机器级编程 4
Machine-Level Programming IV

课 程 名 ：计算机系统

第 6 讲 （2025 年 5 月 7 日）

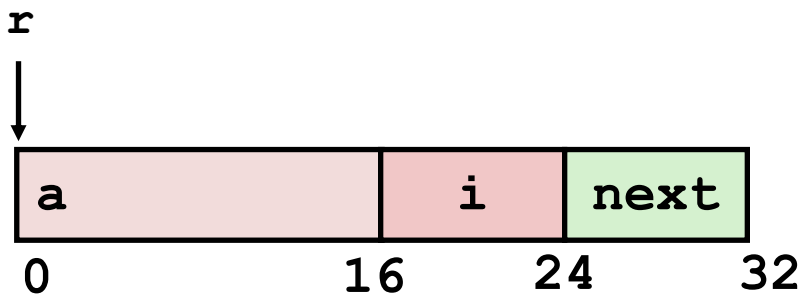
主 讲 人 ：杜海文

本节内容

- **结构体**
  - **内存分配Allocation**
  - **访问Access**
  - **对齐Alignment**
- **浮点数**
- **内存布局Memory Layout**
- **缓冲区溢出Buffer Overflow**
  - **漏洞Vulnerability**
  - **保护Protection**

结构体的表示

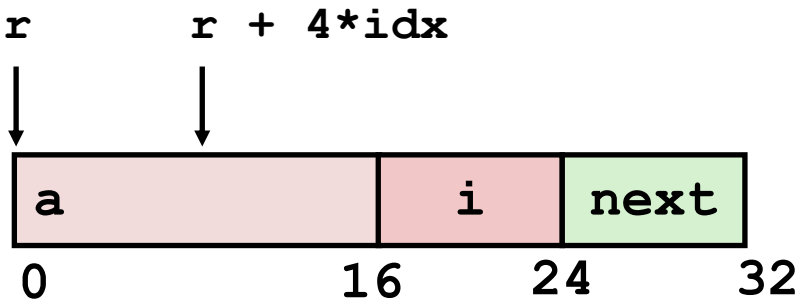
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 结构体用成块的内存表示
  - 块的大小要足以容纳其所有的成员变量
- 各个域的顺序要与声明保持一致
  - 不可更改顺序
  - 即便换序可以使内存分配更紧凑、更节省内存，也不可以
- 由编译器决定该块内存的总大小，以及各个域的位置
  - 机器级程序并不知道源代码当中有结构体的存在（无结构体概念）
  - 练习题 3.44

指向结构体成员的指针如何产生？

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 生成指向数组元素的指针
  - 结构体每个成员的偏移量在编译时确定
  - 结果为：r + 4*idx

```
int *get_ap(struct rec *r,
            size_t idx)
{
    return &r->a[idx];
}
```

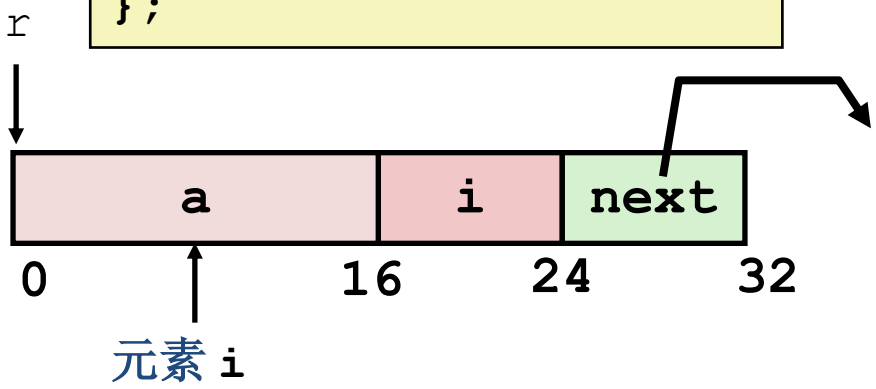
```
# r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

链表的遍历

- C 代码

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

```
void set_val
    (struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

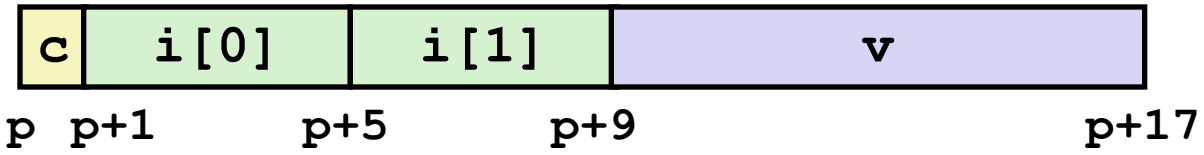


寄存器	取值
%rdi	r
%rsi	val

```
.L11:                                   # loop:
  movslq  16(%rdi), %rax                #    i = M[r+16]
  movl    %esi, (%rdi,%rax,4)           #    M[r+4*i] = val
  movq    24(%rdi), %rdi                #    r = M[r+24]
  testq   %rdi, %rdi                    #    测试 r 的值
  jne     .L11                          #    if != NULL 则循环
```

