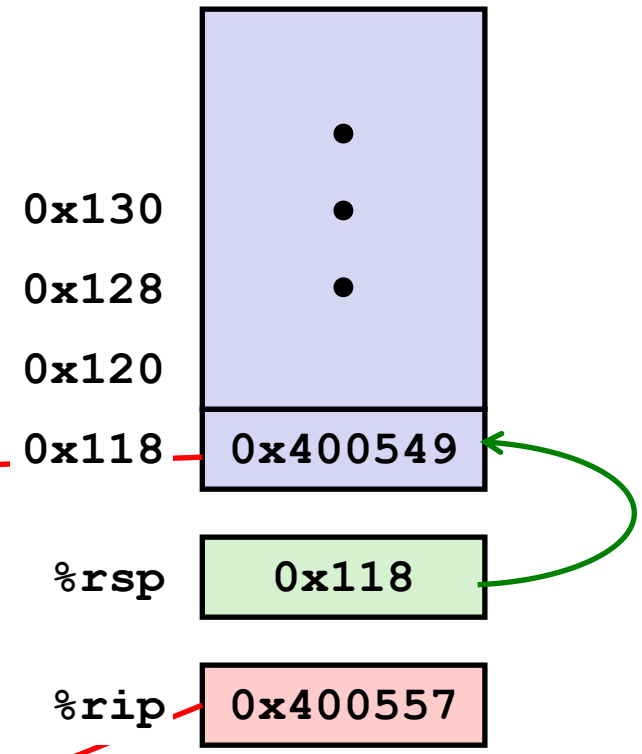
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

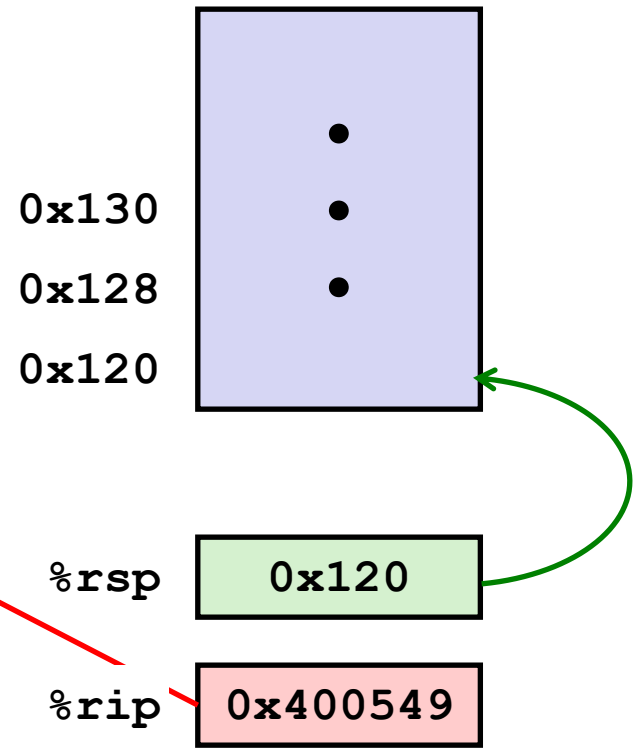
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



课堂练习

- 下面的代码片段经常出现在库函数的编译版本中：

```
        call next
next:
        popl %eax
```

A)eax被设置成什么值？

B)解释为什么这个调用没有与之匹配的ret指令

C)这段代码完成了什么功能？

Today

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - **Managing local data**
    - Passing data
  - Illustrations of Recursion & Pointers

Call Chain Example

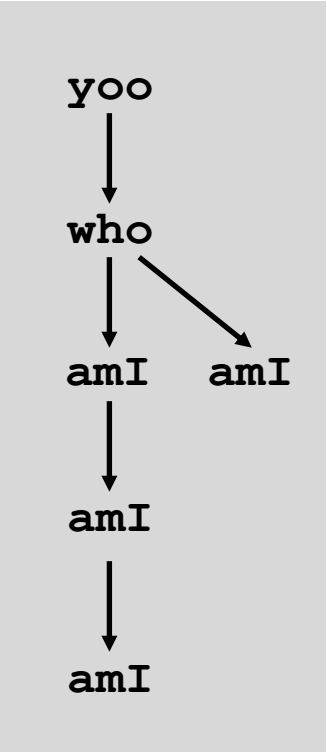
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

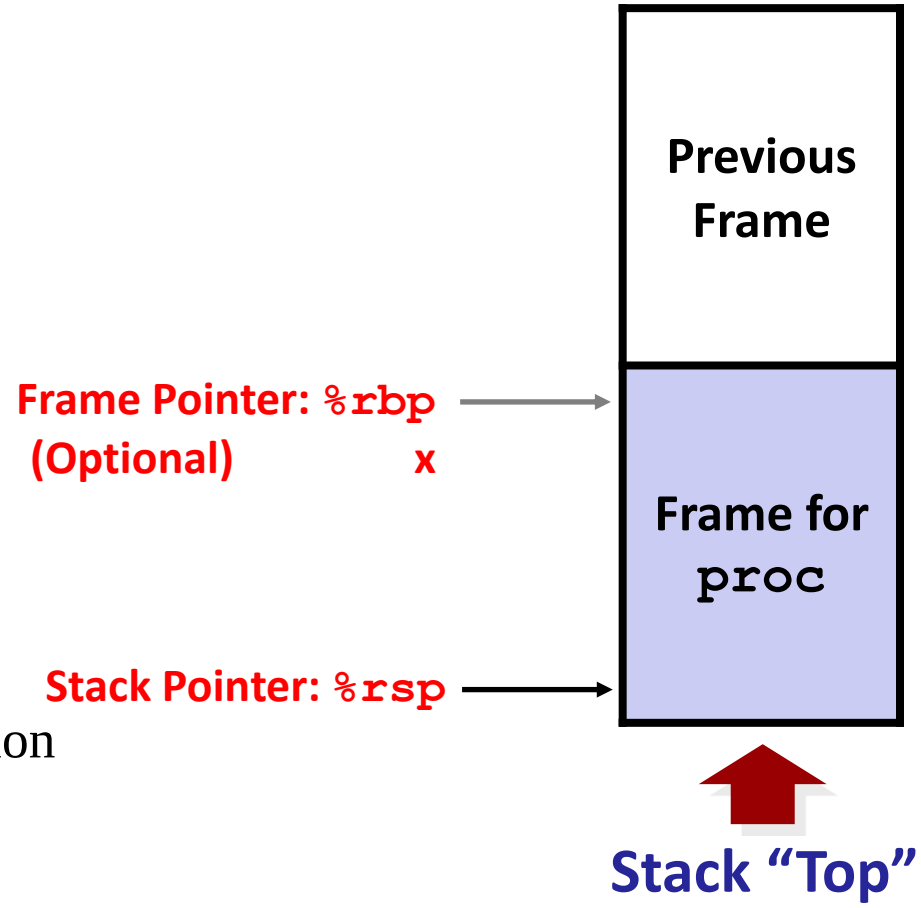
Procedure `amI()` is recursive

Example Call Chain



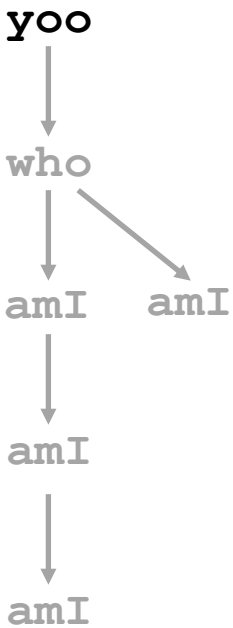
Stack Frames

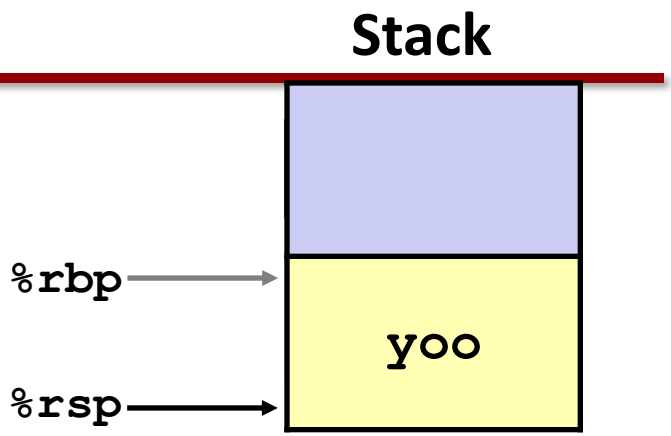
- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction



Example

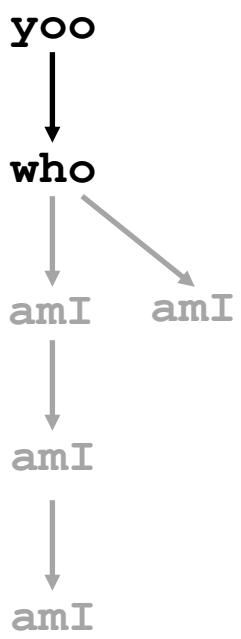
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

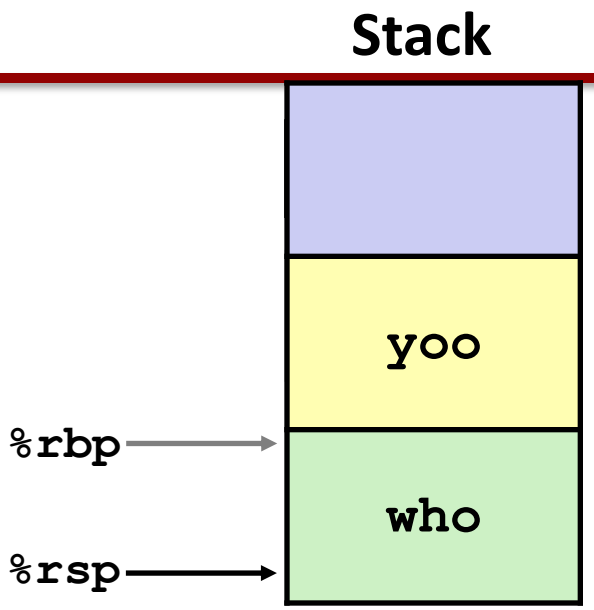




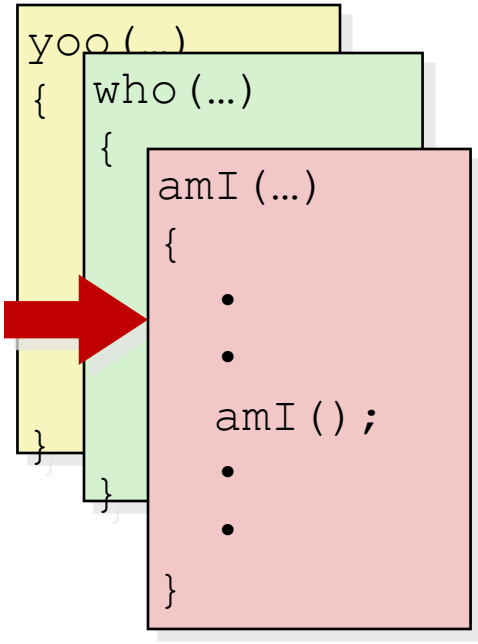
Example

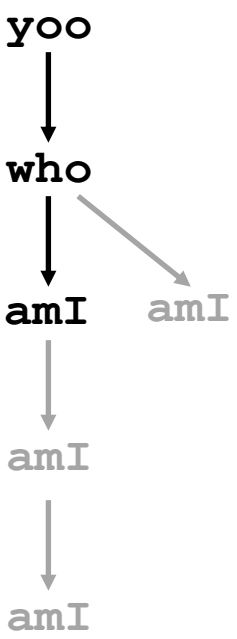
```
yoo(...)
{
    who(...)
    {
        • • •
        amI();
        • • •
        amI();
        • • •
    }
}
```

