汇编

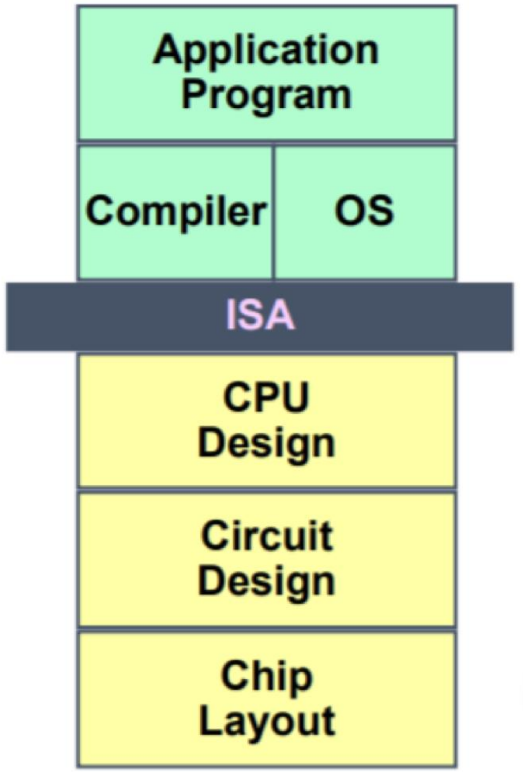
计算机系统导论 讨论班 @ 北京大学

向星雨

9/21/22, 9/28/22, 10/4/22

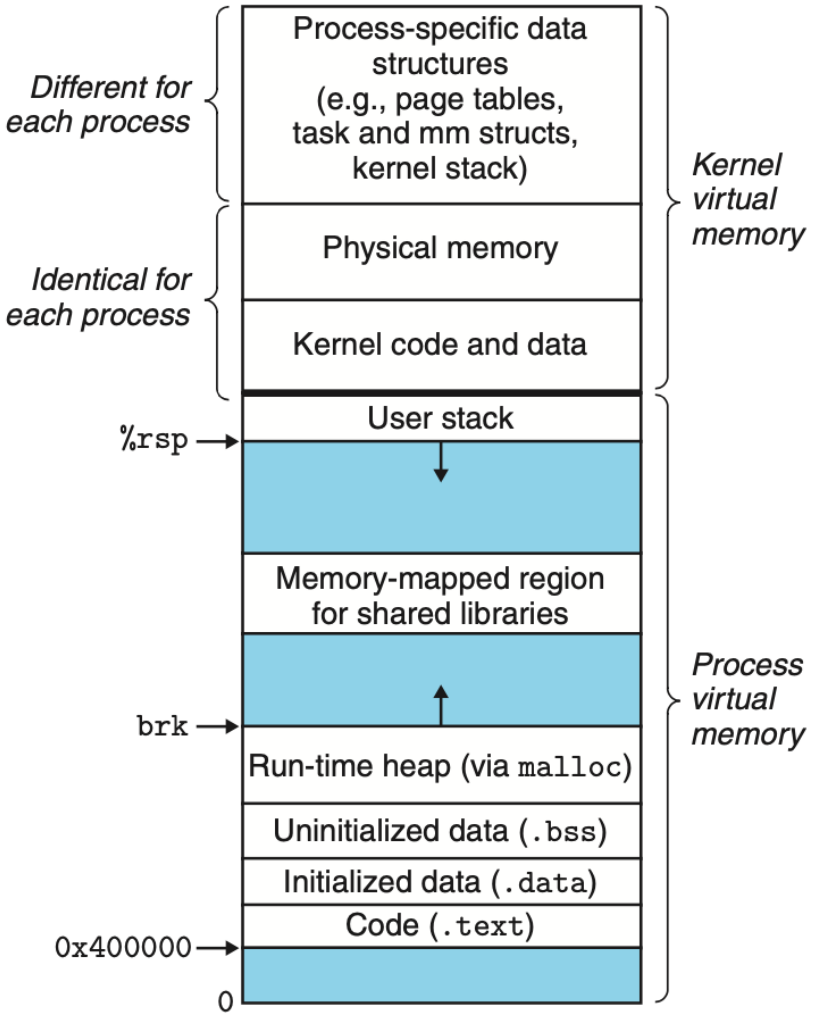
数据

指令集架构


Application Program
Compiler
OS
ISA
CPU Design
Circuit Design
Chip Layout

虚拟地址空间

- 每个 **进程** 享有同样的虚拟地址空间



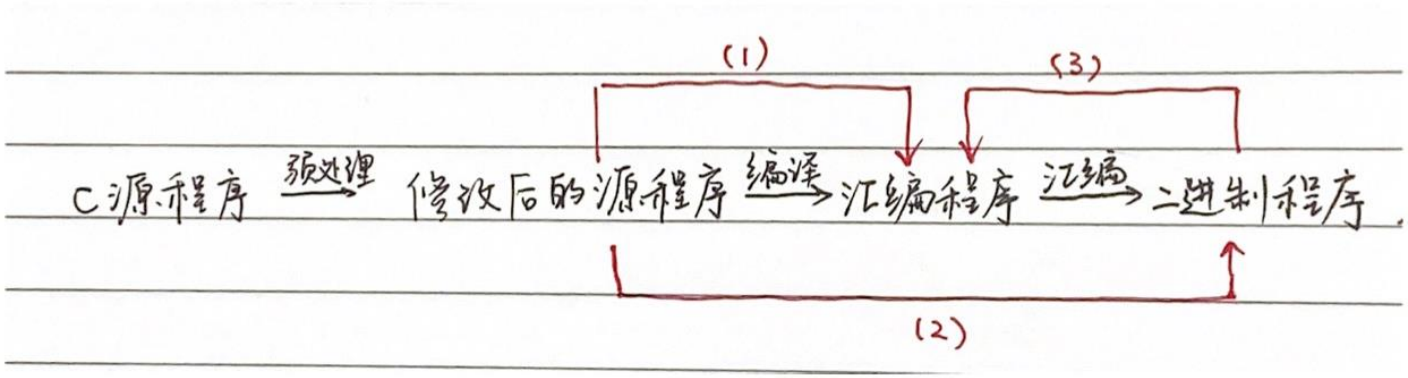
工具链

(1) gcc [-O] -S <file> [-o]

(2) gcc [-O] -c <file> [-o]

(3) objdump -d <file> (> ...)

- gdb and bomblab



编译器优化

助教 李翰禹

再具体解释一下优化等级的问题
-O0是几乎所有优化都不做
-O或者-O1是做一部分优化
-Og是-O1优化去除会影响调试器正常工作的优化
-O2是-O1不考虑时间和空间的权衡的情况下做更多优化
-O3是-O2基础上做更多优化，比如一些向量化和循环展开的优化
-Os是-O2优化去除会增加目标代码长度的优化
-Ofast是无视严格标准的优化

数据的格式

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

- 关于浮点数

整数寄存器

- 存储整型数据和指针
- 各寄存器的功能

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

非运算操作

- 操作数 (operand) 的分类
- 种类
  - 访问 / 寻址
  - 传送　`mov`
  - 栈操作　`push`, `pop`

运算

- 取地址操作 `lea`
- 运算
  - 一元运算 `inc`, `dec`, `neg`, `not`
  - 二元运算 `add`, `sub`, `imul`, `xor`, `or`, `and`
  - 移位运算 `sal`, `shl`, `sar`, `shr`
  - 同时适用于无符号和补码

控制

条件码

- 条件码都只有一个 bit
- 描述 **最近** 的运算的属性

条件码的设置

- 算术运算（+-*/, neg, not）很好理解，直接看结果
- 逻辑运算（xor, or, and）
  - 进位 CF、溢出 OF 一定置 0，零标志 ZF、符号标志 SF 看结果
- 移位运算
  - 进位 CF 设置为最后一个被移出的位；溢出 OF 置为 0；ZF、SF 看结果。
- inc, dec
  - 不会改变进位标志

- cmp (-) 与 test (&)
- (testq %rax, %rax) = (cmpq $0, %rax)