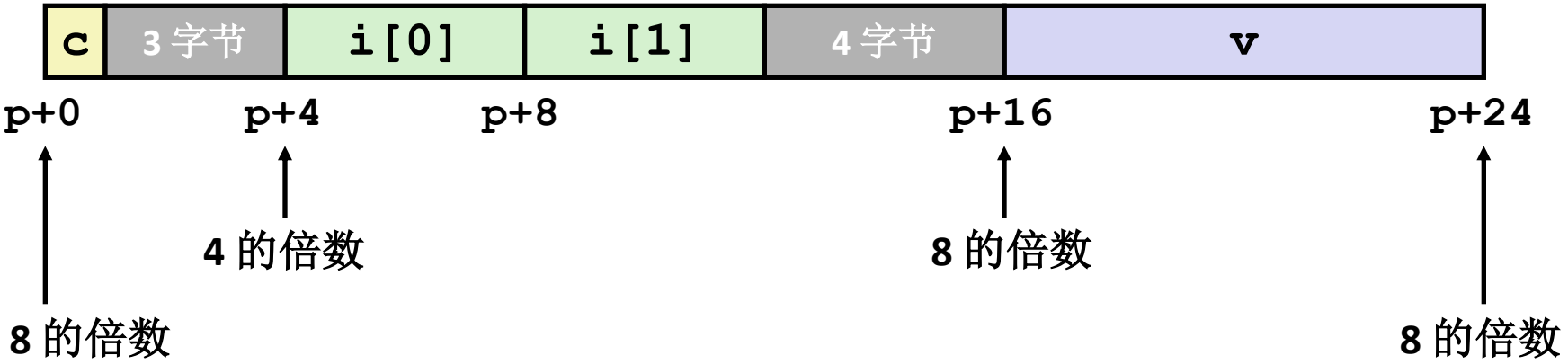
结构体的对齐存放

- 非对齐存放




```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- 对齐存放
  - 某一基本数据类型需占用 K 字节
  - 其起始地址须为 K 的倍数

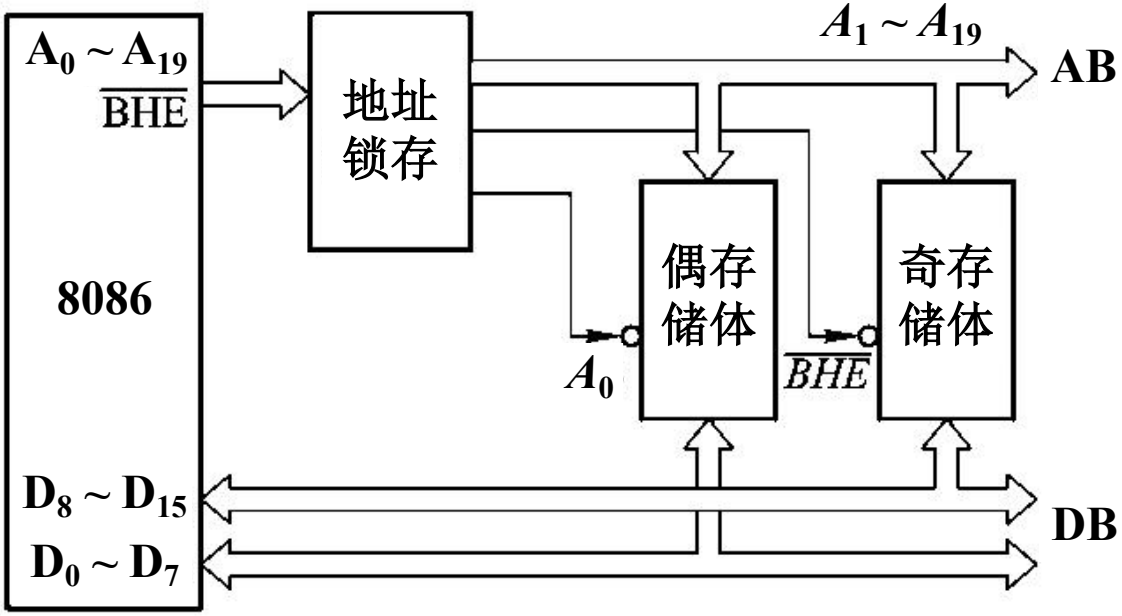


对齐原则

- 对齐存放
  - 某一基本数据类型需占用 K 字节
  - 其起始地址须为 K 的倍数
  - 有些机型强制要求对齐；x86-64 机建议对齐
- 数据对齐存放的原因
  - 主存在物理上是按 4 或 8 字节的倍数地址访问的（具体因不同机型而异）
    - 对跨越 4 字边界的数据进行存取，效率较低
    - 从虚拟内存的角度：当数据跨越页边界时，会引发更多的问题
- 编译器
  - 在结构体中插入间隙，以保证各成员变量的对齐存放

旁注：对齐存放性能高的原因

- 以 16 位 x86 系统为例（每存取周期可按 8/16 位访存）



- 内存总容量：1 MB
  - 地址总线：A_0 ~ A_{19}
  - 数据总线：D_0 ~ D_{15}
- 奇偶存储体：各 512 KB
  - 体地址引脚：A_0 ~ A_{18}
  - 体数据引脚：D_0 ~ D_7
  - 体选信号：负有效

- 例：
  - `movb 2, %al`
  - `movb 3, %al`
  - `movw 2, %ax`
  - `movw 3, %ax`

$\overline{BHE}$	A_0	传送的数据
L	L	两个字节