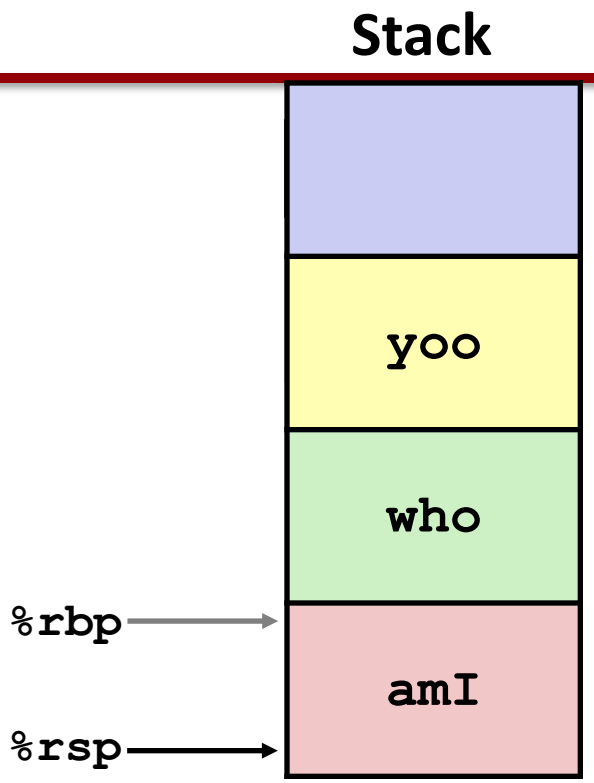




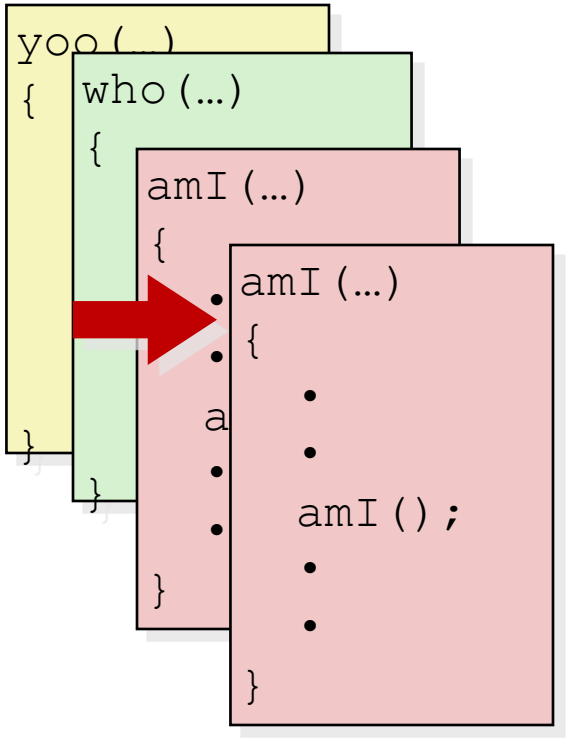
Example

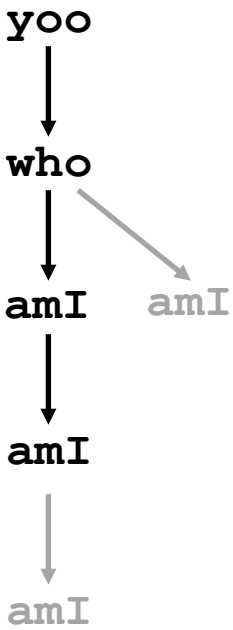

yoo(…)
{
}
who(…)
{
}
amI(…)
{
amI();
}

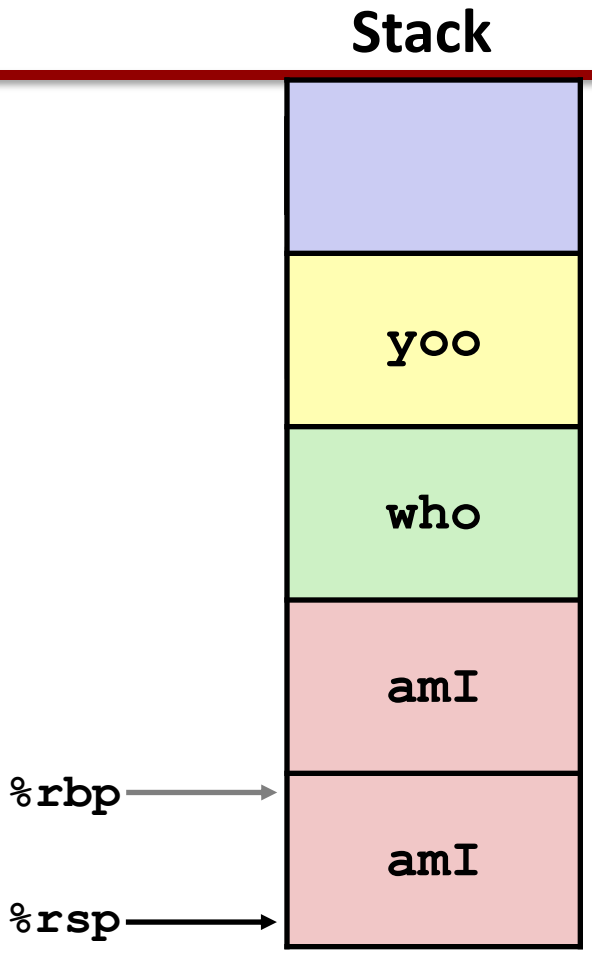

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

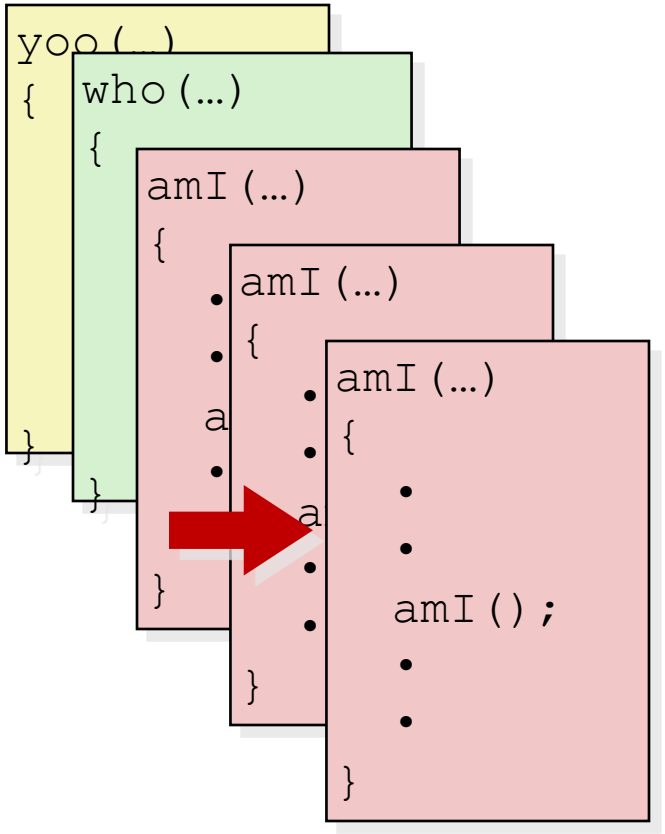
Example

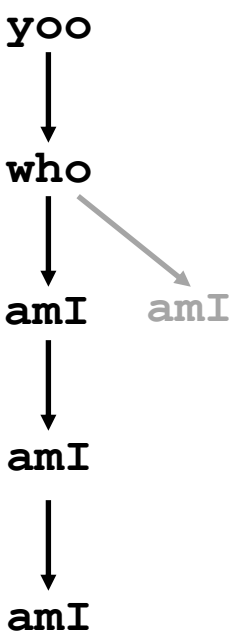

yoo(…)
{
}
who(…)
{
}
amI(…)
{
}
amI(…)
{
amI();
}

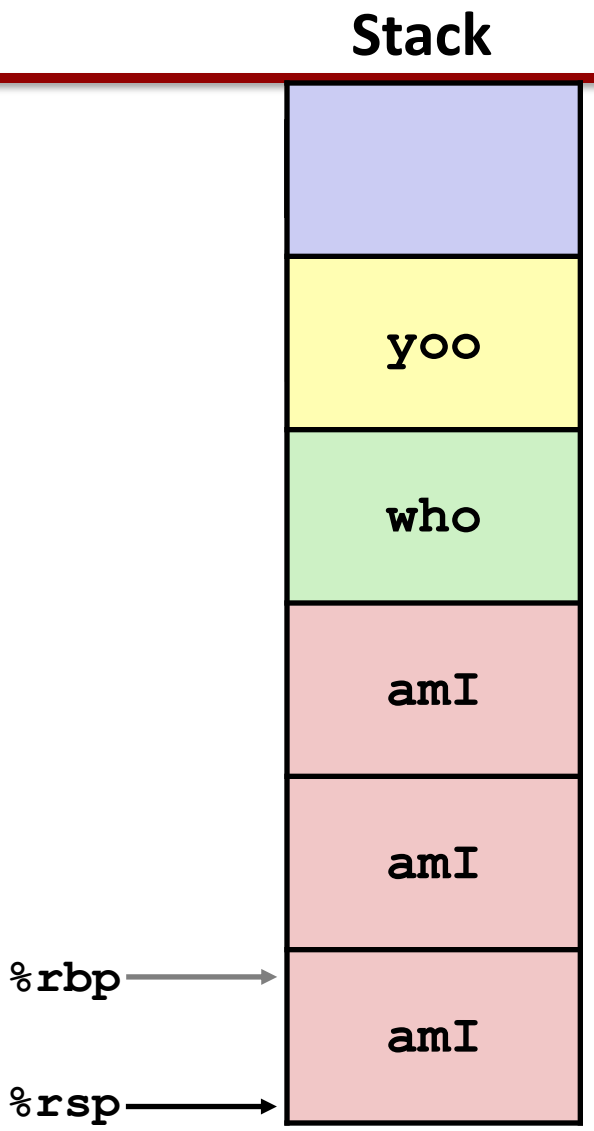

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

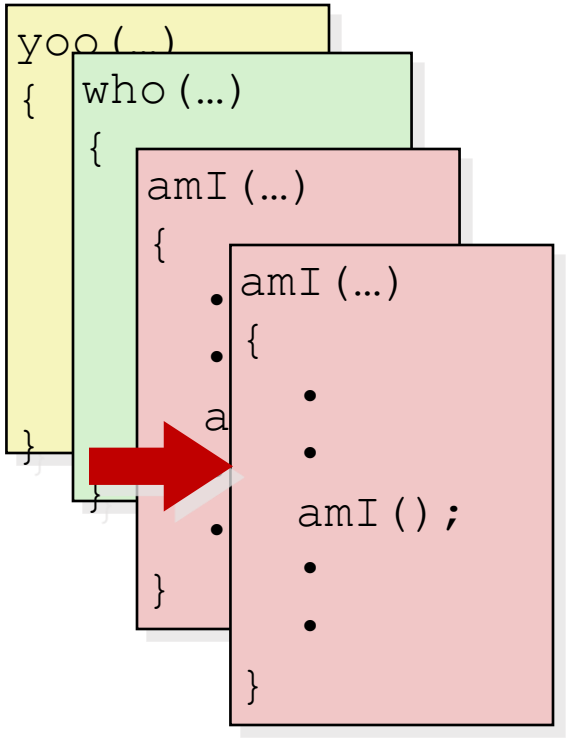
Example

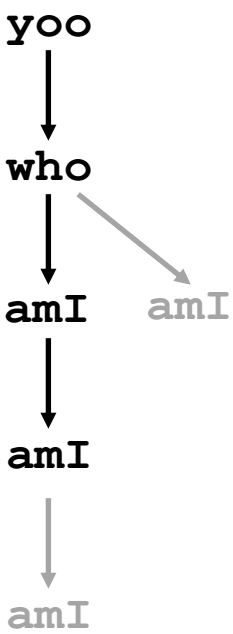

yoo(...)
{
}
who(...)
{
}
amI(...)
{
}
amI(...)
{
}
amI(...)
{
amI();
}

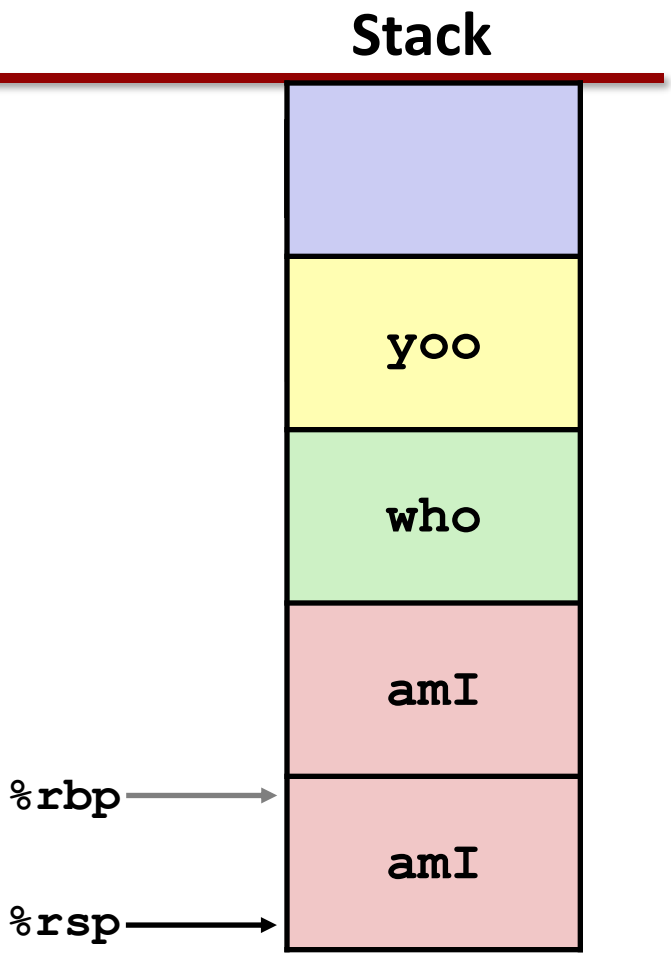

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

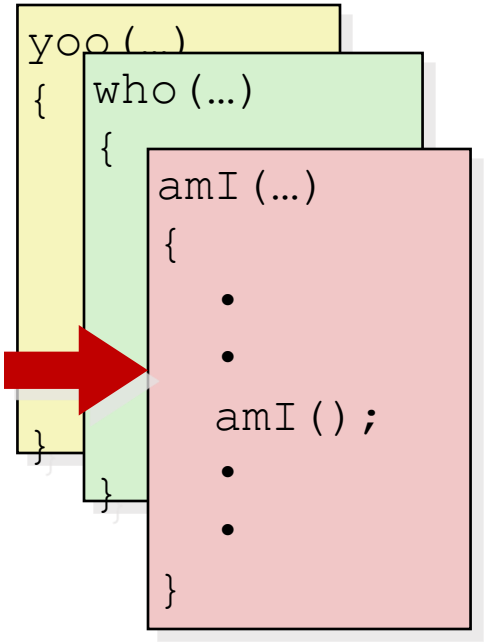
Example

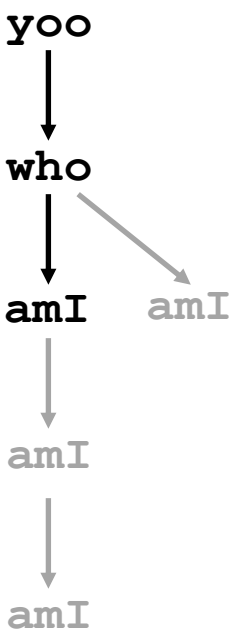

yoo(…)
{
}
who(…)
{
}
amI(…)
{
}
amI(…)
{
amI();
}

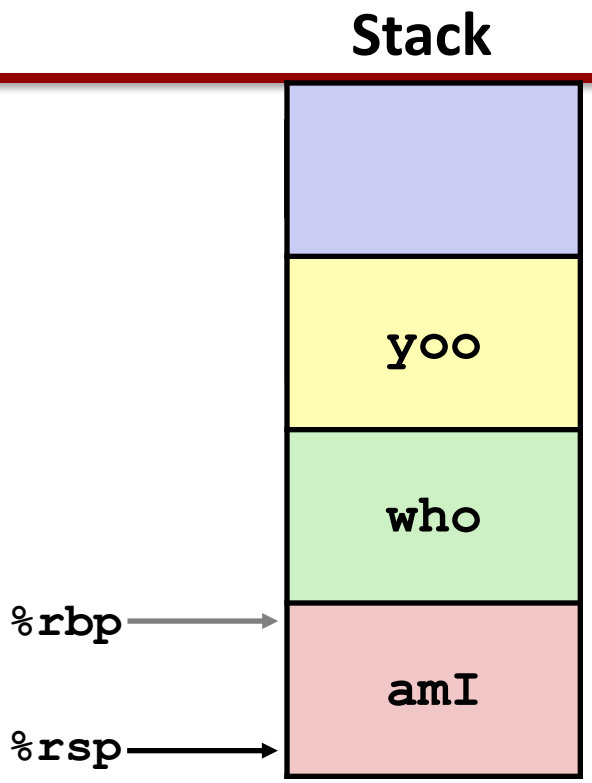

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

Example

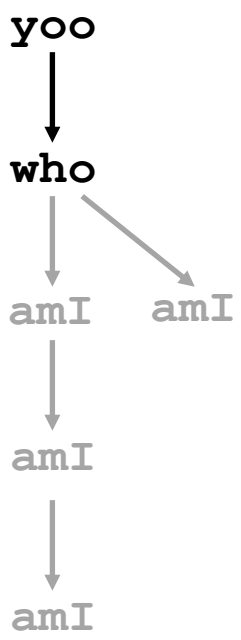

yoo(…)
{
}
who(…)
{
}
amI(…)
{
amI();
}

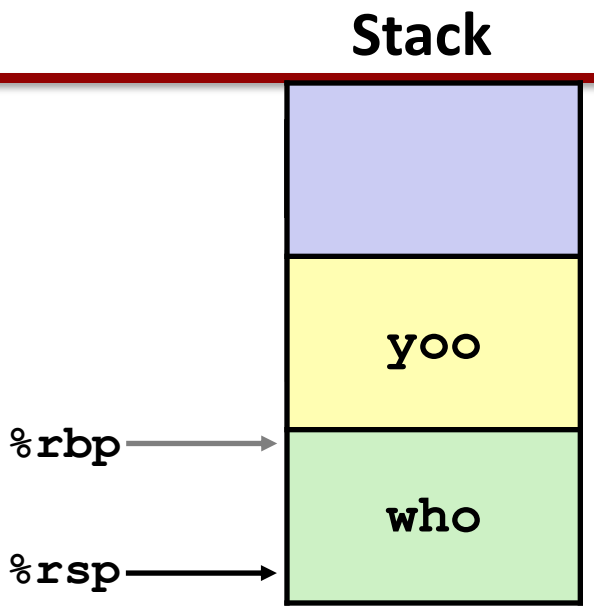

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

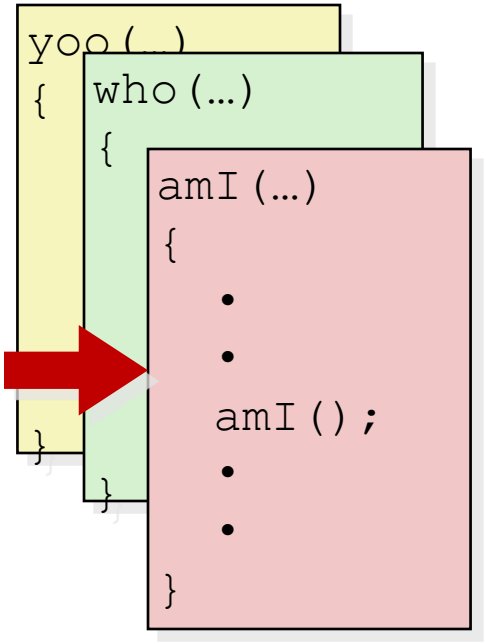
Example

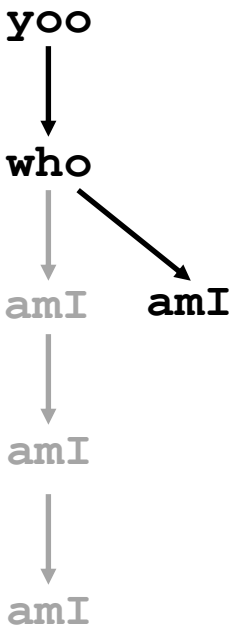
```
yoo(...)
{
    who(...)
    {
        • • •
        amI();
        • • •
        amI();
        • • •
    }
}
```

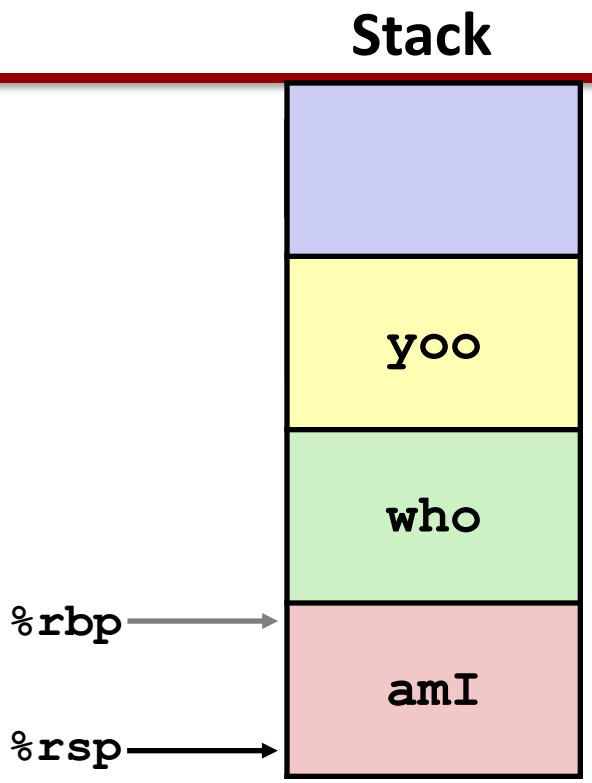




Example

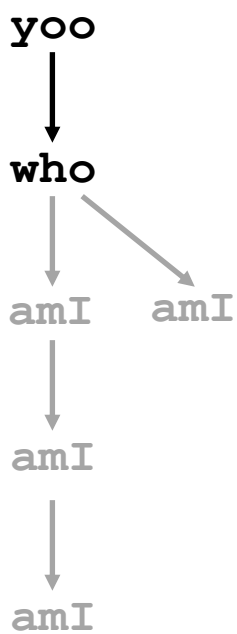

yoo(…)
{
}
who(…)
{
}
amI(…)
{
amI();
}

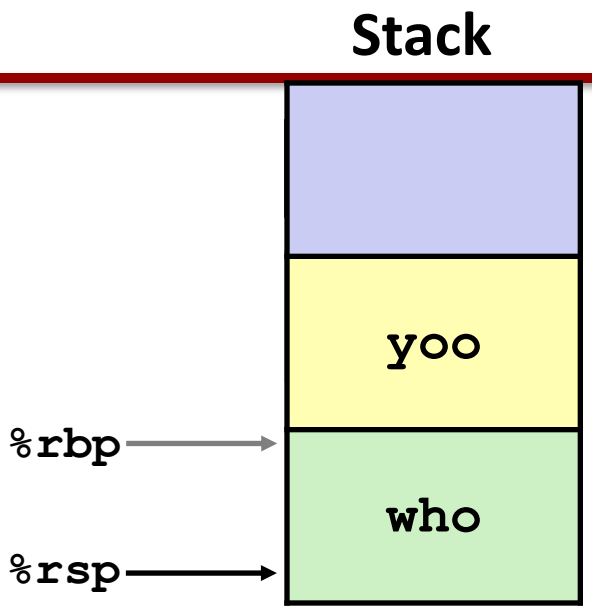

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

Example

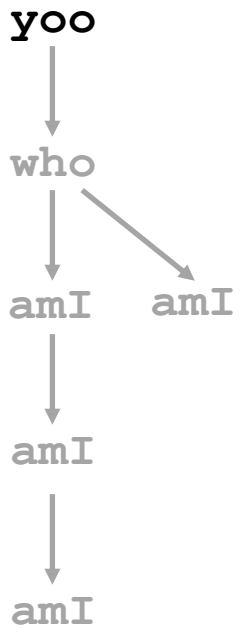
```
yoo(...)
{
    who(...)
    {
        • • •
        amI();
        • • •
        amI();
        • • •
    }
}
```