条件码的使用

- set 指令
  - 目的是寄存器最低 byte 或内存中的一个 byte
- jmp 指令
  - 条件跳转只能是直接的
  - jmp *%rax 与 jmp *(%rax)
- 结合 cmp 和 test

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 <=)

条件传送

- cmov 指令
  - dest 只能是寄存器
  - 不能传送单字节
- 与条件控制的对比
  - 适用情况更少
  - **有时** 性能更好

```
v = test-expr ? then-expr : else-expr;
```

```
v  = then-expr;
ve = else-expr;
t  = test-expr;
if (!t) v = ve;
```

循环：do-while, while, for, switch

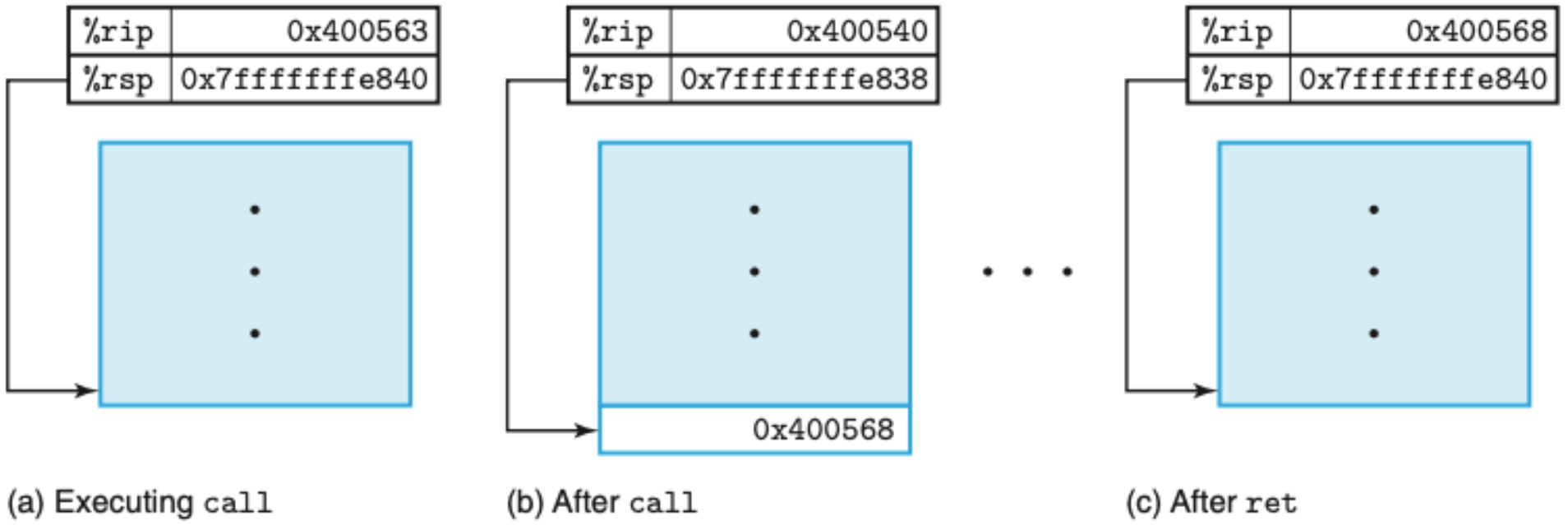
控制语句

- `break / continue`
- 手动添加 `label` 和 `goto`

- 编译优化

函数调用

控制转移



(a) Executing `call` (b) After `call` (c) After `ret`

Figure 3.26 Illustration of `call` and `ret` functions. The `call` instruction transfers control to the start of a function, while the `ret` instruction returns back to the instruction following the call.