L	H	奇地址的高位字节
H	L	偶地址的低位字节
H	H	不传送

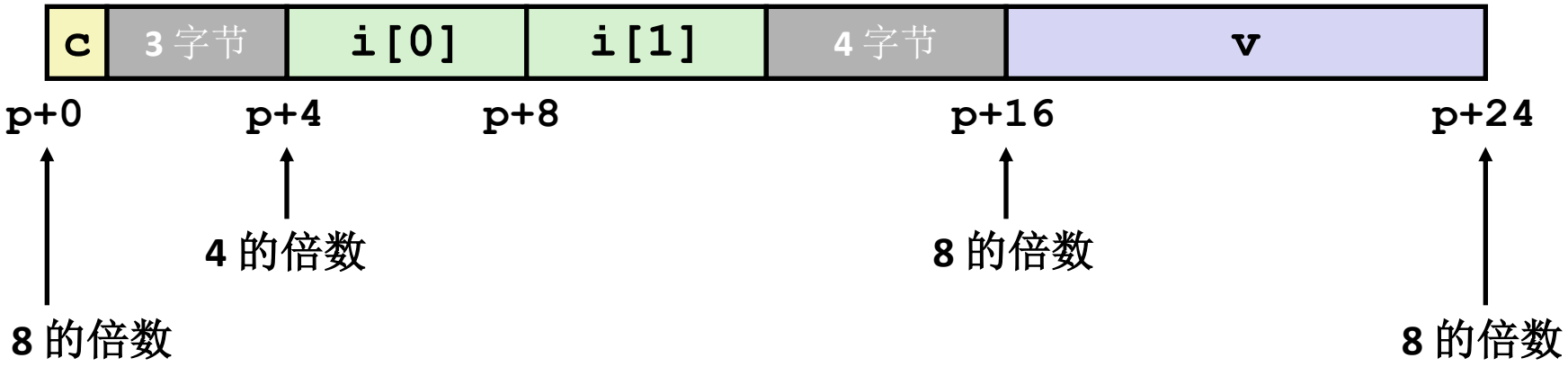
对齐的具体案例（x86-64）

- 1 字节：`char`, …
  - 对地址完全不加限制
- 2 字节：`short`, …
  - 地址最低位为 0_2
- 4 字节：`int`, `float`, …
  - 地址的最低 2 位为 00_2
- 8 字节：`double`, `long`, `char *`, …
  - 地址的最低 3 位为 000_2
- 16 字节：`long double` (GCC on Linux)
  - 地址的最低 4 位为 0000_2

满足结构体的对齐存放要求

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

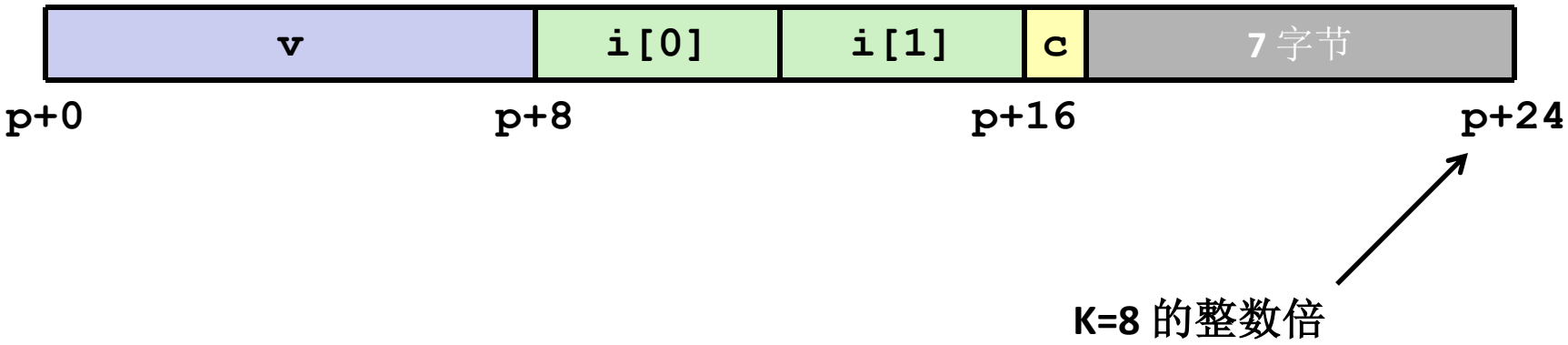
- 结构体内部
  - 须满足每个成员变量的对齐要求
- 对结构体整体的放置要求
  - 每一结构体要求按 *K* 字节对齐
    - *K* = 其内部各成员变量对齐要求的最大者
  - 首地址 & 结构体长度须为 *K* 的倍数
- 例如
  - *K* = 8，原因是存在 double 型成员



满足整体的对齐要求

- 设各成员对齐要求的最大值为 *K*
- 结构体整体须为 *K* 的倍数

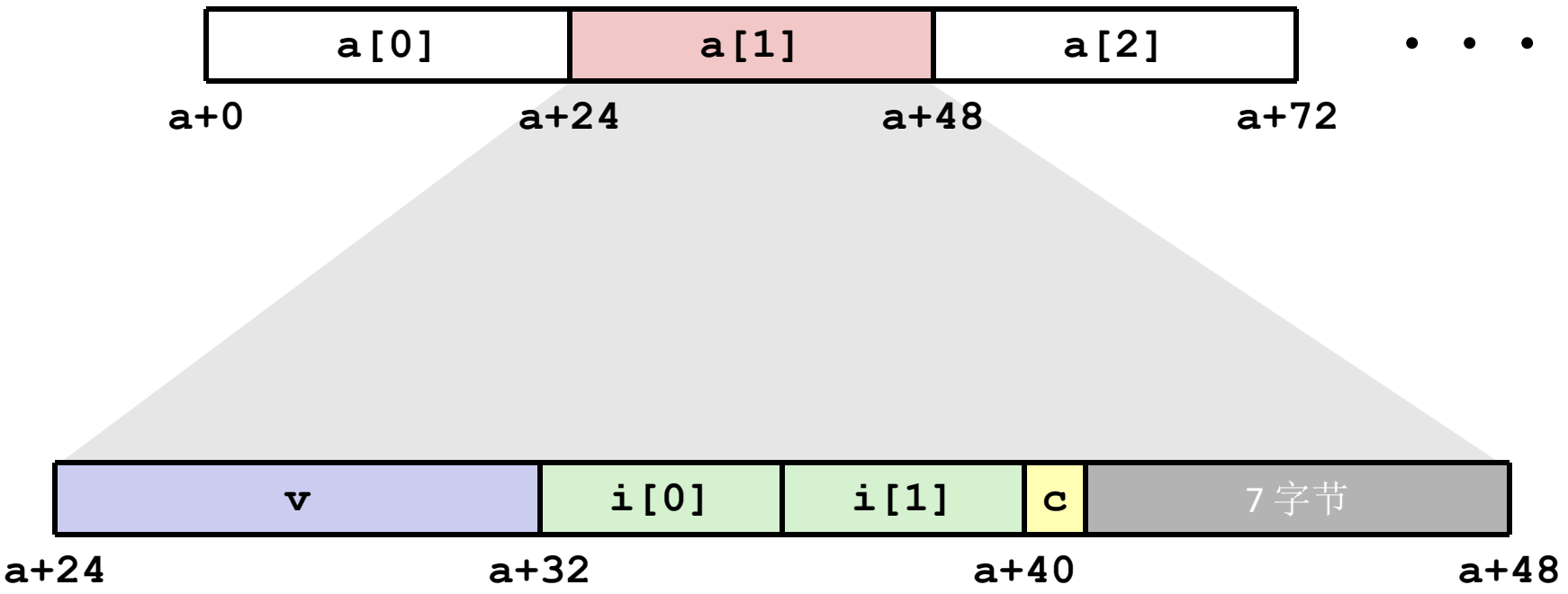
```
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```