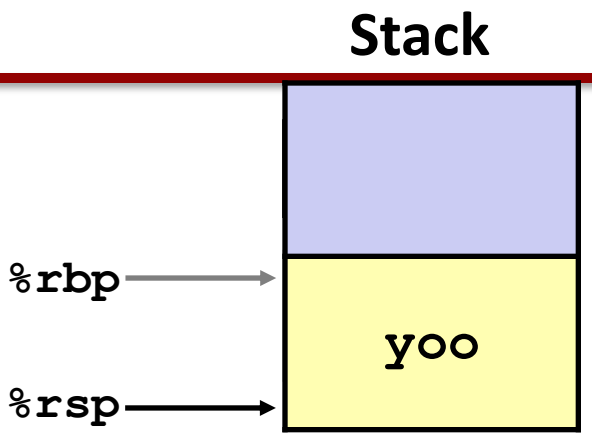




Example

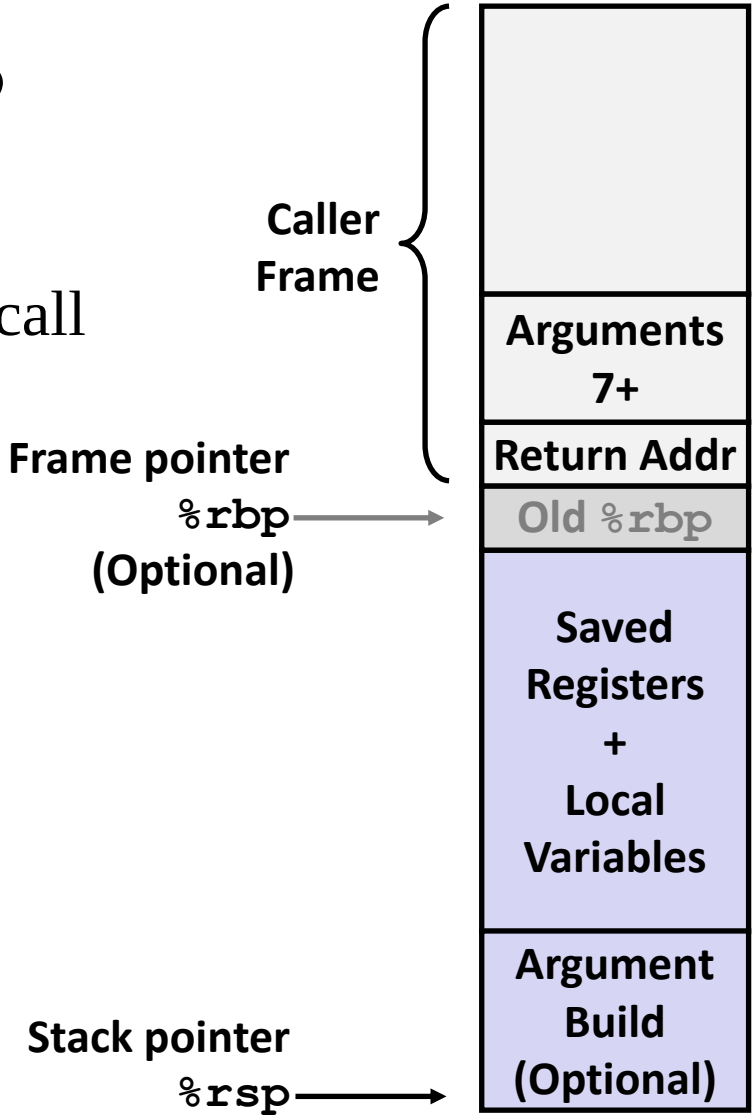
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```





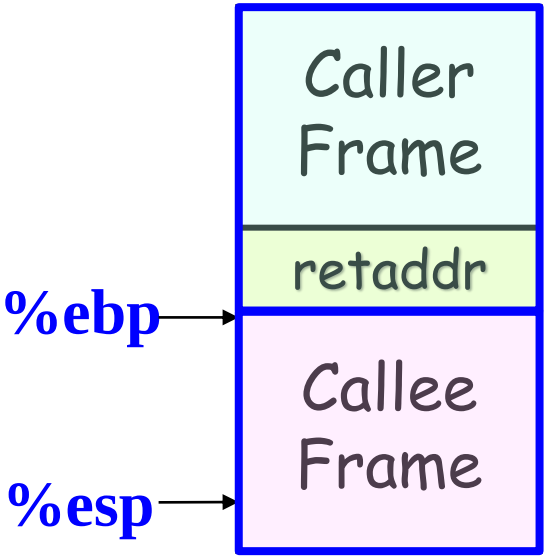
x86-64/Linux Stack Frame

- 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



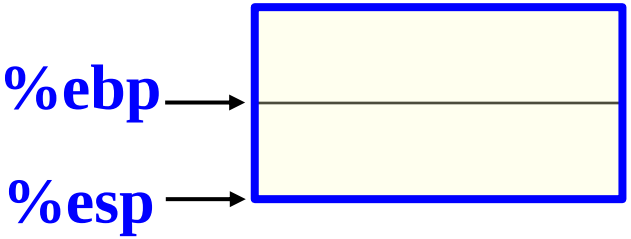
Stack Frame Structure

- 主函数、子函数共享同一个system stack
- call: save *return address* in the stack
- ret: pop *return address* from stack
  - The end of caller's stack frame



Frame Chain

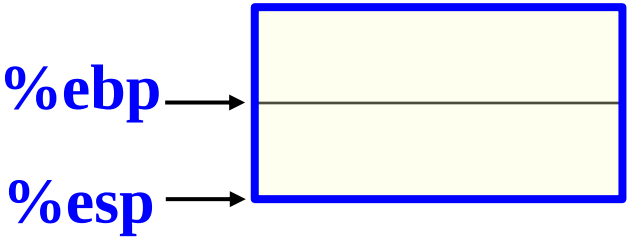
- Pointers (%ebp/%esp) only delimit topmost frame



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee



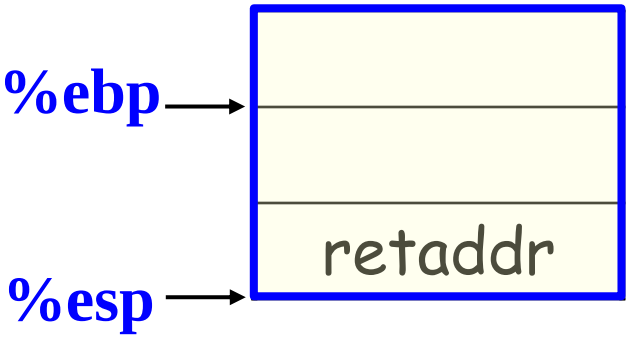
Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

2. push %ebp
3. mov %esp, %ebp



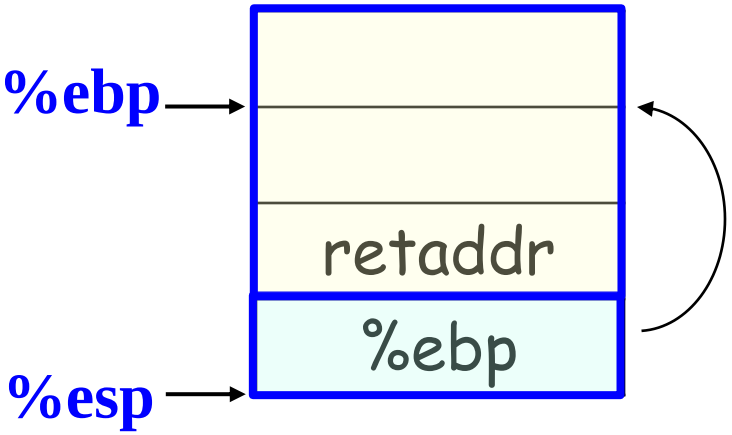
Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

2. push %ebp
3. mov %esp, %ebp



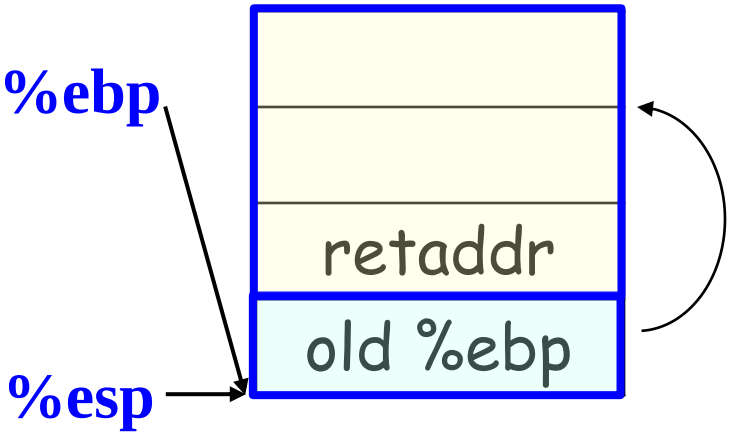
Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

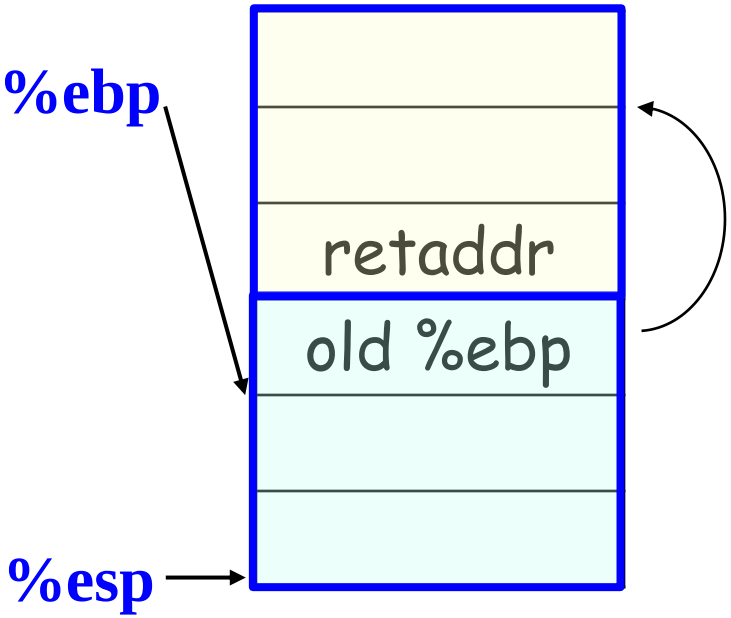
2. push %ebp
3. mov %esp, %ebp



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

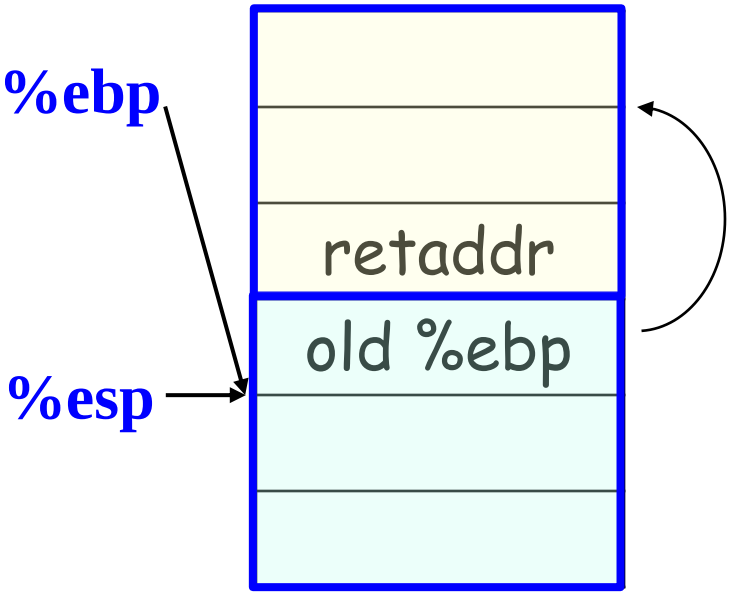
```
   1.  call callee
callee:
   2.  push %ebp
   3.  mov %esp, %ebp
       . . .
n-2.  mov %ebp, %esp
n-1.  pop %ebp
   n.  ret
```



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

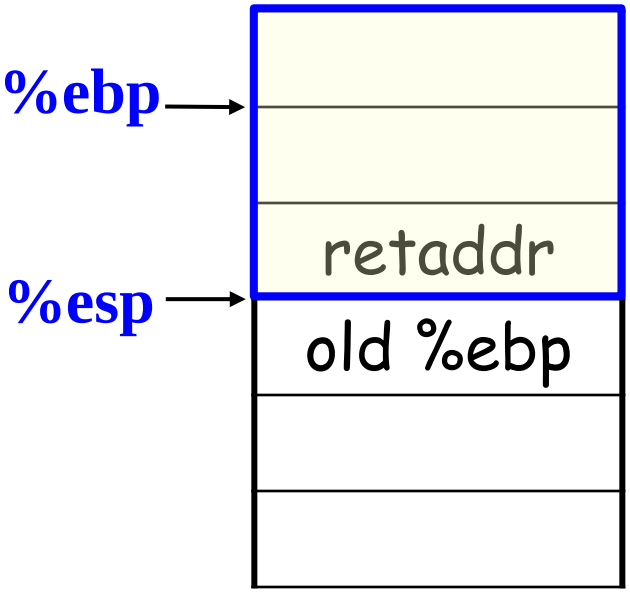
```
   1.  call callee
callee:
   2.  push %ebp
   3.  mov %esp, %ebp
       . . .
n-2.   mov %ebp, %esp
n-1.   pop %ebp
   n.  ret
```



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

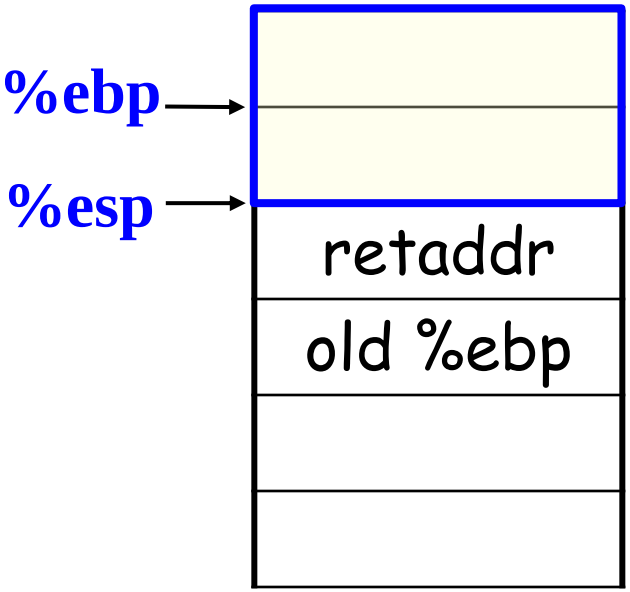
```
   1.  call callee
callee:
   2.  push %ebp
   3.  mov %esp, %ebp
       . . .
n-2.  mov %ebp, %esp
n-1.  pop %ebp
   n.  ret
```



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

```
   1.  call callee
callee:
   2.  push %ebp
   3.  mov %esp, %ebp
       . . .
n-2.   mov %ebp, %esp
n-1.   pop %ebp
   n.  ret
```



Restore Caller %ebp

- Instruction
  - `leave`
- Behavior description (by hardware)
  - Adjust %esp to callee %ebp
  - Pop *caller %ebp* from stack

leave = mov + pop

```
mov %ebp, %esp
pop %ebp
```

Execution of call and ret

```
//beginning of function sum
1. 08048394 <sum>:
2.   8048394:  55                push  %ebp
     . . .
3.   80483a4:                    ret
     . . .
//call to sum from main
4.   80483dc:  e8 b3 ff ff ff    call  8048394<sum>
5.   80483e1:  83 c4 14          add   $0x14, %esp
```

Executing call	
%eip	0x080483dc
%esp	0xff9bc960
ret	-

After call	
%eip	0x08048394
%esp	0xff9bc95c
ret	0x080483e1

After ret	
%eip	0x080483e1
%esp	0xff9bc960
ret	-

Today

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Managing local data
    - **Passing data**
  - Illustrations of Recursion & Pointers