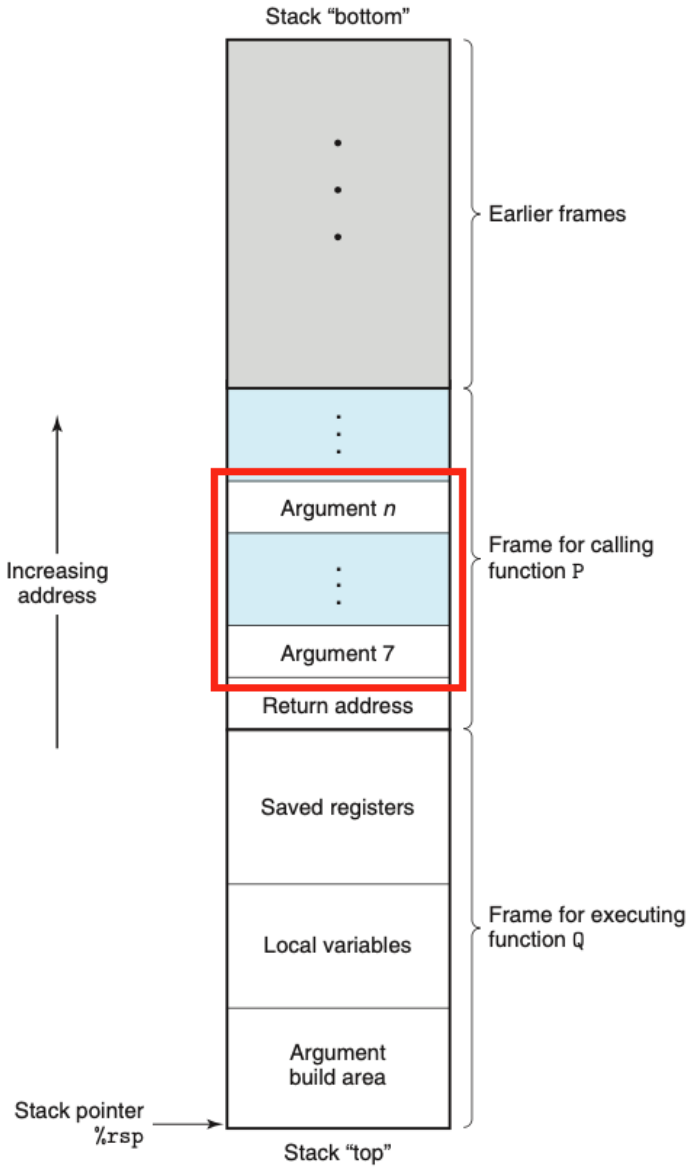
控制转移

```
     Disassembly of leaf(long y)
     y in %rdi
1    0000000000400540 <leaf>:
2      400540:  48 8d 47 02          lea    0x2(%rdi),%rax    L1: z+2
3      400544:  c3                   retq                     L2: Return

4    0000000000400545 <top>:
     Disassembly of top(long x)
     x in %rdi
5      400545:  48 83 ef 05          sub    $0x5,%rdi         T1: x-5
6      400549:  e8 f2 ff ff ff       callq  400540 <leaf>     T2: Call leaf(x-5)
7      40054e:  48 01 c0             add    %rax,%rax         T3: Double result
8      400551:  c3                   retq                     T4: Return

     . . .
       Call to top from function main
9      40055b:  e8 e5 ff ff ff       callq  400545 <top>      M1: Call top(100)
10     400560:  48 89 c2             mov    %rax,%rdx         M2: Resume
```

数据传送

- 前六个到寄存器 %rdi %rsi %rdx %rcx %r8 %r9
- 后面的存在栈中
- 返回值永远在 %rax

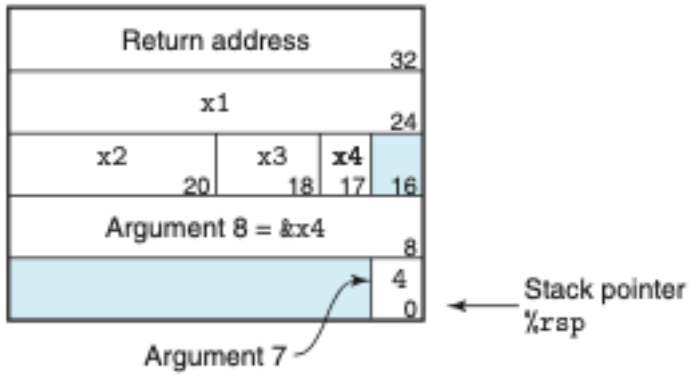
Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b