结构体数组

- 结构体的整体长度为 *K* 的倍数
- 满足每个成员变量的对齐要求

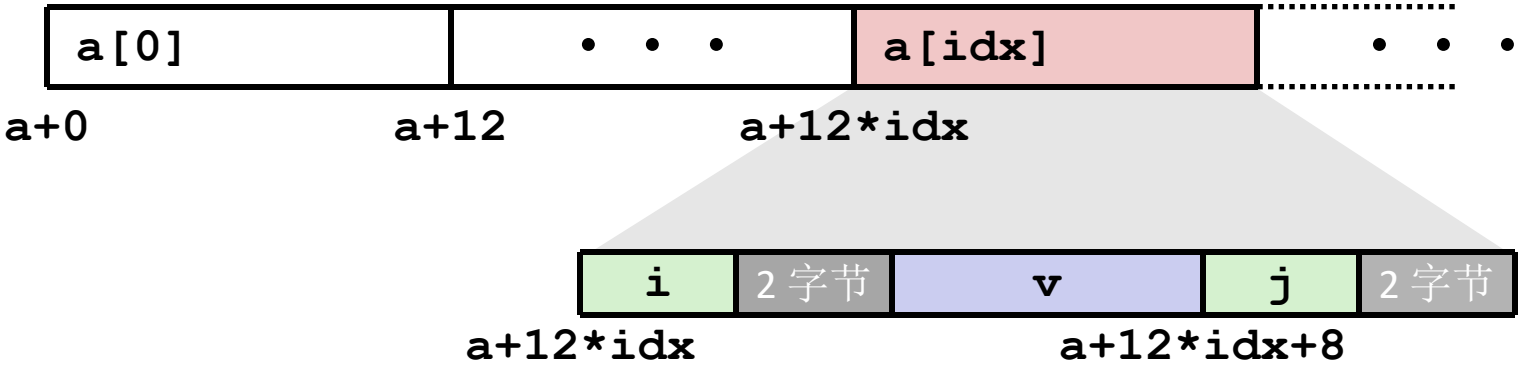
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



访问数组元素

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

- 计算数组内部偏移量：12 * idx
  - sizeof(S3)包含填充部分
- 成员变量 j 位于结构体内部偏移量 8 的位置
- 汇编器生成偏移量：a+8
  - 到链接环节再进一步定位



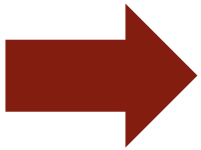

```
short get_j(int idx)
{
    return a[idx].j;
}
```

```
# %rdi = idx
  leaq (%rdi,%rdi,2),%rax    # 3*idx
  movzwl a+8(,%rax,4),%eax
```

节省空间

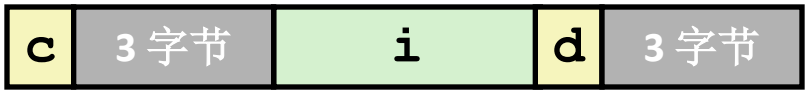
- 把较大的数据类型放在前面

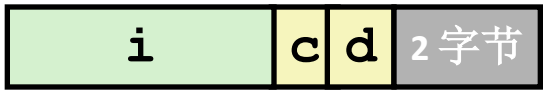
```
struct S4 {
    char c;
    int i;
    char d;
} *p;
```



```
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

- 结果（ $K = 4$ ）





本节内容

- 结构体
  - 内存分配Allocation
  - 访问Access
  - 对齐Alignment
- **浮点数Floating Point**
- 内存布局Memory Layout
- 缓冲区溢出Buffer Overflow
  - 漏洞Vulnerability
  - 保护Protection

背景知识

- 历史
  - x87 浮点运算协处理器
    - 与 IEEE 754 协同开发的硬件产品
    - 历史遗存，其编程模型相当粗陋
  - SSE 浮点指令
    - streaming SIMD extensions
    - 应用向量指令的特例
  - AVX 浮点指令
    - advanced vector extensions
    - 最新的版本
    - 与 SSE 类似
    - 课本中有详述

例：使用 SSE3 编程

XMM 寄存器

- 共 16 个，每个 16 字节：`%xmm0` ~ `%xmm15`
- 16 个单字节整数

- 8 个 16-bit 整数

- 4 个 32-bit 整数

- 4 个单精度浮点数

- 2 个双精度浮点数

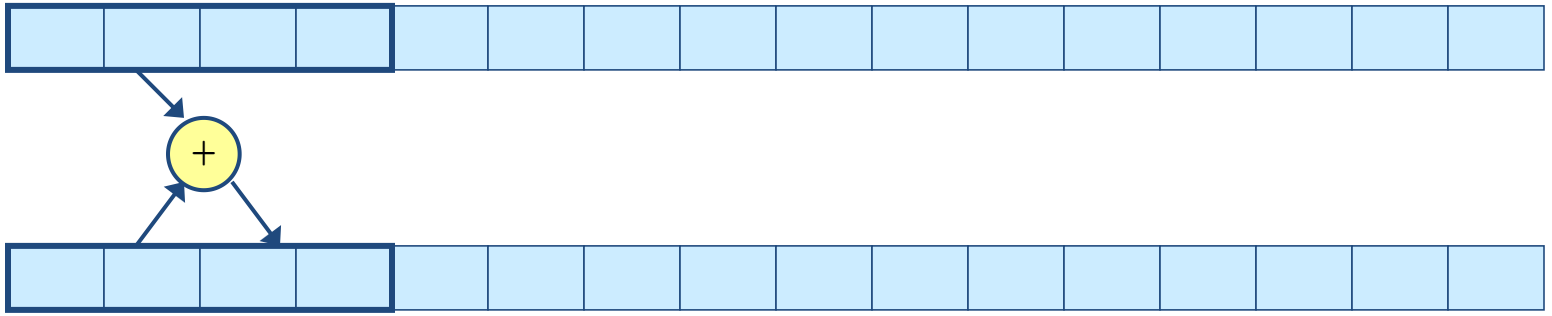
- 1 个单精度浮点数

- 1 个双精度浮点数

标量 & SIMD 操作

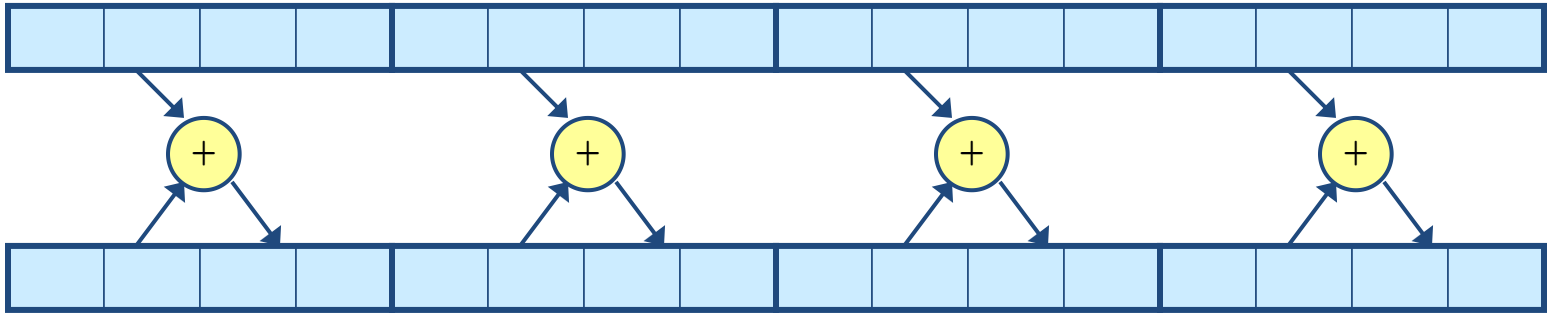
- 标量操作：单精度

`addss %xmm0,%xmm1`



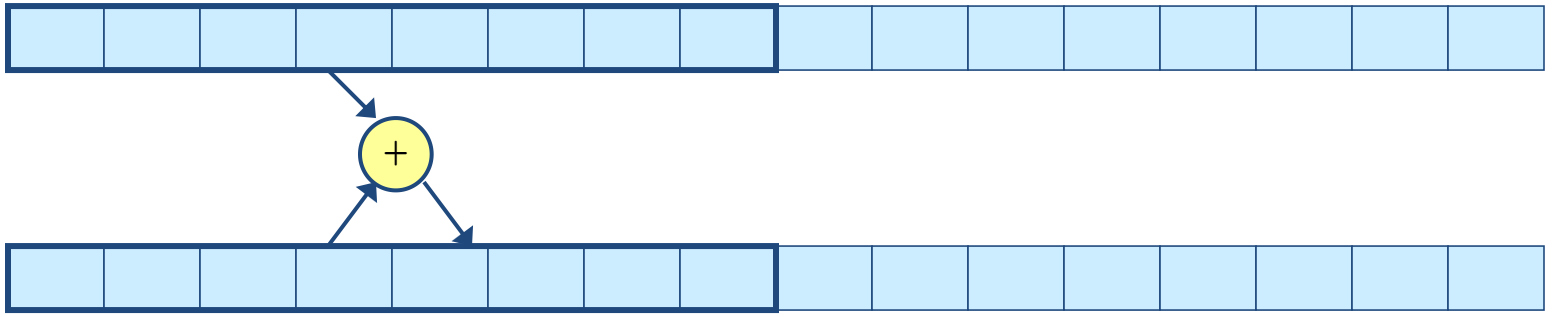
- SIMD 操作：单精度

`addps %xmm0,%xmm1`



- 标量操作：双精度

`addsd %xmm0,%xmm1`



浮点操作基础

- 参数通过 %xmm0、%xmm1、...传递
- 结果通过 %xmm0 返回
- 所有 XMM 寄存器均为 caller-saved

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double
y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss   %xmm1, %xmm0
  ret
```

```
 # x in %xmm0, y in %xmm1
  addsd   %xmm1, %xmm0
  ret
```