Procedure Data Flow

Registers

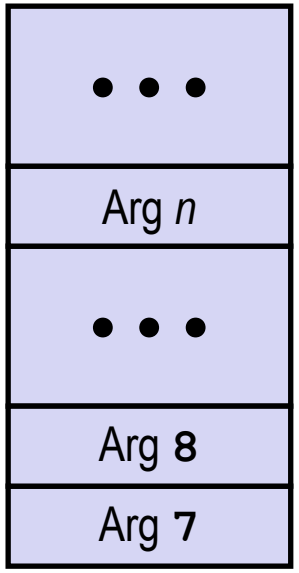
- First 6 arguments

`%rdi`
`%rsi`
`%rdx`
`%rcx`
`%r8`
`%r9`

- Return value

`%rax`

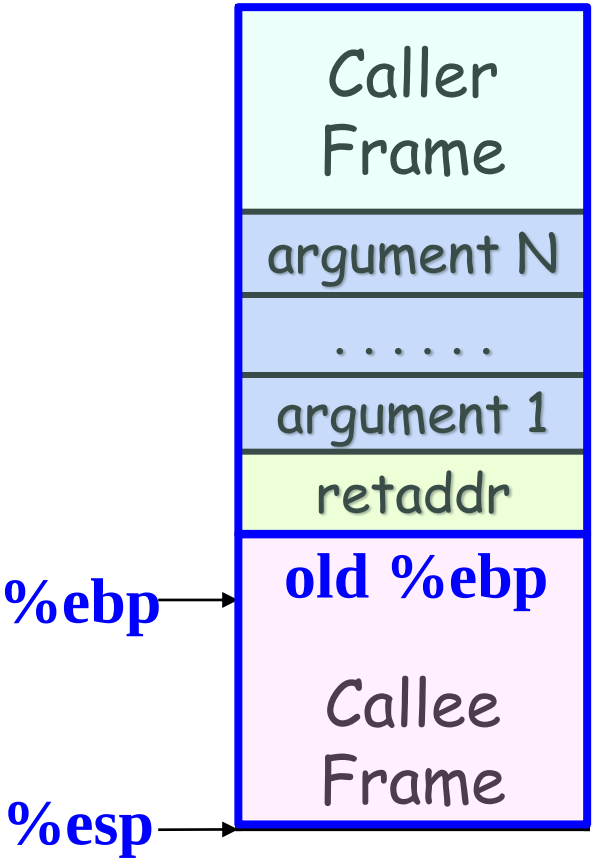
Stack



- Only allocate stack space when needed

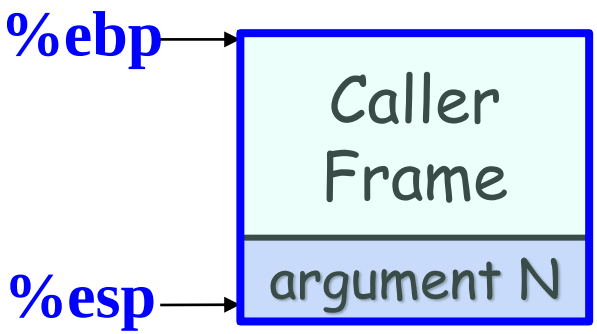
Passing Data: Arguments

- Pushed by Caller
  - Saved in caller frame
  - Just upon of return address
  - From Nth to 1st (from right to left)
- Used by Callee
  - Relative to %ebp
  - Offset: 4+ 4*i + %ebp



Passing Data: Arguments

push argument N

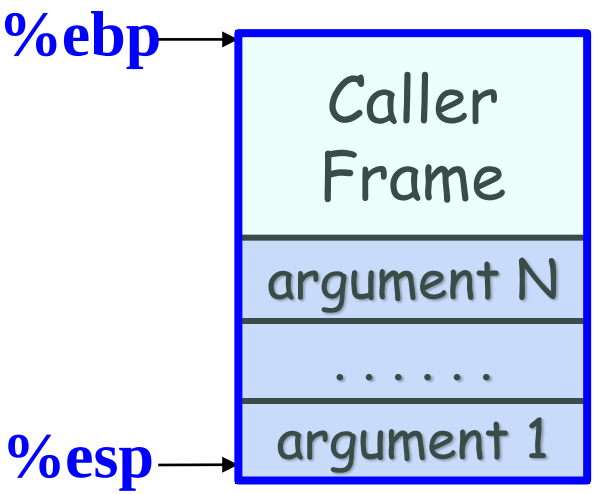


Passing Data: Arguments

push argument N

. . .

push argument 1



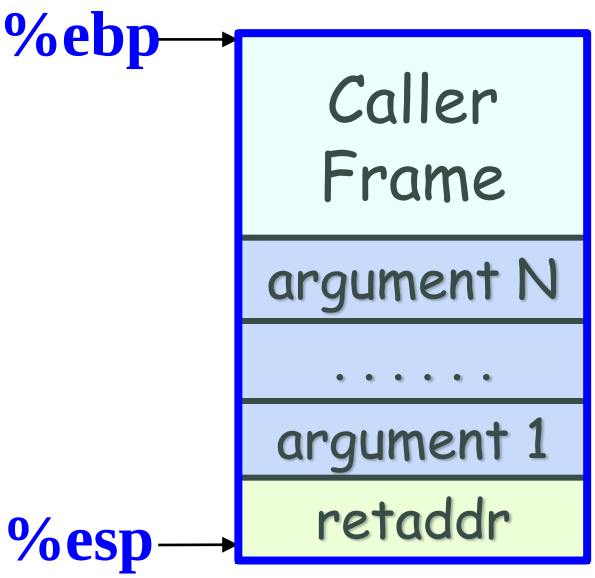
Passing Data: Arguments

push argument N

. . .

push argument 1

call callee



Passing Data: Arguments

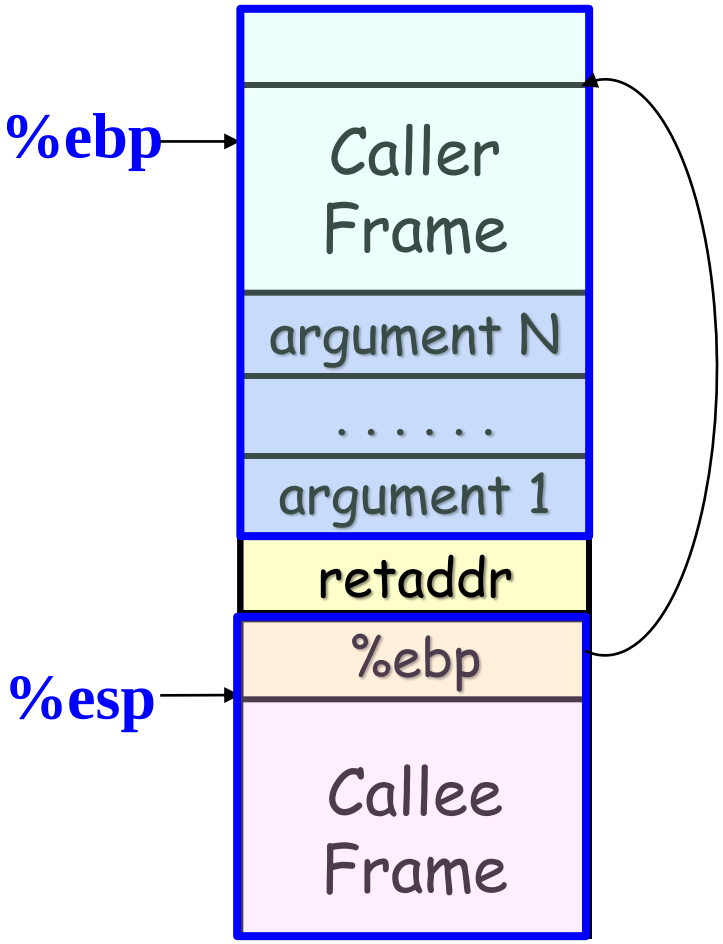
push argument N

. . .

push argument 1

call callee

push %ebp



Passing Data: Arguments

push argument N

. . .

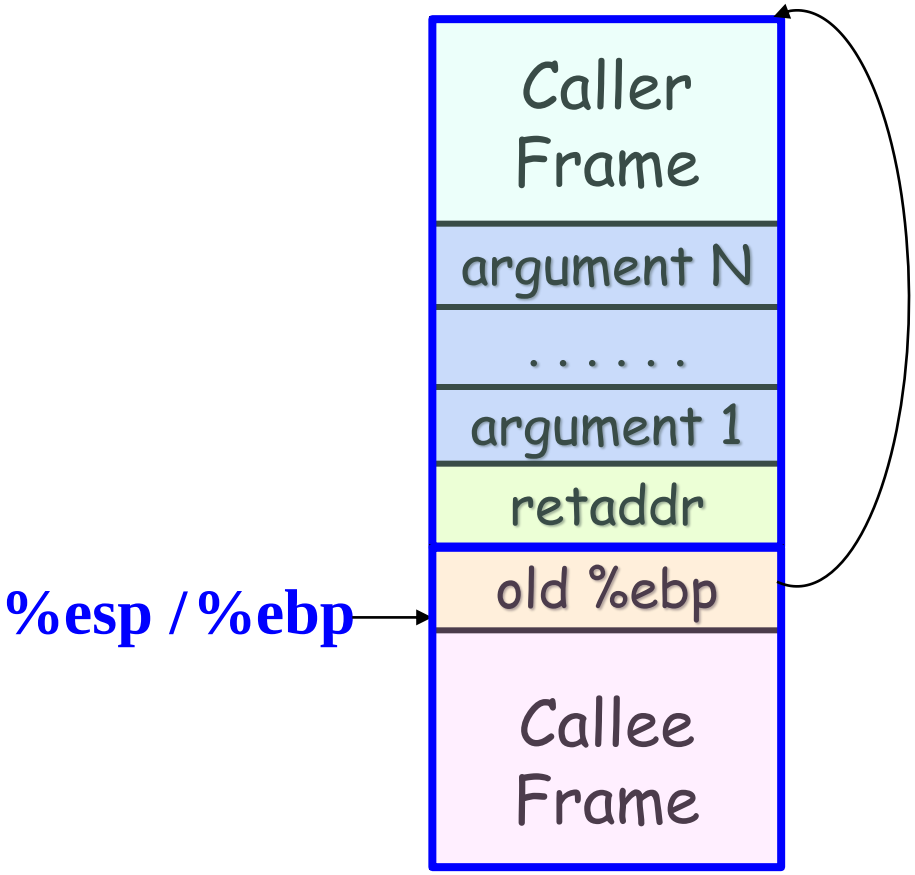
push argument 1

call callee

push %ebp

mov %esp, %ebp

. . .



Passing Data: Arguments

- X86 64位，传参数先用寄存器
  - %rdi，%rsi，%rdx，%rcx，%r8，%r9 用作函数参数，依次对应第1参数，第2参数....第6个参数（从左到右）
  - 剩余参数还和以前的算法一样，从右往左依次入栈
  - 例如func(p1, p2, p3, p4, p5, p6, p7, p8)
    - p1: %rdi, p2: %rsi, p3: %rdx, p4: %rcx, p5: %r8, p6: %r9
    - 栈：(低地址）←ret addr, p7, p8←（高地址）

Passing Data: Return Value

- Specific register to keep the return value
  - %eax / %rax is used to pass the result of callee to caller

Register Saving Conventions

- When procedure **`yoo`** calls **`who`**:
  - **`yoo`** is the ***caller***
  - **`who`** is the ***callee***
- Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

  - This could be trouble → something should be done!
    - Need some coordination

Calling Convention

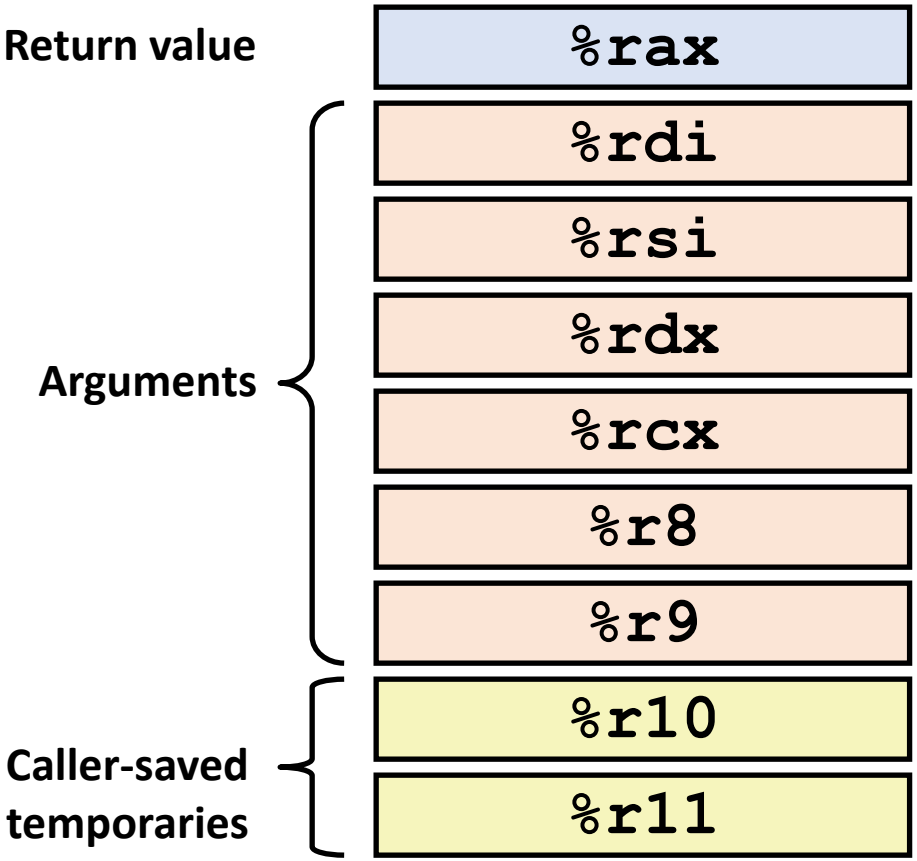
- Registers act as a single resource shared by all of the procedures
  - Only 1 procedure can be active
  - Partition registers between caller and callee (必须遵守惯例)
    - Caller-save register
    - Callee-save register
  - Only consider the registers used by the procedure

Register Saving Conventions

- When procedure **`yoo`** calls **`who`**:
  - **`yoo`** is the ***caller***
  - **`who`** is the ***callee***
- Can register be used for temporary storage?
- Conventions
  - ***"Caller Saved"***
    - Caller saves temporary values in its frame before the call
  - ***"Callee Saved"***
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

x86-64 Linux Register Usage #1

- **%rax**
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- **%rdi, ..., %r9**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- **%r10, %r11**
  - Caller-saved
  - Can be modified by procedure



- **Caller must restore them if it tries to use them after calling**

Caller-save Registers

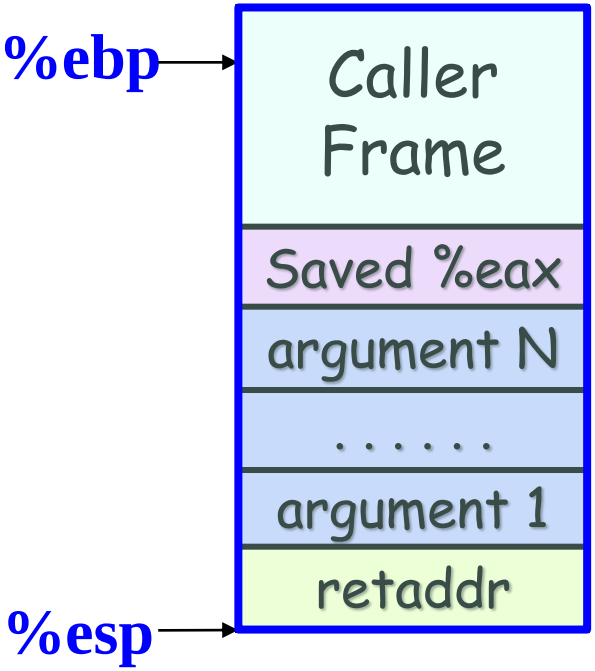
push %eax

push argument N

. . .

push argument 1

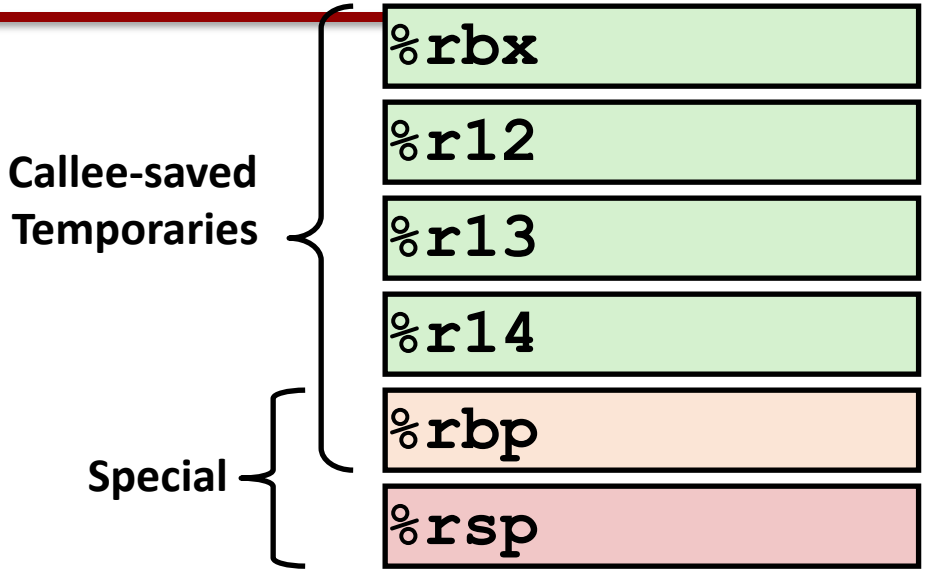
call callee



Calling Convention

- Callee-save registers
  - %rbx, %rbp, %r12~%r14
  - Saved by callee
  - Caller can use these registers freely

  - Callee must save them before using
  - Callee must restore them before return



Callee-save Registers

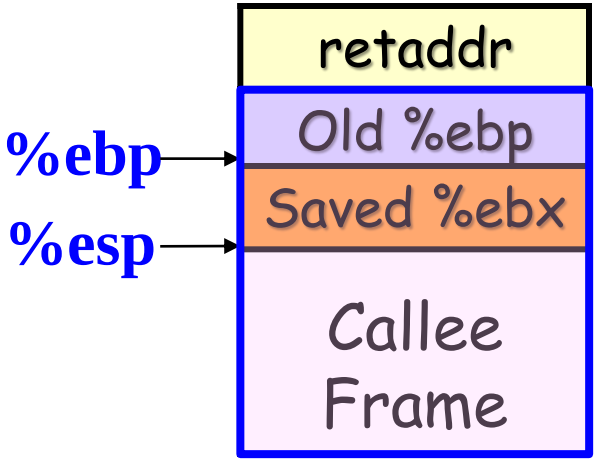
call callee

push %ebp

mov %esp, %ebp

push %ebx

. . .