数据传送

(a) C code for calling function

```
long call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```



(b) Generated assembly code

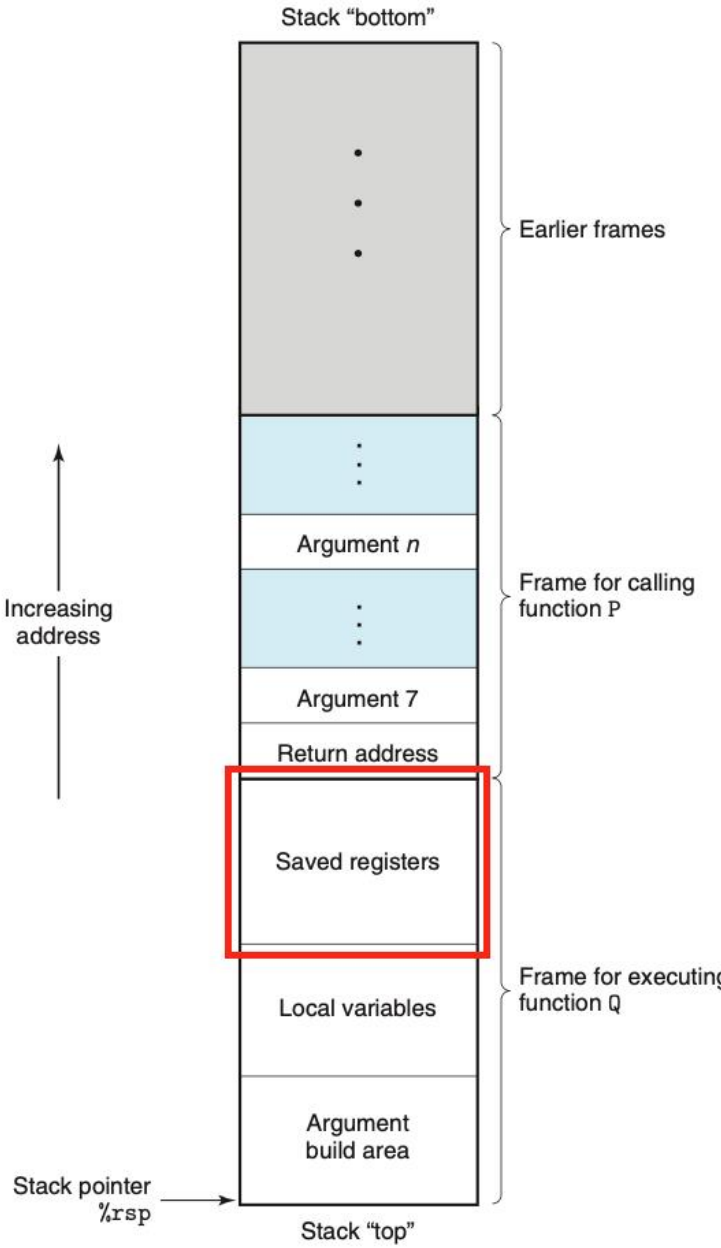
```
    long call_proc()
1   call_proc:
      Set up arguments to proc
2     subq    $32, %rsp          Allocate 32-byte stack frame
3     movq    $1, 24(%rsp)       Store 1 in &x1
4     movl    $2, 20(%rsp)       Store 2 in &x2
5     movw    $3, 18(%rsp)       Store 3 in &x3
6     movb    $4, 17(%rsp)       Store 4 in &x4
7     leaq    17(%rsp), %rax     Create &x4
8     movq    %rax, 8(%rsp)      Store &x4 as argument 8
9     movl    $4, (%rsp)         Store 4 as argument 7
10    leaq    18(%rsp), %r9      Pass &x3 as argument 6
11    movl    $3, %r8d           Pass 3 as argument 5
12    leaq    20(%rsp), %rcx     Pass &x2 as argument 4
13    movl    $2, %edx           Pass 2 as argument 3
14    leaq    24(%rsp), %rsi     Pass &x1 as argument 2
15    movl    $1, %edi           Pass 1 as argument 1
      Call proc
16    call    proc
```

栈指针对齐

- 在 `call` 之前 `%rsp` 对齐到 `16` 的倍数