与浮点数有关的内存访问

- 整数（包括指针）参数通过普通的整数寄存器传递
- 浮点数值通过 XMM 寄存器传递
- 采用不同的 move 指令完成数据在 XMM 寄存器之间，以及内存和 XMM 寄存器之间传递

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd  %xmm0, %xmm1     # Copy v
movsd   (%rdi), %xmm0    # x = *p
addsd   %xmm0, %xmm1     # t = x + v
movsd   %xmm1, (%rdi)    # *p = t
ret
```

关于浮点数相关代码的其它内容

- 许许多多的浮点指令
  - 各种不同操作、不同数据格式
  - 不同指令集扩展（指令开头带 v 与不带 v）
- 浮点数的比较
  - 指令 `ucomiss` 和 `ucomisd`
  - 设置条件码 CF、ZF、PF
    - 整数指令结果的 LSB 中 1 的个数为偶数时 PF 为 1
    - 浮点比较指令的两个操作数中有 *NaN* 时，PF 为 1
  - 当 PF 为 1，结果为 unordered
- 常量的使用
  - 将 XMM0 寄存器清零：`xorpd %xmm0, %xmm0`
  - 其它常量需要从内存获得

小结

- 数组
  - 各元素被装入彼此相邻的内存区域
  - 通过对索引（下标）的算术运算，定位到各个元素
- 结构体
  - 各元素（成员、域）被装入整片内存区域
  - 对成员的访问要通过偏移量（由编译器决定）进行定位
  - 可能需要内部和（或）外部的填充，以保证数据的对齐存放
- 上述二者的组合
  - 结构体和数组可相互任意嵌套
- 浮点数
  - 数据通过 XMM 寄存器存放、运算

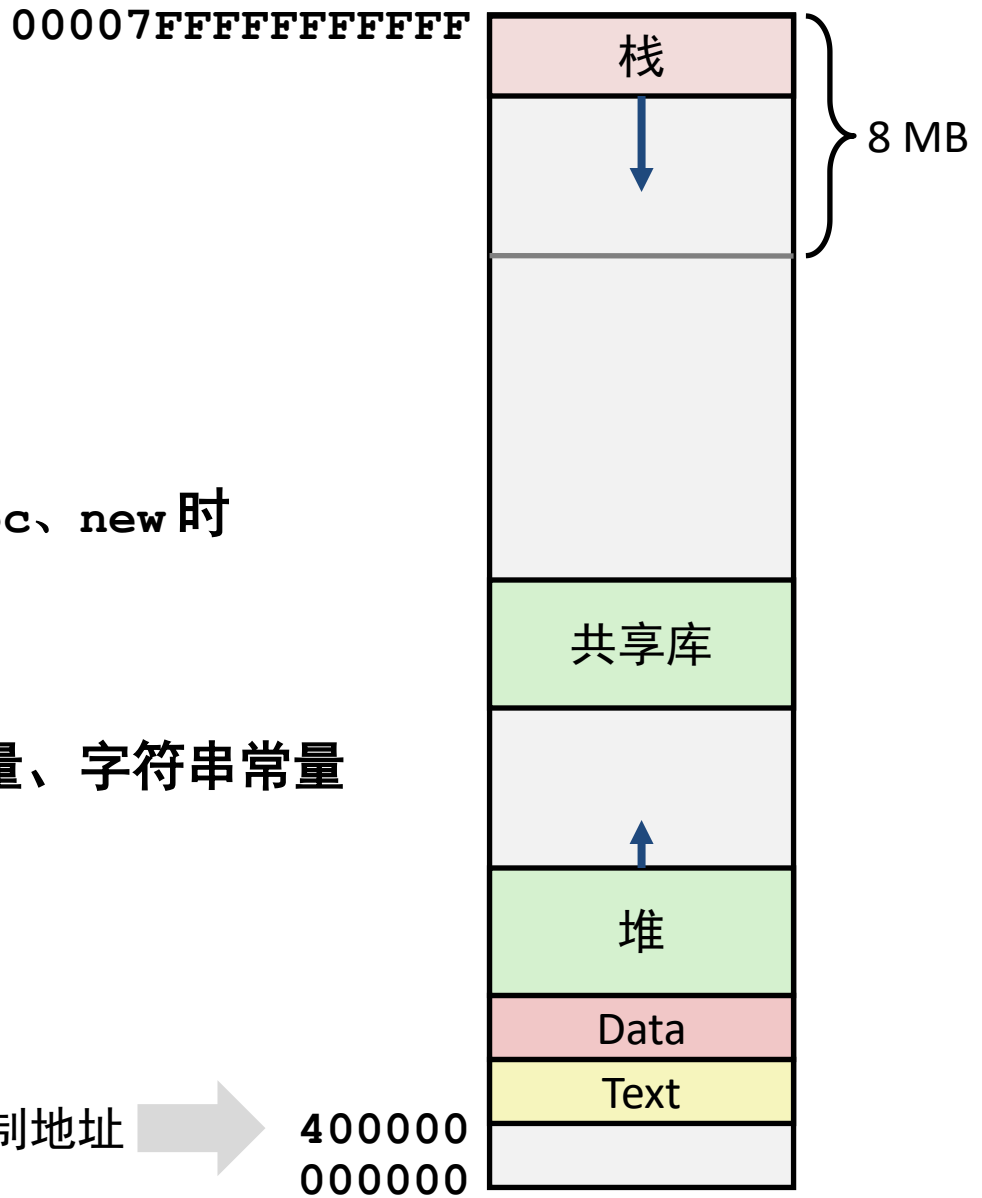
Today

- 结构体
  - 内存分配Allocation
  - 访问Access
  - 对齐Alignment
- 浮点数Floating Point
- **内存布局Memory Layout**
- 缓冲区溢出Buffer Overflow
  - 漏洞Vulnerability
  - 保护Protection

x86-64 Linux 内存布局

非等比例缩放

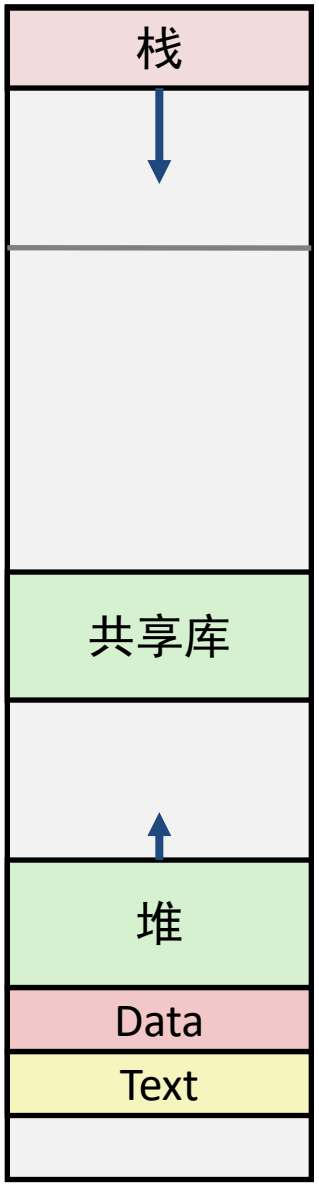
- 栈
  - 运行时栈（上限为 8 MB）
  - 例：局部变量
- 堆
  - 需要时动态分配