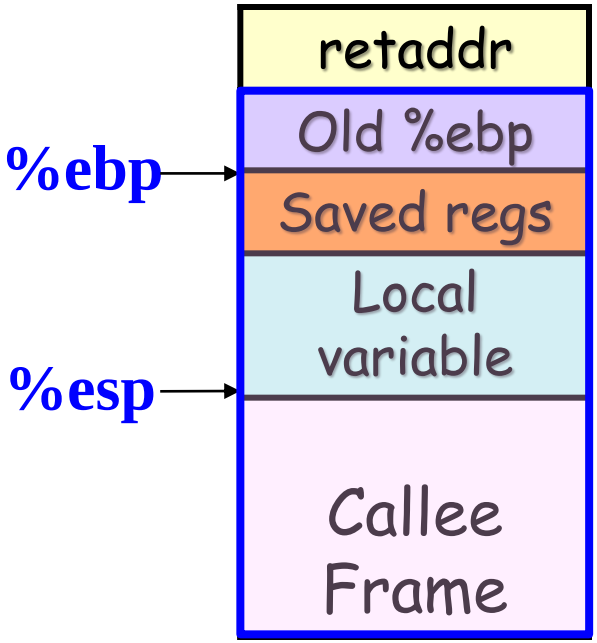


Local Variable

- Why not store local variables in registers ?
  - No enough registers
  - Array and structures (e.g., a[2])
  - Need address (e.g., &a)

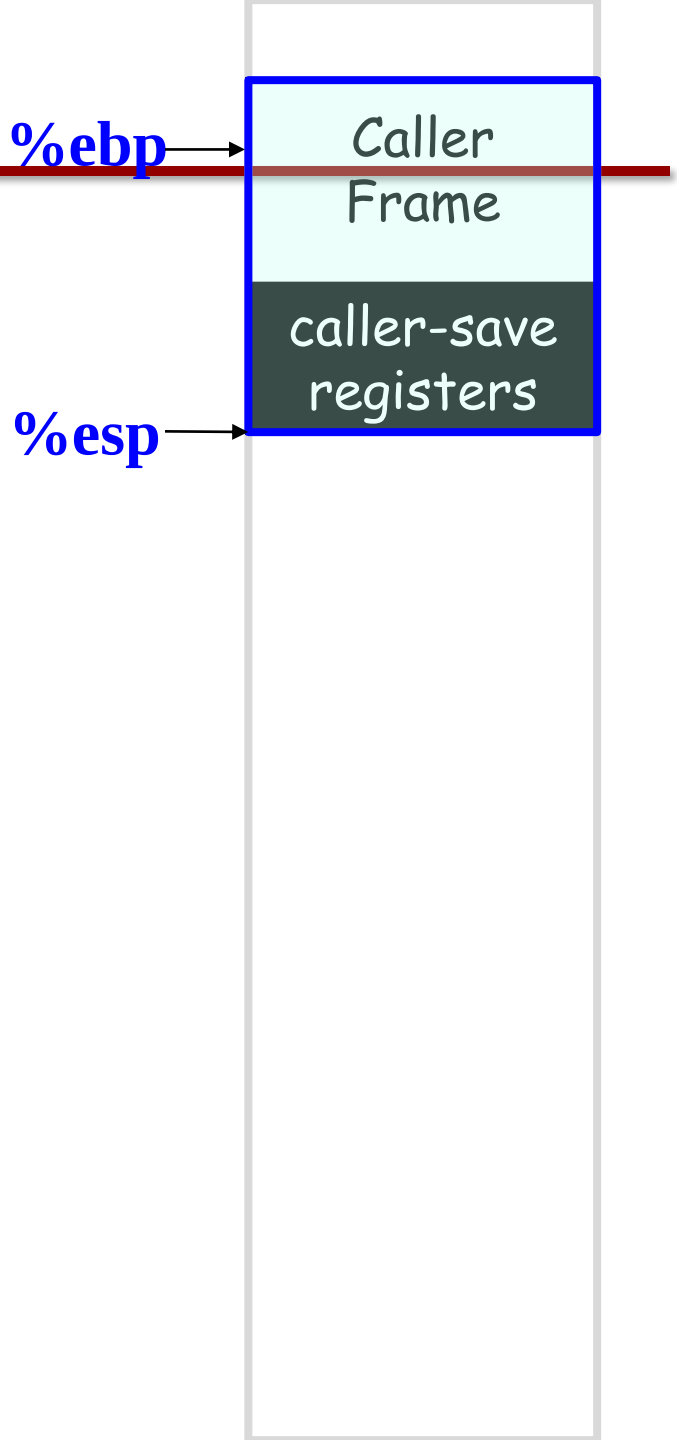
Local Variable

- Allocation
  - Below saved regs or old %ebp
  - move/sub %esp, (e.g., subl $4, %esp)
- De-allocation
  - move/add %esp, (e.g., addl $4, %esp)
- Usage
  - Relative to %esp/%ebp, (e.g., movl %eax, 8(%esp))



Put it Together

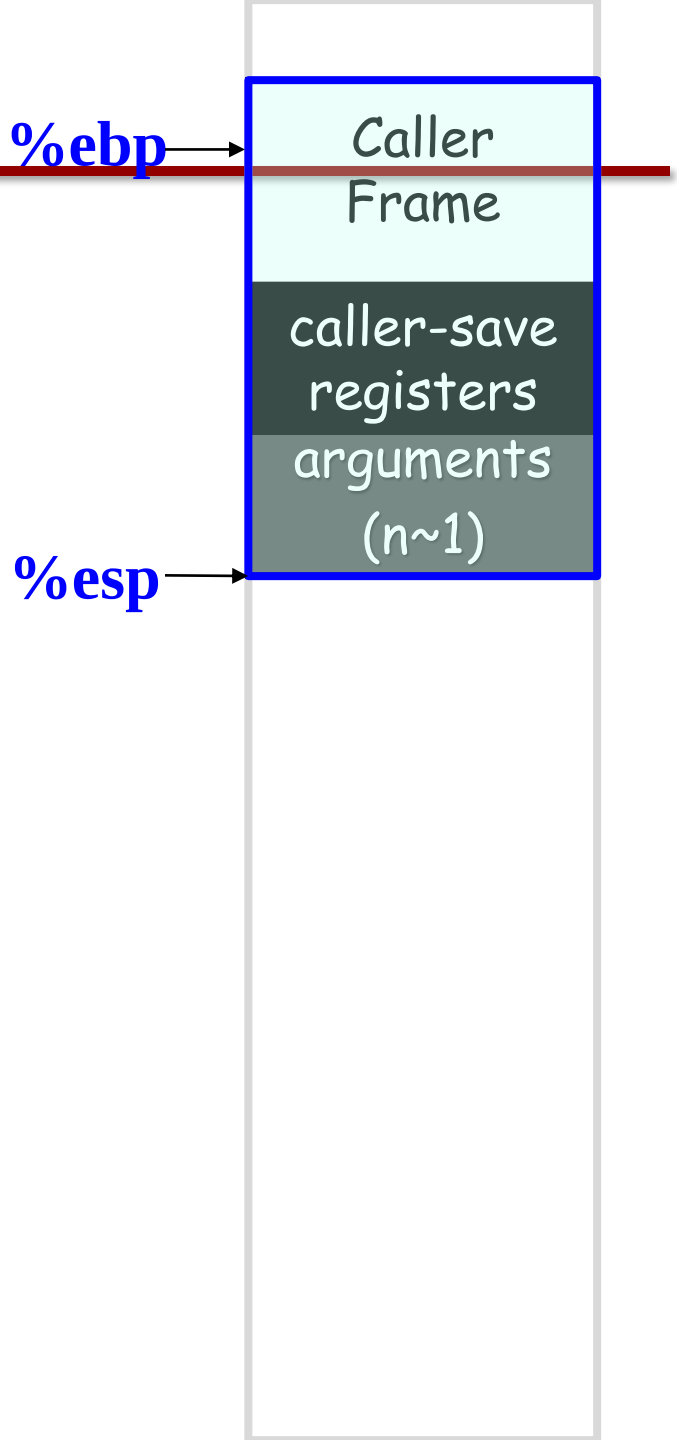
Put it Together



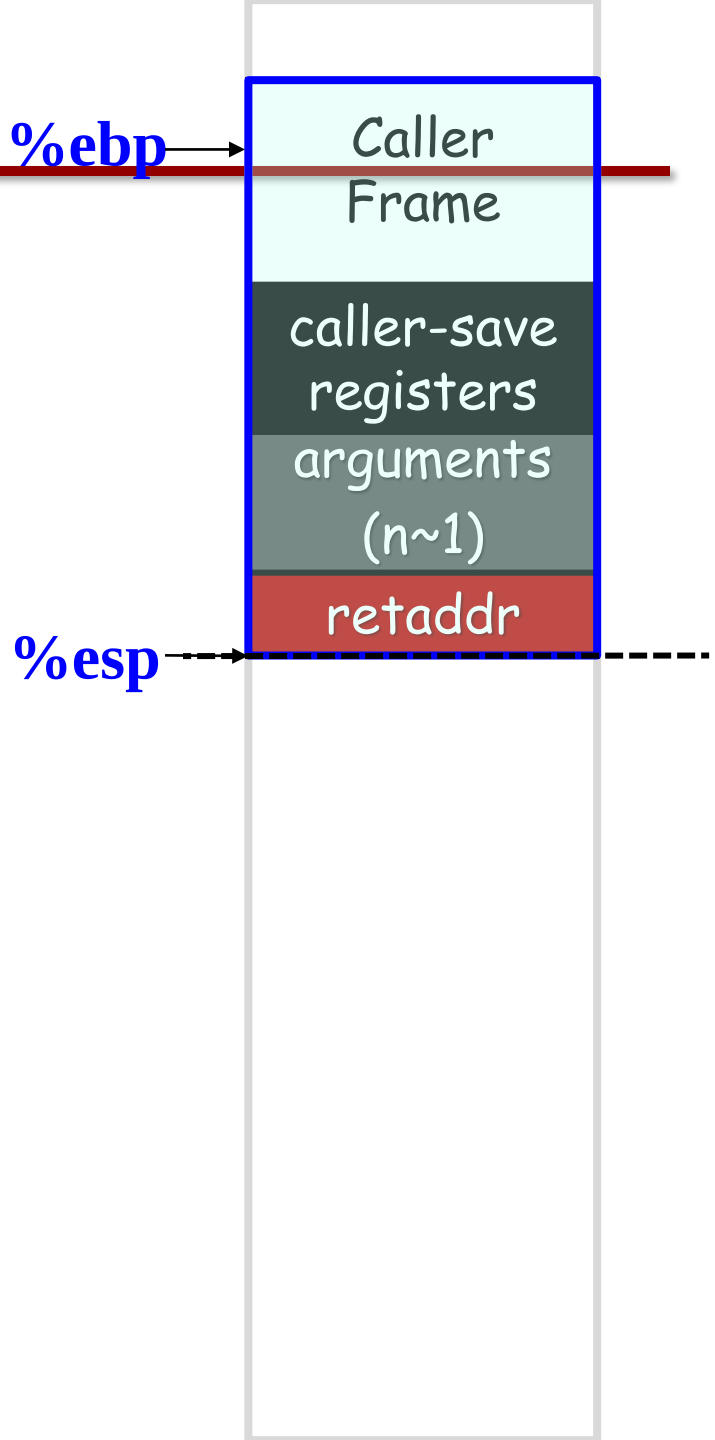
1. Save caller-save registers (%rbx, %rbp, %r12~15, %rsp以外的寄存器)

Put it Together

1. Save caller-save registers
2. Push actual arguments from right to left



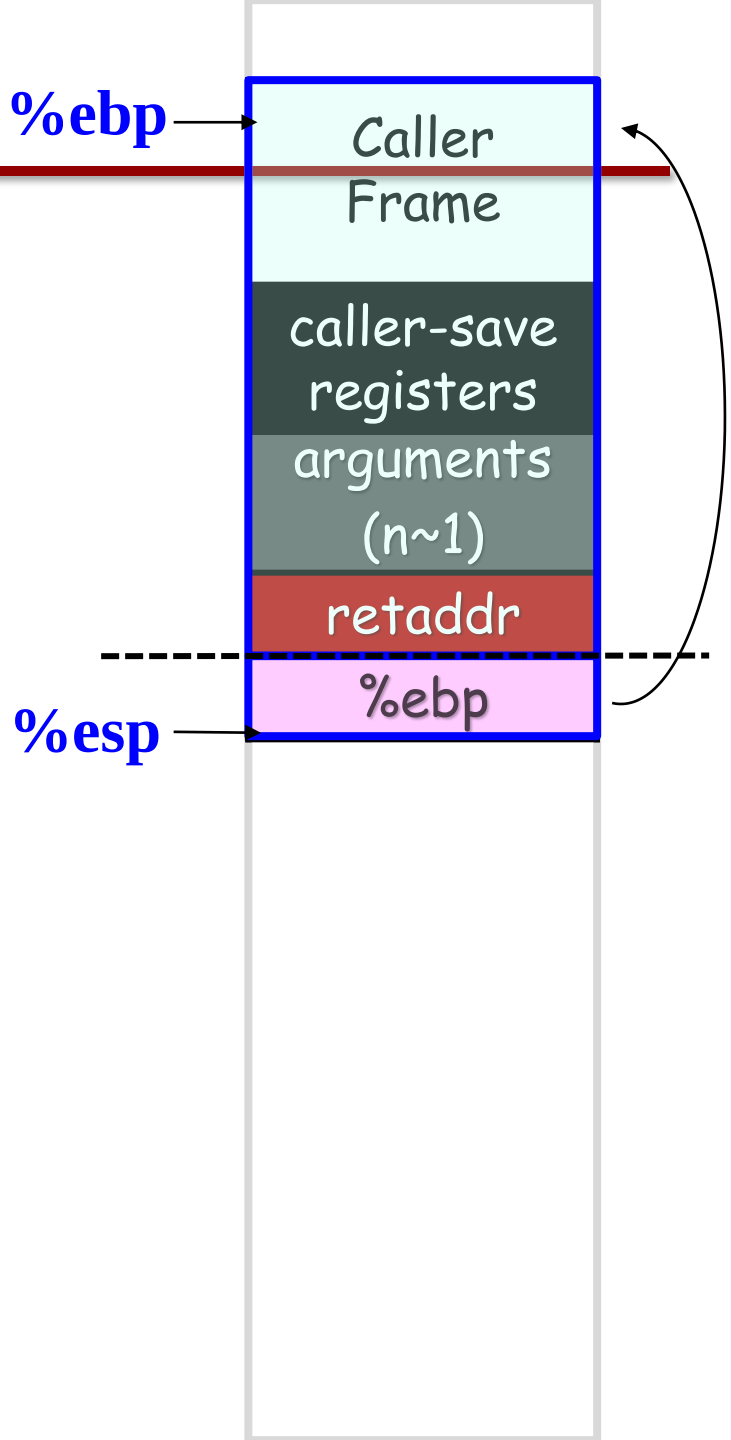
Put it Together



1. Save caller-save registers
2. Push actual arguments from right to left
3. Call instruction
   - Save return address
   - Transfer control to callee