- 将返回地址压栈后，进入新函数时 `%rsp` $\equiv 8\ (mod\ 16)$

Caller-saved / callee-saved

- `callee-saved`
  - `%rbx, %rbp, %r12-%r15`
- 其他（包括传参数的寄存器）：`caller-saved`



数据结构

鸣谢：20 级冉盛文

C 语言指针

- 指针有值，代表某个内存地址
- 指针有类型，表示以什么方式解释这块内存的数据
  - `void *` 表示通用指针
- 指针的类型和它的值无关
  - 可以通过指针类型（强制）转换使得同一个指针指向的内存区域被解读为不同类型
- `&lvalue` 用于取指针，`*ptr` 用于取值

指针运算

多重指针

函数指针

高维数组

变长数组

指针阅读

- `int (*(*vtable)[])();`
- 先定位变量名
- 之后向右阅读，直到右括号，确定其类型（是否是数组）
- 再向左阅读，直到左括号，确定其内容类型
- 之后跳出这个括号，再执行

- 右侧并列的括号可能是函数

struct

- 一个 struct 中不同子变量在内存中几乎相邻（对齐可能会影响）
- 可利用 struct 地址 + 偏移来访问子变量
- C 语言中定义 struct T 类型变量时需要写全 struct T
  - 除非使用 typedef

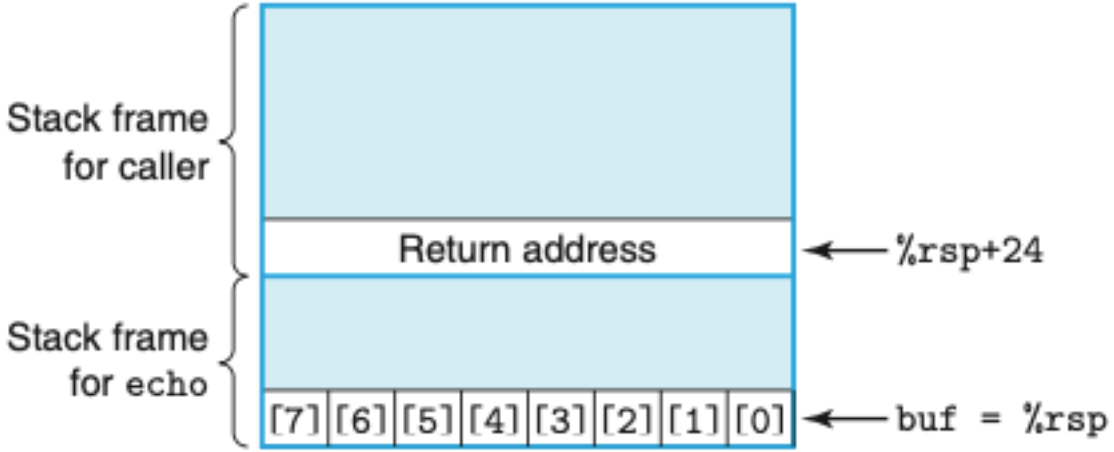
union

- 所有子变量在内存中起始位置相同
  - 通常情况下各个子变量是互斥的（不会同时被用到）
- union 的大小是所有子变量大小的最大值
- 可用于同位级表示的互转
  - **注意大小端！**
  - 还有一种方法是指针类型转换

struct 与 union 对齐

缓冲区溢出攻击

- 最基本的形式：覆盖返回地址



Characters typed	Additional corrupted state
0–7	None
9–23	Unused stack space
24–31	Return address
32+	Saved state in caller

防御

- 栈随机化 / 基址随机化（PIE）⊆ ASLR（地址空间随机化）
- Canary
  - https://en.wiktionary.org/wiki/canary_in_a_coal_mine
  - %fs:0x28
- NX（No-eXecute）