  - 例：在调用 `malloc`、`calloc`、`new` 时
- Data
  - 静态分配的数据
  - 例：全局变量、`static` 变量、字符串常量
- Text / 共享库
  - 可执行的机器指令
  - 只读



内存分配举例

非等比例缩放

```
char big_array[1L << 24];          /* 16 MB  */
char huge_array[1L << 31];         /*  2 GB  */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28);         /* 256 MB */
    p2 = malloc(1L << 8);          /* 256  B */
    p3 = malloc(1L << 32);         /*   4 GB */
    p4 = malloc(1L << 8);          /* 256  B */
/* 若干 printf 语句 ... */
}
```



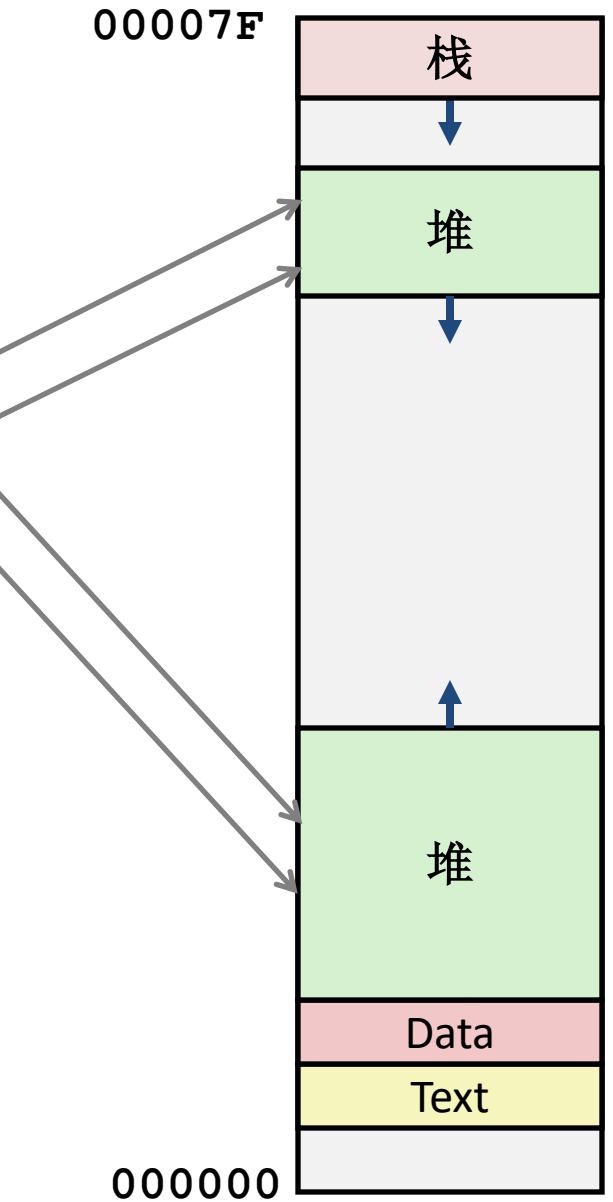
本程序涉及的所有内容在内存中如何分布？

x86-64 地址示例

地址范围：2^{47}

非等比例缩放

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



Today

- 结构体
  - 内存分配Allocation
  - 访问Access
  - 对齐Alignment
- 浮点数Floating Point
- 内存布局Memory Layout
- **缓冲区溢出Buffer Overflow**
  - **漏洞Vulnerability**
  - **保护Protection**

回顾：系统之坑 #3 ——一个访存 Bug

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i)
{
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824;      /* 可能产生越界 */
    return s.d;
}
```

```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.00000061035156
fun(4)  →  3.14
fun(6)  →  Segmentation fault
```