Put it Together

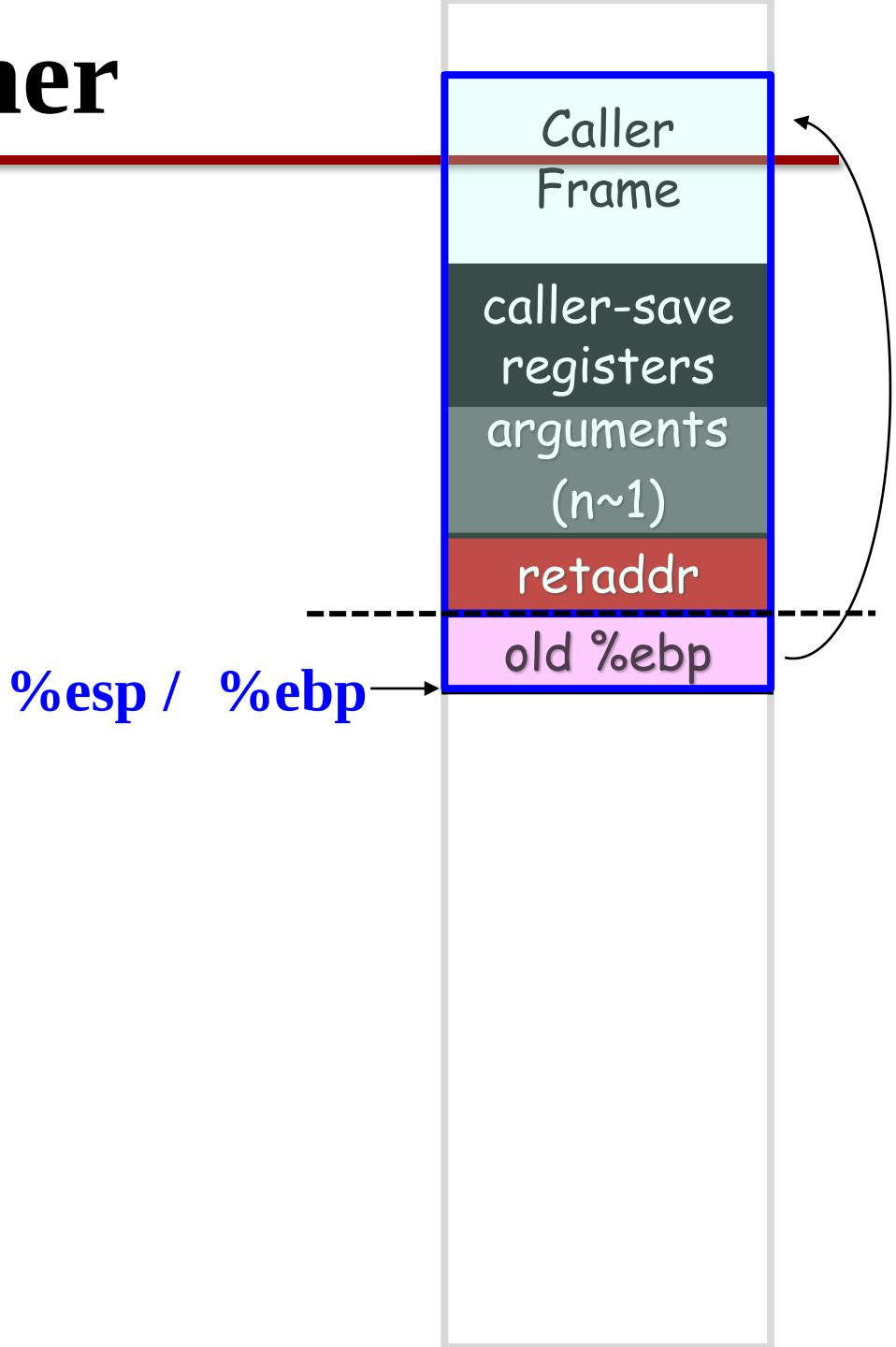
4. Save caller %ebp



Put it Together

4. Save caller %ebp

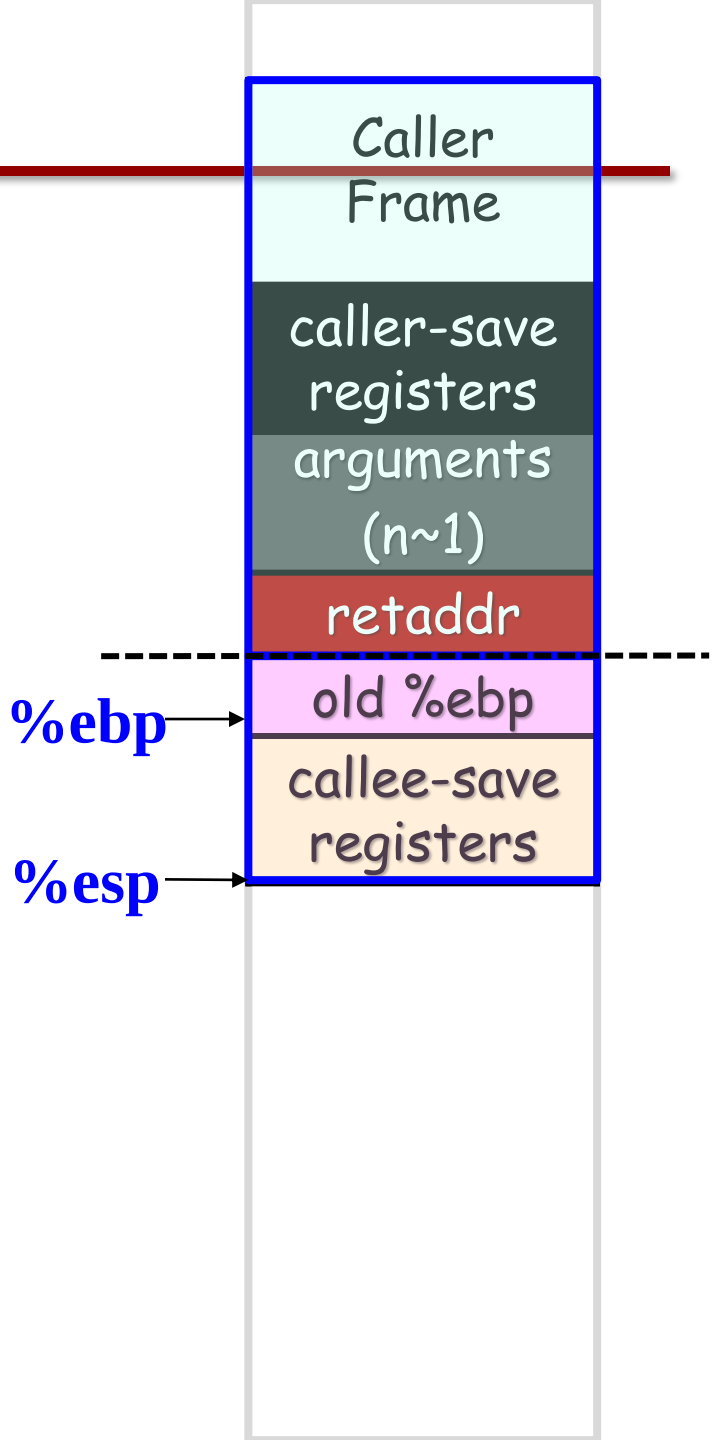
5. Set callee %ebp



Put it Together

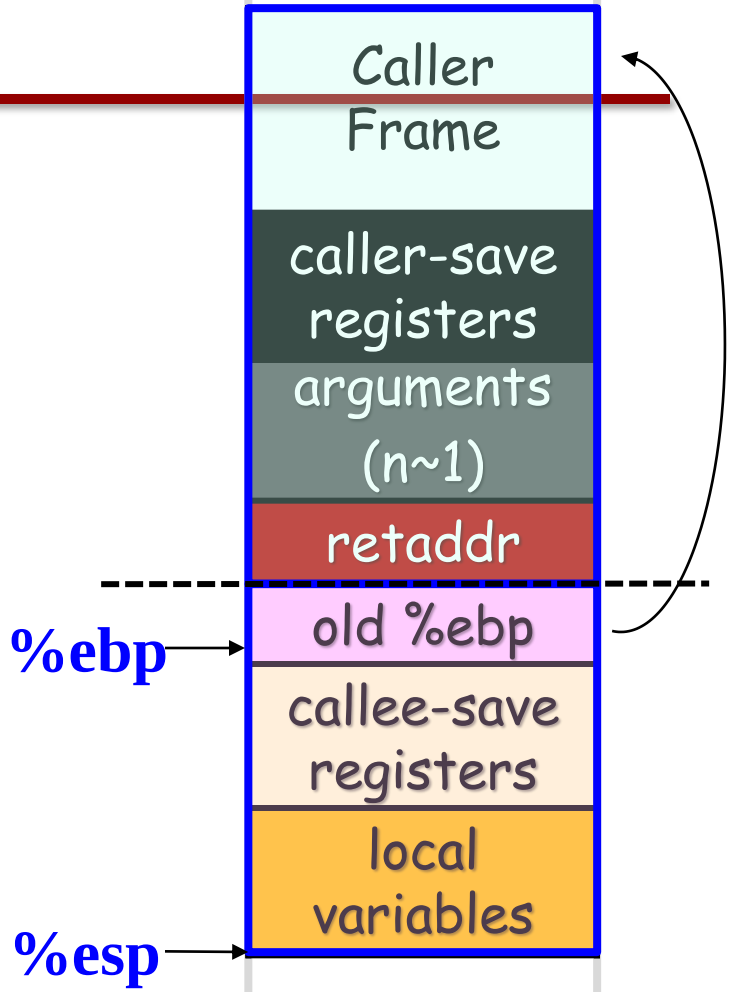
4. Save caller %ebp
5. Set callee %ebp
6. Save callee-save registers

   (%rbx, %rbp, %r12~15)



Put it Together

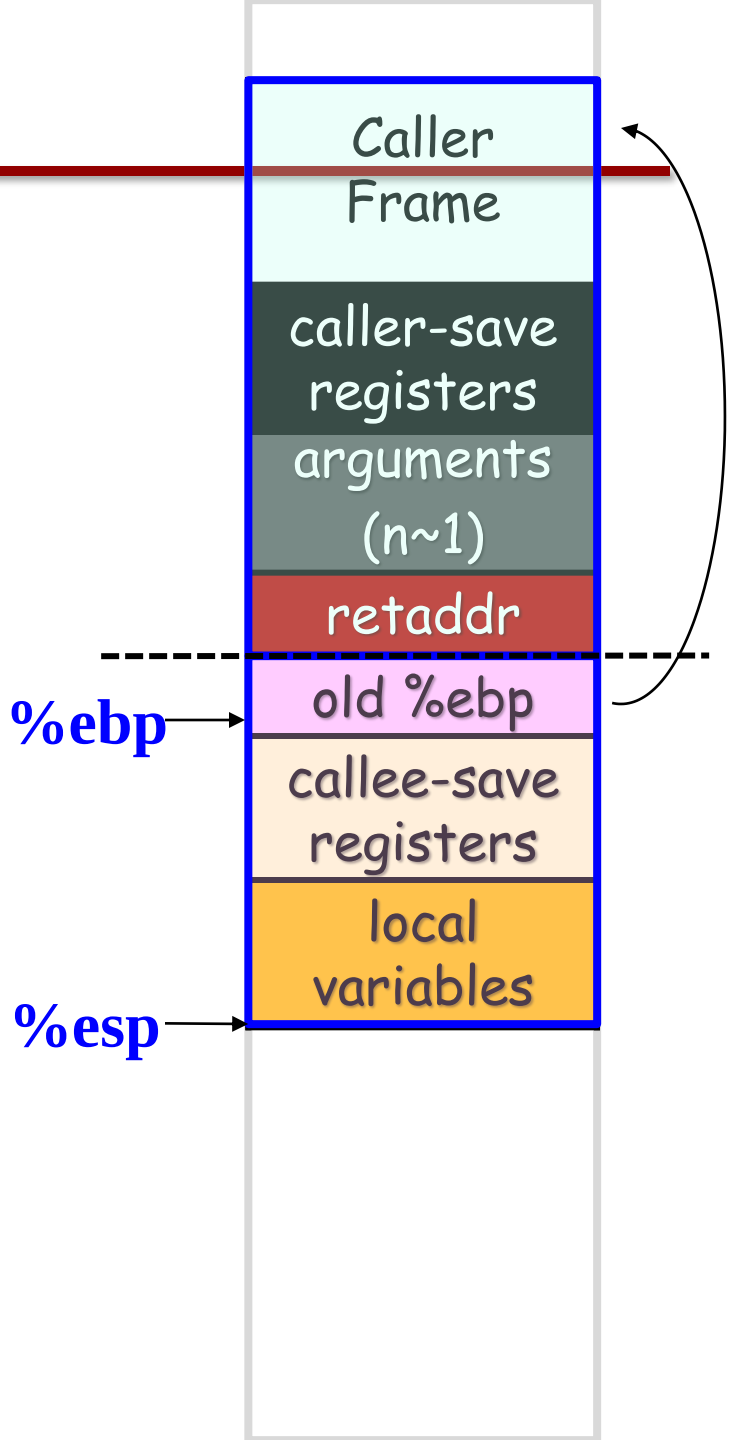
4. Save caller %ebp
5. Set callee %ebp
6. Save callee-save registers (%rbx, %rbp, %r12~15)
7. Allocate space for local variable



Put it Together

. . .

n-4. save return value in %eax

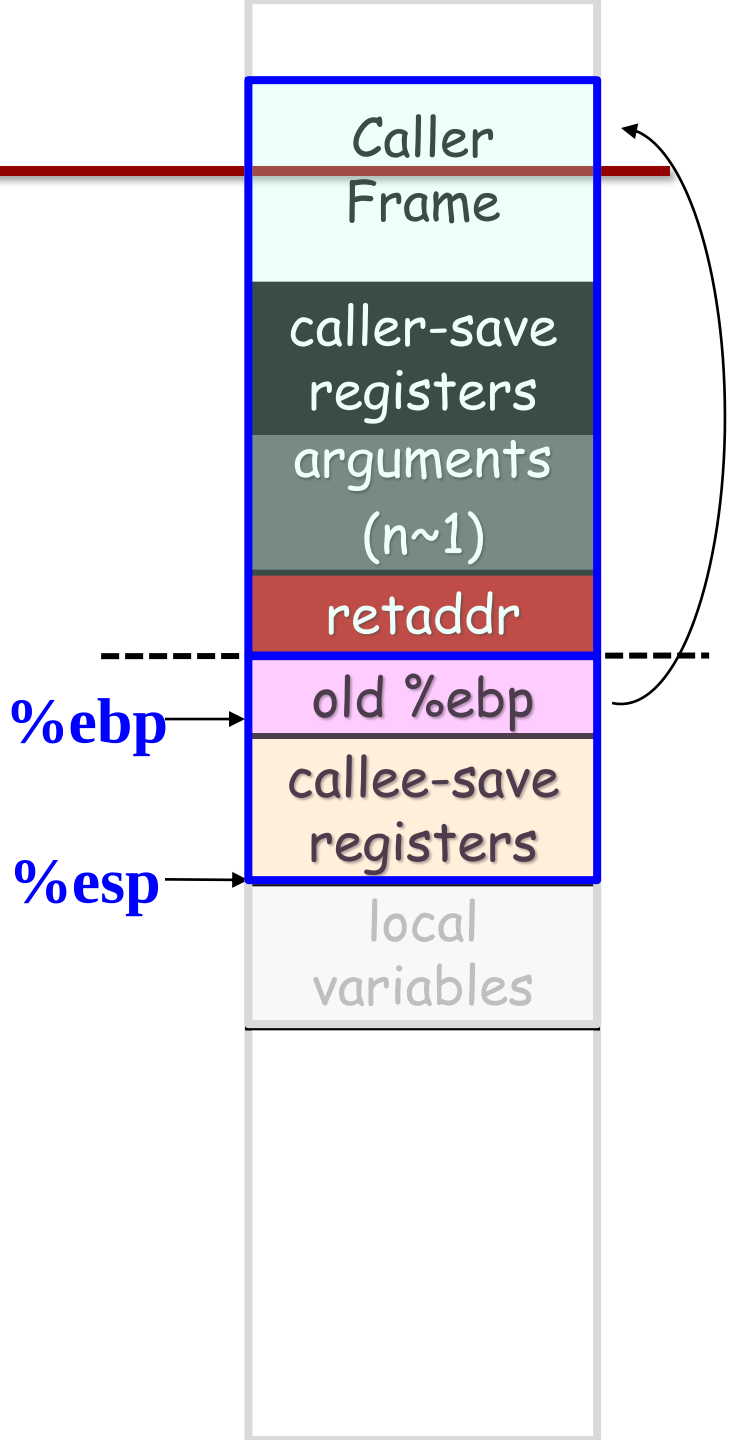


Put it Together

. . .

n-4. save return value in %eax

n-3. de-allocate local variable



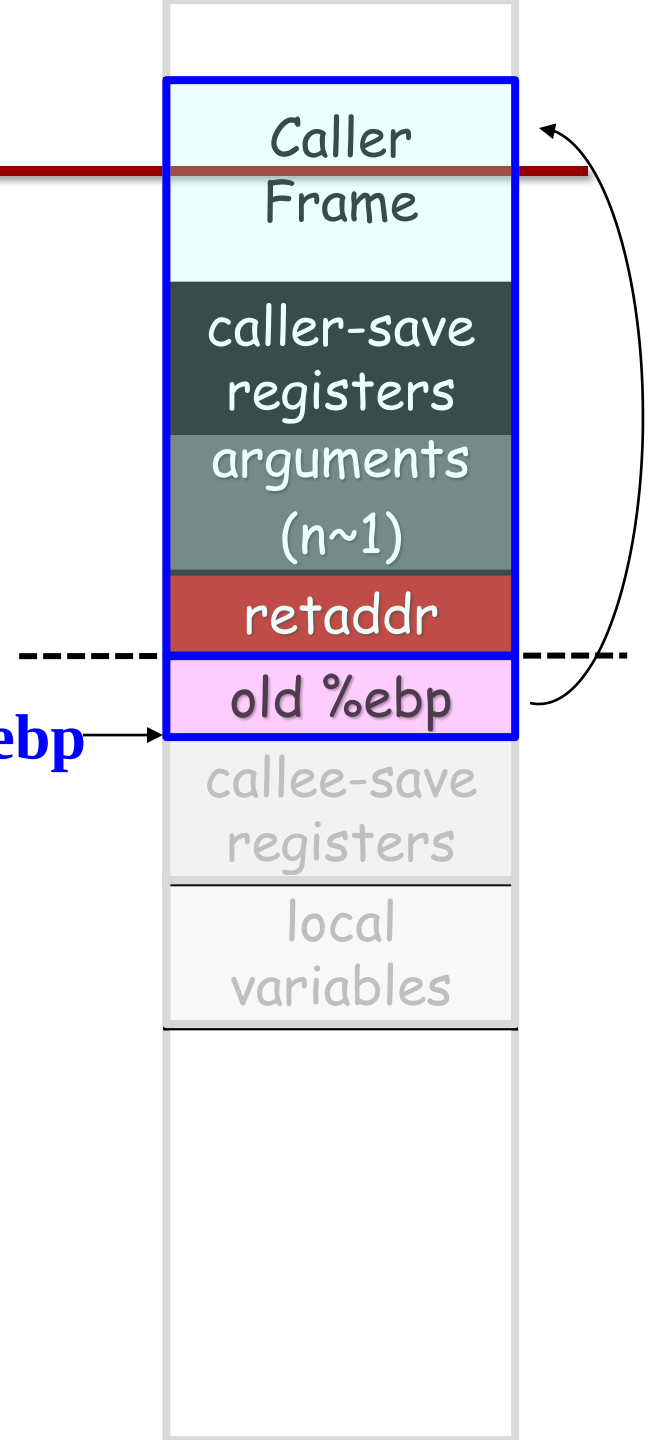
Put it Together

. . .

n-4. save return value in %eax

n-3. de-allocate local variable

n-2. Restore callee-save registers



Put it Together

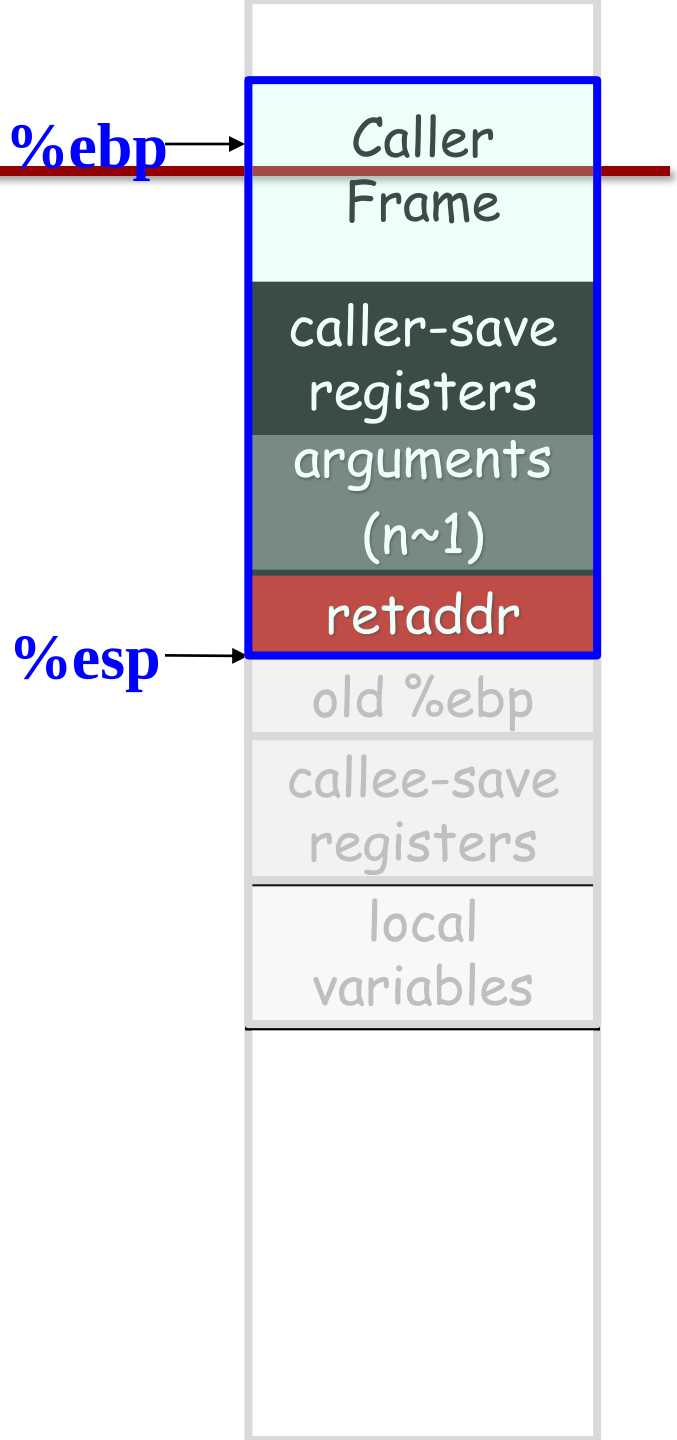
. . .

n-4. save return value in %eax

n-3. de-allocate local variable

n-2. Restore callee-save registers

n-1. Restore caller %ebp



Put it Together

. . .

n-4. save return value in %eax

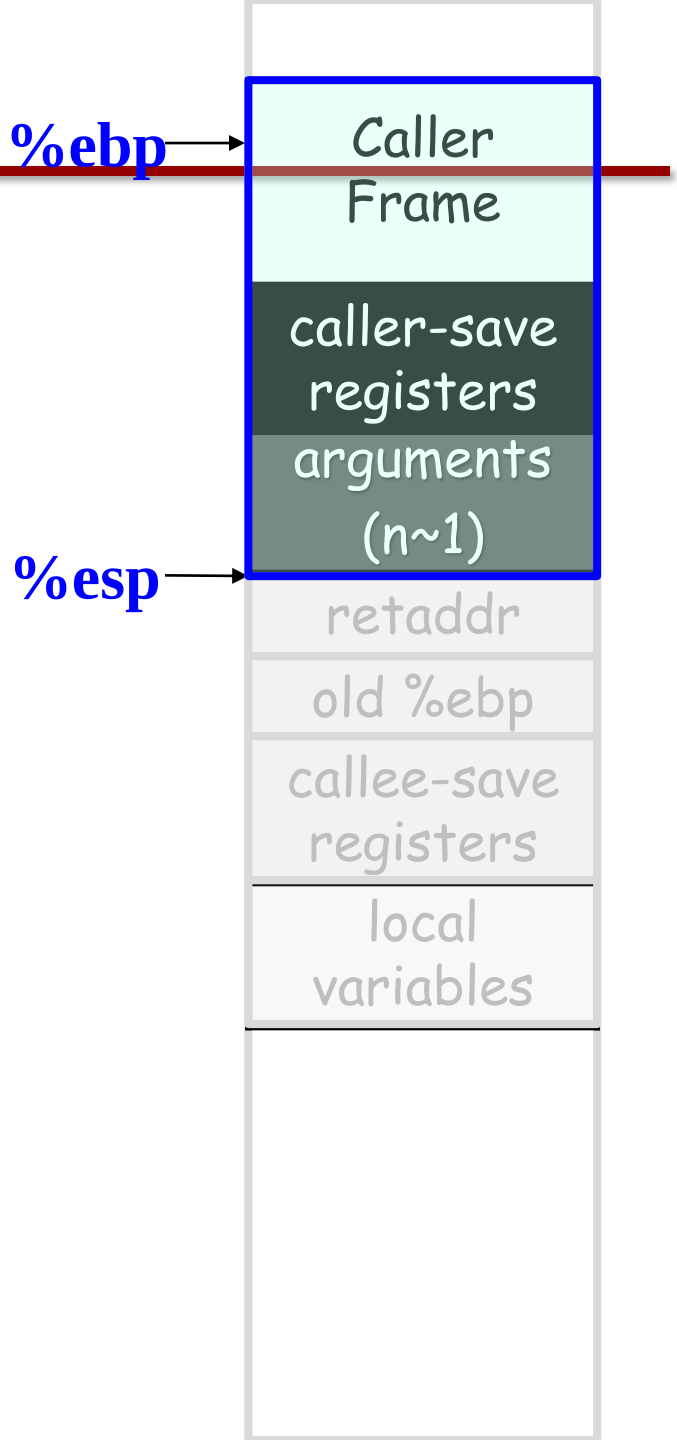
n-3. de-allocate local variable

n-2. Restore callee-save registers

n-1. Restore caller %ebp

n. Ret instruction

- pop return address
- Transfer control to caller



Example

```
int swap_add(int *xp, int *yp)
{
        int x = *xp;
        int y = *yp;

        *xp = y;
        *yp = x;
        return x + y;
}
```

Example

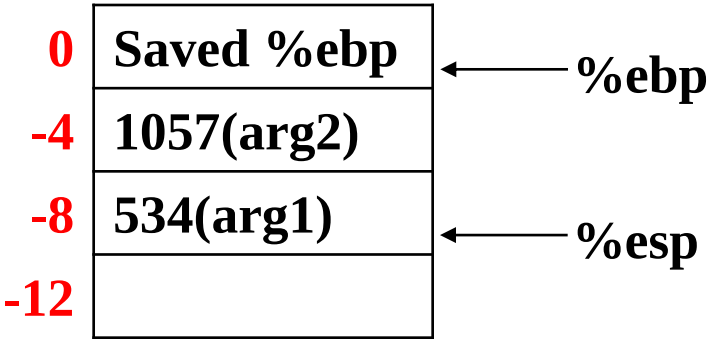
```
int caller()
{
        int arg1 = 534;
        int arg2 = 1057;
        int sum = swap_add(&arg1, &arg2);
        int diff = arg1 - arg2;

        return sum * diff;
}
```

Example

Before

int sum = swap_add(&arg1, &arg2);

Stack frame for caller



Parameter Passing

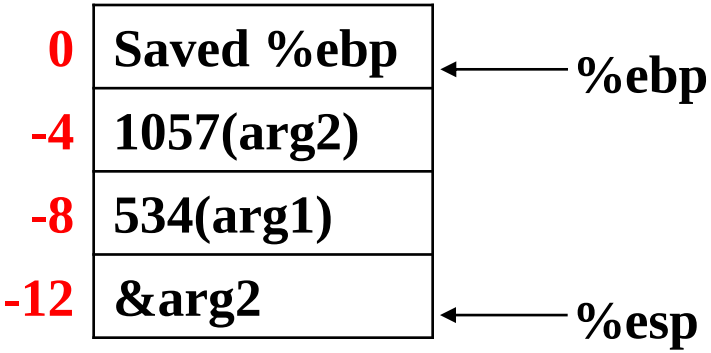
1 leal -4(%ebp),%eax

Compute &arg2

2 pushl %eax

Push &arg2

Stack frame for caller



Parameter Passing

1 leal -4(%ebp),%eax

Compute &arg2

2 pushl %eax

Push &arg2

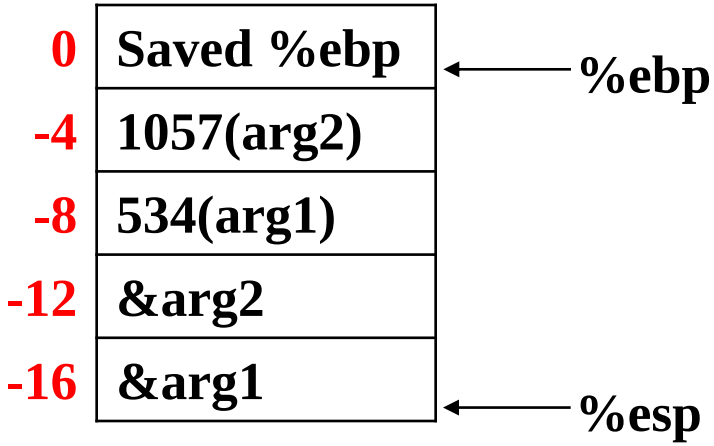
3 leal -8(%ebp),%eax

Compute &arg1

4 pushl %eax

Push &arg1

Stack frame for caller



Call Instruction

1 leal -4(%ebp),%eax

Compute &arg2

2 pushl %eax

Push &arg2

3 leal -8(%ebp),%eax

Compute &arg1

4 pushl %eax

Push &arg1

5 call swap_add

Call the swap_add function

Stack frame for caller

0	Saved %ebp	← %ebp
-4	1057(arg2)	
-8	534(arg1)	
-12	&arg2	
-16	&arg1	
-20	Return Addr	← %esp

Setup code in swap_add

swap_add:

1 pushl %ebp

***Save old* %ebp**

Stack frame for caller

0	**Saved %ebp**	← **%ebp**
-4	**1057(arg2)**	
-8	**534(arg1)**	
-12	**yp(&arg2)**	
-16	**xp(&arg1)**	
-20	**Return Addr**	
-24	**Saved %ebp**	← **%esp**

Setup code in swap_add

swap_add:

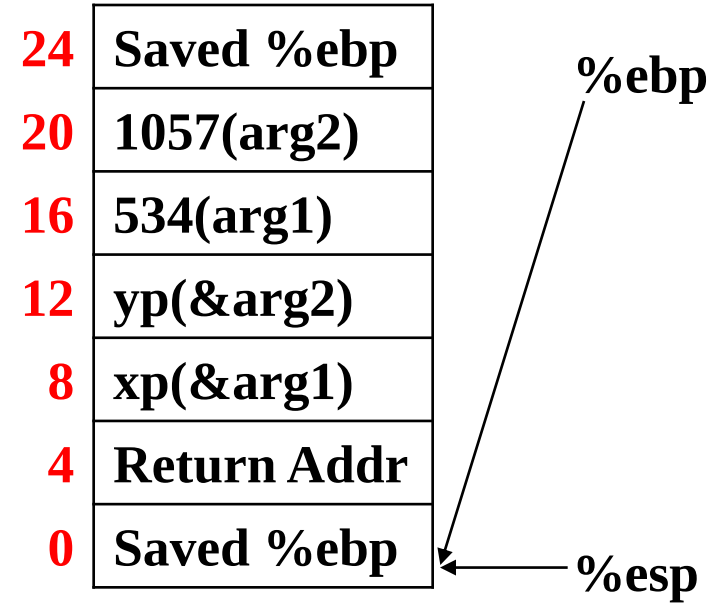
1 pushl %ebp

Save old **%ebp**

2 movl %esp,%ebp

Set **%ebp** *as frame pointer*

Stack frame for caller



Stack frame for swap_add

Setup code in swap_add

swap_add:

1 pushl %ebp

Save old **%ebp**

2 movl %esp,%ebp

Set **%ebp** *as frame pointer*

3 pushl %ebx

Save %ebx

Stack frame for caller

24	Saved %ebp	
20	1057(arg2)	
16	534(arg1)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	← %ebp
-4	Saved %ebx	← %esp

Stack frame for swap_add

Body code in swap_add

5 movl 8(%ebp),%edx

Get xp

%edx	xp(=&arg1=%ebp+16)

Stack frame for caller

24	Saved %ebp	
20	1057(arg2)	
16	534(arg1)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	← %ebp
-4	Saved %ebx	← %esp

Stack frame for swap_add

Body code in swap_add

5 movl 8(%ebp),%edx

Get xp

6 movl 12(%ebp),%ecx

Get yp

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)

Stack frame for caller

24	Saved %ebp	
20	1057(arg2)	
16	534(arg1)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	← %ebp
-4	Saved %ebx	← %esp

Stack frame for swap_add

Body code in swap_add

```
5  movl 8(%ebp),%edx
                    Get xp
6  movl 12(%ebp),%ecx
                    Get yp
7  movl (%edx),%ebx
                    Get x
8  movl (%ecx),%eax
                    Get y
```

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	534
%eax	1057

Stack frame for caller

24	Saved %ebp	
20	1057(arg2)	
16	534(arg1)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	← %ebp
-4	Saved %ebx	← %esp

Stack frame for swap_add

Body code in swap_add

9 movl %eax, (%edx)

Store y at *xp

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	534
%eax	1057

Stack frame for caller

24	Saved %ebp	
20	1057(arg2)	
16	1057(arg1)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	← %ebp
-4	Saved %ebx	← %esp

Stack frame for swap_add

Body code in swap_add

9 movl %eax, (%edx)

*Store y at *xp*

10 movl %ebx, (%ecx)

*Store x at *yp*

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	534
%eax	1057

Stack frame for caller

24	Saved %ebp	
20	534(arg2)	
16	1057(arg1)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	← %ebp
-4	Saved %ebx	← %esp

Stack frame for swap_add

Body code in swap_add

9 movl %eax, (%edx)

*Store y at *xp*

10 movl %ebx, (%ecx)

*Store x at *yp*

11 addl %ebx,%eax

Set return value = x+y

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	534
%eax	1591

Stack frame for caller

24	Saved %ebp	
20	534(arg2)	
16	1057(arg1)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	← %ebp
-4	Saved %ebx	← %esp

Stack frame for swap_add

Finishing code in swap_add

12 popl %ebx *Restore* %ebx

Stack frame for caller

24	Saved %ebp
20	534(arg2)
16	1057(arg1)
12	yp(&arg2)
8	xp(&arg1)
4	Return Addr
0	Saved %ebp
-4	Saved %ebx

%ebp

%esp

Stack frame for swap_add

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

Finishing code in swap_add

12 popl %ebx *Restore* %ebx

13 movl %ebp, %esp

Restore %esp

Stack frame for caller

24	Saved %ebp
20	534(arg2)
16	1057(arg1)
12	yp(&arg2)
8	xp(&arg1)
4	Return Addr
0	Saved %ebp
-4	Saved %ebx

%ebp

%esp

Stack frame for swap_add

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

Finishing code in swap_add

12 popl %ebx *Restore* %ebx

13 movl %ebp, %esp *Restore* %esp

14 popl %ebp *Restore* %ebp

Stack frame for caller

0	Saved %ebp	← %ebp
-4	534(arg2)	
-8	1057(arg1)	
-12	yp(&arg2)	
-16	xp(&arg1)	
-20	Return Addr	← %esp
	Saved %ebp	
	Saved %ebx	

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

Finishing code in swap_add

```
12  popl %ebx        Restore %ebx
13  movl %ebp, %esp
                     Restore %esp
14  popl %ebp        Restore %ebp
15  ret              Return to caller
```

- Call by value

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

Stack frame for caller

0	Saved %ebp	← %ebp
-4	534(arg2)	
-8	1057(arg1)	
-12	yp(&arg2)	
-16	xp(&arg1)	← %esp
-20	Return Addr	
	Saved %ebp	
	Saved %ebx	

```
int proc(void) {
    int x, y;
    scanf("%x %x", &y, &x);
    return x-y;
}
```

过程proc开始时，esp值为0x800040，ebp值为0x00060，scanf输入值为0x46和0x53，“%x %x”地址为0x300070.