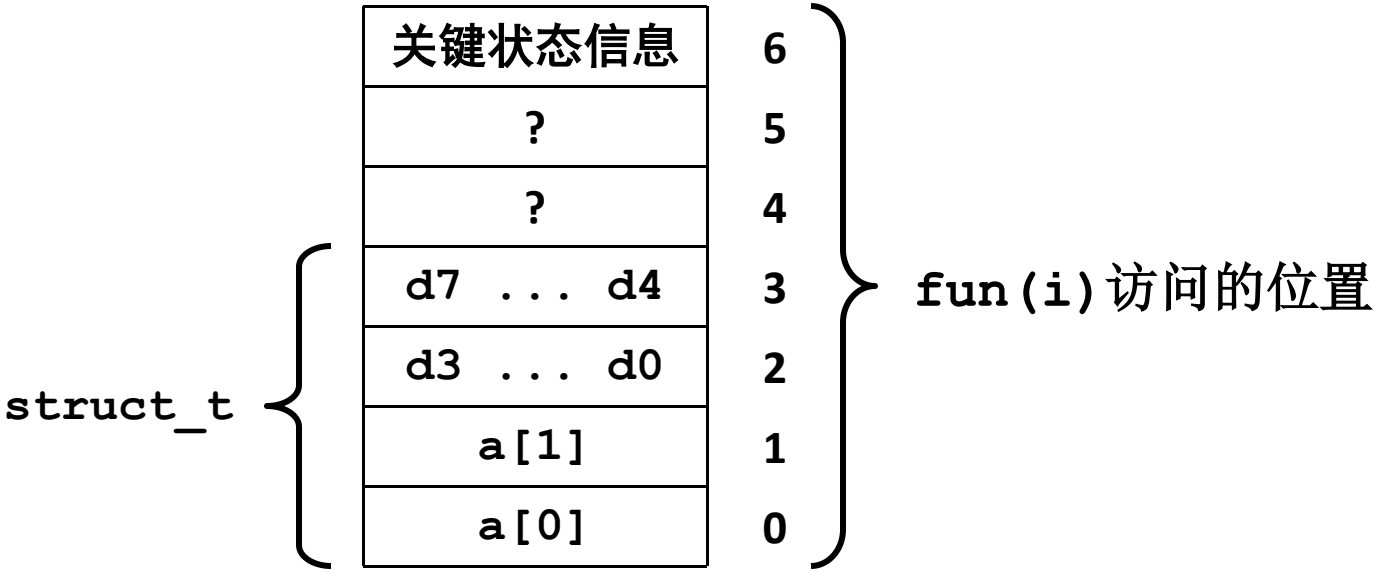
- 结果因具体系统而异

一个访存 bug

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  → 3.14
fun(1)  → 3.14
fun(2)  → 3.1399998664856
fun(3)  → 2.0000006103515 6
fun(4)  → 3.14
fun(6)  → Segmentation fault
```

解释：



此类问题关系重大

- 一般称之为缓冲区溢出buffer overflow
  - 发生在访存地址超出 为某一数组分配的内存大小 之时
- 为什么重要？
  - 导致安全漏洞的头号技术问题
    - 若将所有因素考虑在内，头号原因则是社会学层面的（用户专业素养不足）
- 最常见的形式
  - 字符串输入长度不加限制
  - 特别是针对 位于栈区的 有边界的 字符数组
    - 有时也称作 stack smashing

字符串库函数代码

- Unix 函数 gets() 的一种实现

```
/* 从标准输入获取字符串 */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;

    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - 无法对所读取字符的数量加以限制
- 其它库函数也存在类似问题
  - strcpy、strcat：复制任意长度的字符串
  - scanf、fscanf、sscanf：当采用 %s 格式时

存在漏洞的缓冲区代码

```
/* echo 一行字符 */
void echo()
{
    char buf[4];  // 太小！
    gets(buf);
    puts(buf);
}
```

← 到底多大算大？

```
void
call_echo()
{
    echo();
}
```

unix> *./bufdemo-nsp*
Type a string: *012345678901234567890123*
012345678901234567890123

unix> *./bufdemo-nsp*
Type a string: *0123456789012345678901234*
Segmentation Fault

对此缓冲区代码进行反汇编

echo():