A. 第3行ebp的值被设为多少？

B. 第4行esp的值被设为多少？

C. 局部变量x和y的地址是多少？

D. 画出scanf返回后的栈图

E. 指出proc函数未使用的栈区的地址

■ GCC产生如下汇编代码

```
1.  proc
2.  Pushl %ebp
3.  Movl  %esp, %ebp
4.  Subl  $40, %esp
5.  Leal   -4(%ebp), %eax
6.  Movl  %eax, 8(%esp)
7.  Leal  -8(%ebp), %eax
8.  Movl  %eax, 4(%esp)
9.  Movl  $.LC0, (%esp);常量字符串"%x %x"的地址
10. Call scanf
11. Movl -4(%ebp), %eax
12. Subl -8(%ebp), %eax
13. Movl %ebp, %esp,
14. Popl %ebp
15. Ret
```

```
int proc(void) {
    int x, y;
    scanf("%x %x", &y, &x);
    return x-y;
}
```

过程proc开始时，esp值为0x800040，ebp值为0x00060，scanf输入值为0x46和0x53，“%x %x”地址为0x300070.

A. 第3行ebp的值被设为多少？**0x80003c;**

B. 第4行esp的值被设为多少？**0x800014;**

C. 局部变量x和y的地址是多少？**x:0x800038, y: 0x800034;**

D. 画出scanf返回后的栈图

E. 指出proc函数未使用的栈区的地址

0x800020~0x800033

■ GCC产生如下汇编代码

```
1.  proc
2.  Pushl %ebp
3.  Movl  %esp, %ebp
4.  Subl  $40, %esp
5.  Leal   -4(%ebp), %eax
6.  Movl  %eax, 8(%esp)
7.  Leal  -8(%ebp), %eax
8.  Movl  %eax, 4(%esp)
9.  Movl  $.LC0, (%esp);常量字符串"%x %x"的地址
10. Call scanf
11. Movl -4(%ebp), %eax
12. Subl -8(%ebp), %eax
13. Movl %ebp, %esp,
14. Popl %ebp
15. Ret
```

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

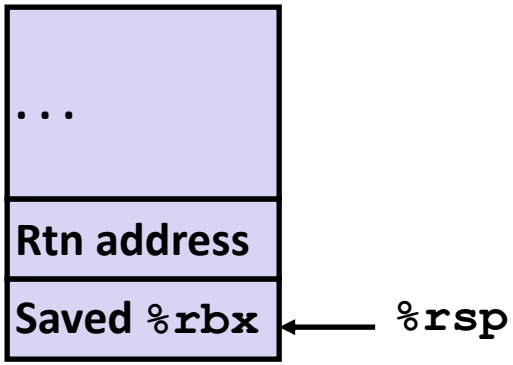
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

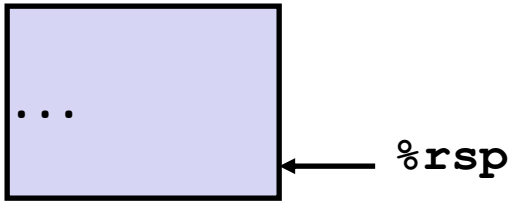
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value



课堂练习

- 一个具有通用结构的C函数如下：
- long rfun (unsigned long x) {
- if (_____)
- return _____;
- unsigned long nx =_____;
- long rv = rfun(nx);
- return _____;
- }
- 请填写C语言代码中缺失的表达式；rfun存储在被调用者保存寄存器%rbx中的值是_____。

- **GCC产生的汇编代码如下：**
- **long rfun (unsigned long x)**
- **x in %rdi**
- **rfun:**
- **pushq %rbx**
- **movq %rdi, %rbx**
- **movl $0, %eax**
- **testq %rdi, %rdi**
- **je .L2**
- **shrq $2, $rdi**
- **call rfun**
- **addq %rbx, %rax**
- **.L2**
- **popq %rbx**
- **ret**

课堂练习

程序填空，变量映射关系，并解释函数功能

```
Int fun_a(unsigned x) {
  int val =
                x     ) {

   x>>=1
  }
  return val&0x01;
}
```

- x@$ebp+8

```
  Movl 8(%ebp), %edx
  Movl $0, %eax
  Testl %edx, %edx
  Je .L7
.L10:
  xorl %edx, %eax
  shrl %edx  ;逻辑右移1位
  jne .L10
.L7:
  andl $1, %eax
```

课堂练习

程序填空，变量映射关系，并解释函数功能

```
Int fun_a(unsigned x) {
  int val = 0;
  while (    x     ) {
    val ^= x;
    x>>=1
  }
  return val&0x01;
}
```

- x@$ebp+8

```
  Movl 8(%ebp), %edx
  Movl $0, %eax
  Testl %edx, %edx
  Je .L7
.L10:
  xorl %edx, %eax
  shrl %edx  ;逻辑右移1位
  jne .L10
.L7:
  andl $1, %eax
```

课堂练习

- 函数fun_b(unsigned long x) {
  - Long val = 0;
  - Long i;
  - For (… ; … ; …) {
    - …
  - }
  - Return val;
- }
- Gcc生成的汇编代码如右所示。
- 完成下面工作：
  - A. 根据汇编代码，填写C代码缺失的部分；
  - B. 解释循环前为什么没有初始测试，也没有初始跳转到循环内部的测试部分；
  - C. 用自然语言描述这个函数的功能

```
# x in %rdi
func_b:
    Movl $64, %edx
    movl $0, %eax
.L10:
    movq %rdi, %rcx
    andl  $1, %ecx
    addq %rax, %rax
    orq % rcx, %rax
    shrq %rdi
    subq $1, %rdx
    jne .L10
    ret
```

练习答案

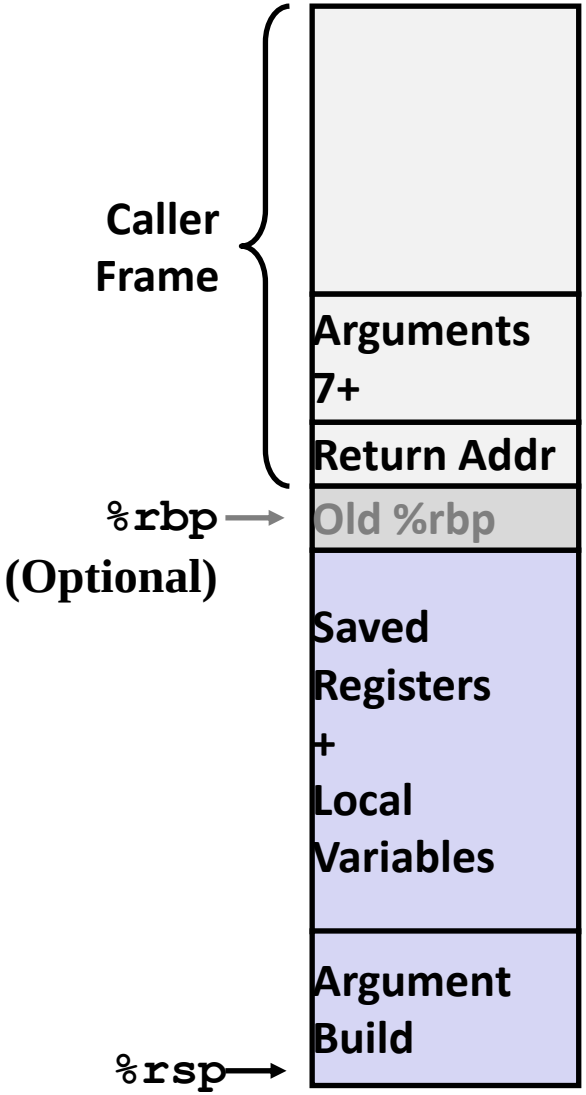
- fun_b(unsigned long x) {
  - Long val = 0;
  - Long i;
  - For (i=64; i!=0 ; i--) { // 尽量忠于原意
    - Val = (val << 1) | (x&1)
    - X >>= 1;
  - }
  - Return val;
- }
- 2）因为循环次数是常量
- 3）将x镜像反转

Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P

x86-64 Procedure Summary

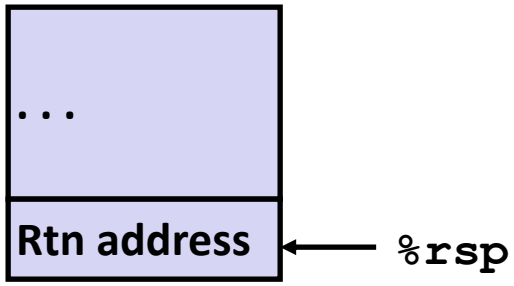
- Important Points
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in **`%rax`**
- Pointers are addresses of values
  - **On stack or global**



Callee-Saved Example #1

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Initial Stack Structure



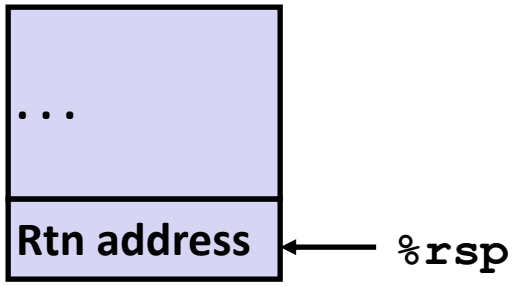
- **`x` comes in register `%rdi`.**
- **We need `%rdi` for the call to `incr`.**
- **Where should be put `x`, so we can use it after the call to `incr`?**

Callee-Saved Example #2

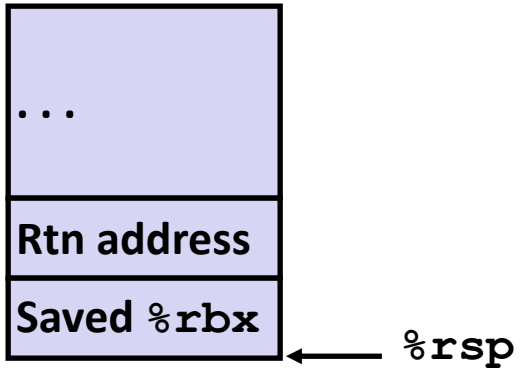
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

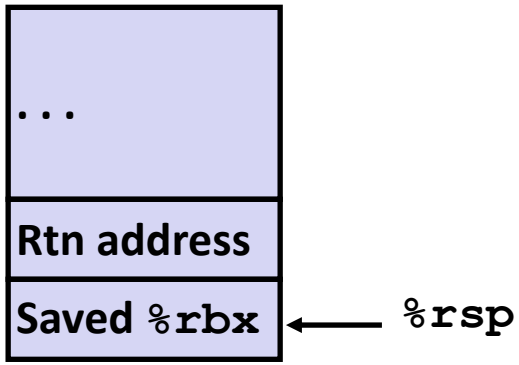


Callee-Saved Example #3

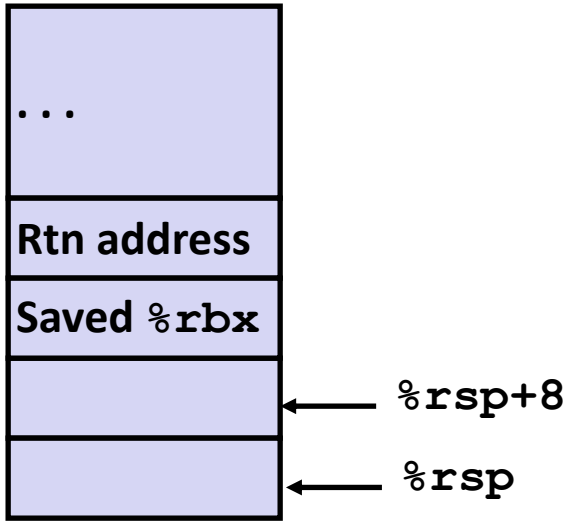
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

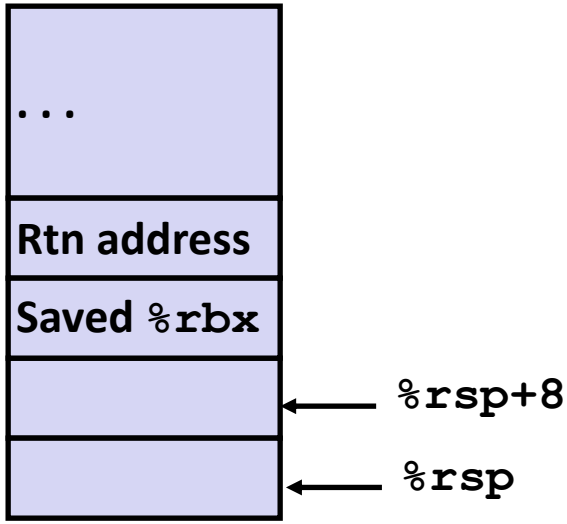


Callee-Saved Example #4

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



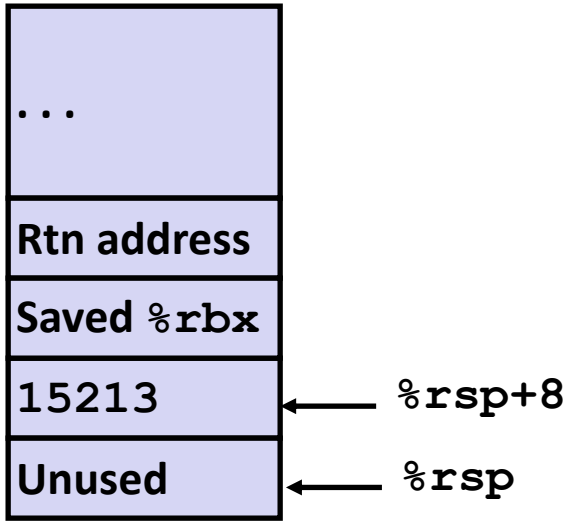
- **`x` is saved in `%rbx`, a callee saved register**

Callee-Saved Example #5

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



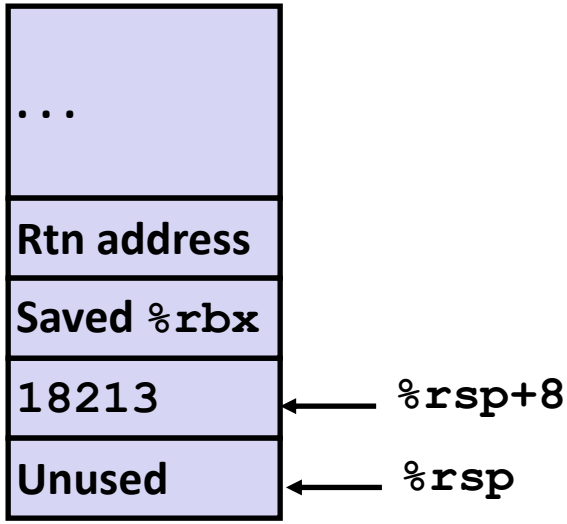
- **`x` is saved in `%rbx`, a callee saved register**

Callee-Saved Example #6

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



Upon return from `incr`:

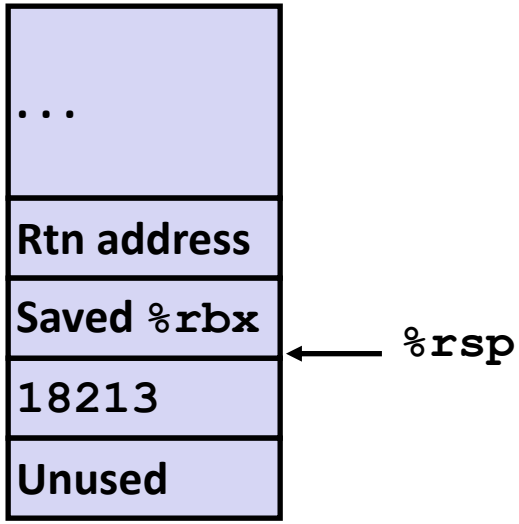
- `x` safe in `%rbx`
- Return val `v2` in `%rax`
- Compute `x+v2`:
  `addq %rbx, %rax`

Callee-Saved Example #7

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



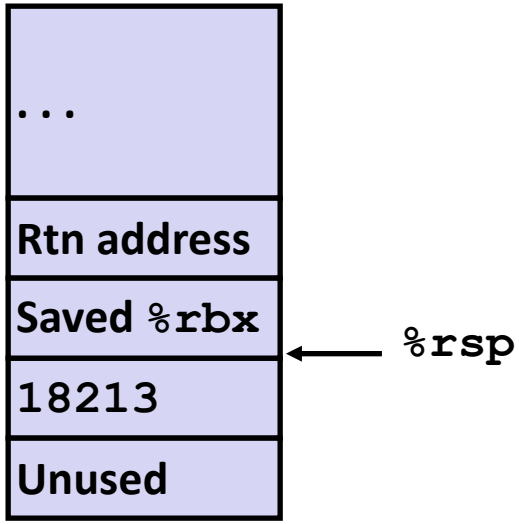
- **Return result in `%rax`**

Callee-Saved Example #8

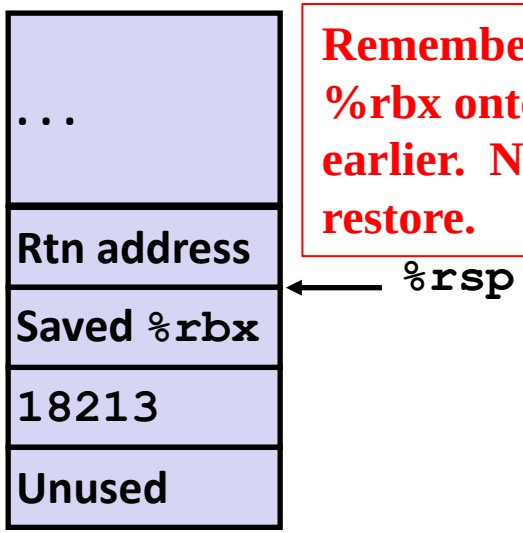
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



final Stack Structure



Remember: we saved %rbx onto stack earlier. Need to restore.