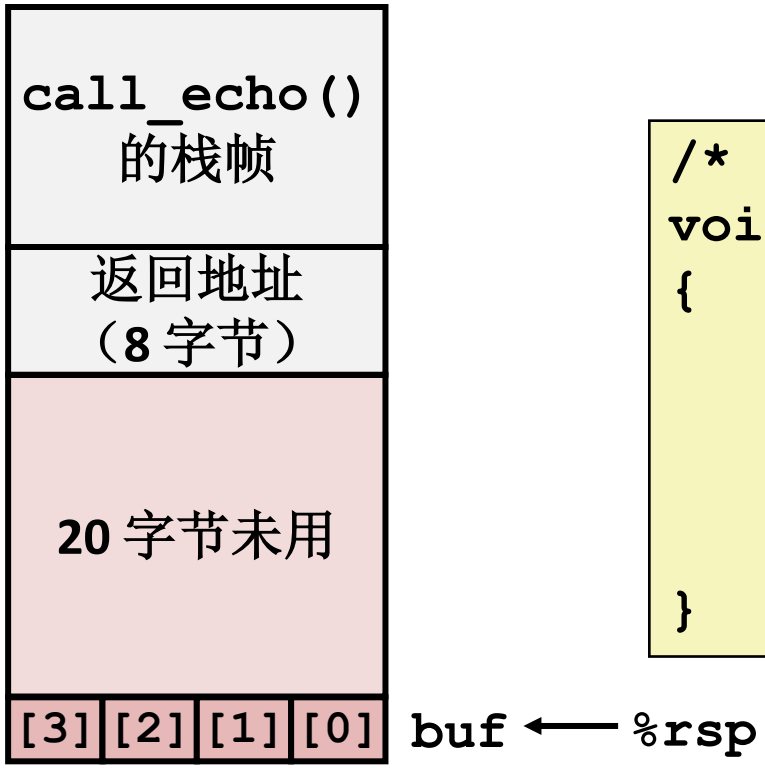
```
0000000004006cf <echo>:
 4006cf: 48 83 ec 18          sub    $0x18,%rsp
 4006d3: 48 89 e7             mov    %rsp,%rdi
 4006d6: e8 a5 ff ff ff       callq  400680 <gets>
 4006db: 48 89 e7             mov    %rsp,%rdi
 4006de: e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3: 48 83 c4 18          add    $0x18,%rsp
 4006e7: c3                   retq
```

call_echo():

```
4006e8:  48 83 ec 08          sub    $0x8,%rsp
 4006ec: b8 00 00 00 00       mov    $0x0,%eax
 4006f1: e8 d9 ff ff ff       callq  4006cf <echo>
 4006f6: 48 83 c4 08          add    $0x8,%rsp
 4006fa: c3                   retq
```

缓冲区溢出攻击

调用 gets()之前

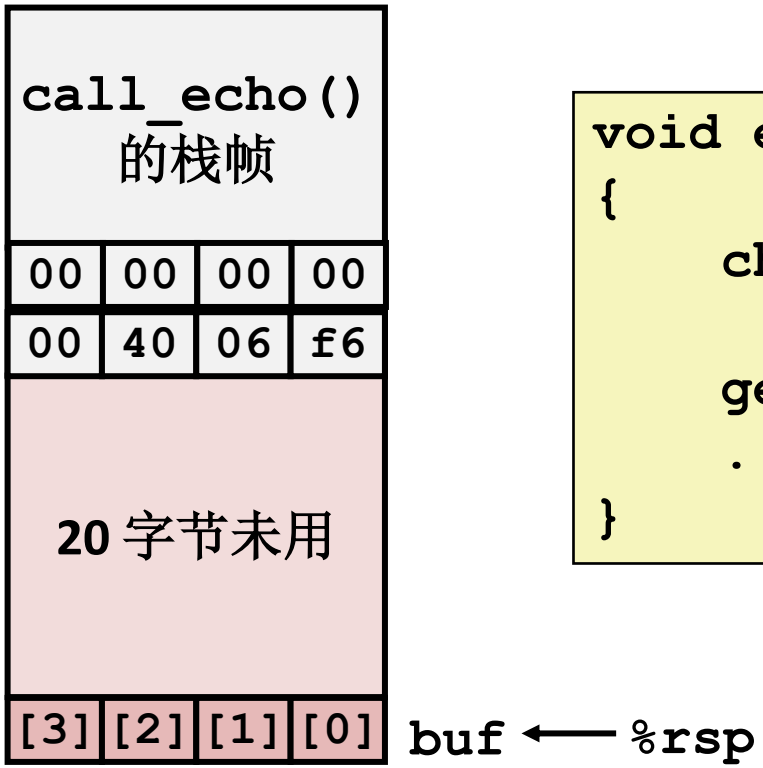



```
/* echo 一行字符 */
void echo()
{
    char buf[4];  /* 太小！ */

    gets(buf);
    puts(buf);
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

缓冲区溢出攻击举例

调用 gets()之前




```
void echo()
{
    char buf[4];

    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

```
call_echo():
   . . .
   4006f1: callq  4006cf <echo>
   4006f6: add    $0x8,%rsp
   . . .
```

缓冲区溢出堆栈示例 #1

调用 gets()之后

call_echo()的栈帧			
00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ⟵ %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo():

```
   . . .
   4006f1: callq  4006cf <echo>
   4006f6: add    $0x8,%rsp
   . . .
```

```
unix> ./bufdemo-nsp
Type a string: 0123456789012345678901 2
0123456789012345678901 2
```

缓冲区溢出，但并未侵占关键状态信息（断点）

缓冲区溢出堆栈示例 #2

调用 gets()之后

call_echo()
的栈帧

00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ⟵ %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo():

```
   . . .
   4006f1: callq  4006cf <echo>
   4006f6: add    $0x8,%rsp
   . . .
```

```
unix> ./bufdemo-nsp
Type a string: 012345678901234567890123
Segmentation Fault
```

缓冲区溢出，侵占了断点（返回地址）

缓冲区溢出堆栈示例 #3

调用 gets()之后

call_echo()的栈帧			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo():

```
   . . .
   4006f1: callq  4006cf <echo>
   4006f6: add    $0x8,%rsp
   . . .
```

```
unix> ./bufdemo-nsp
Type a string: 0123456789012345678901 23
0123456789012345678901 23
```

缓冲区溢出，侵占了断点，但程序看似运行正常！

对缓冲区溢出堆栈示例 #3 的解释

调用 gets()之后

call_echo()
的栈帧

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ⟵ %rsp

register_tm_clones():

```
  . . .
400600: mov    %rsp,%rbp
400603: mov    %rax,%rdx    # %rax值为0
400606: shr    $0x3f,%rdx   #   见call_echo
40060a: add    %rdx,%rax    #   的汇编代码
40060d: sar    %rax
400610: jne    400614
400612: pop    %rbp
400613: retq
```

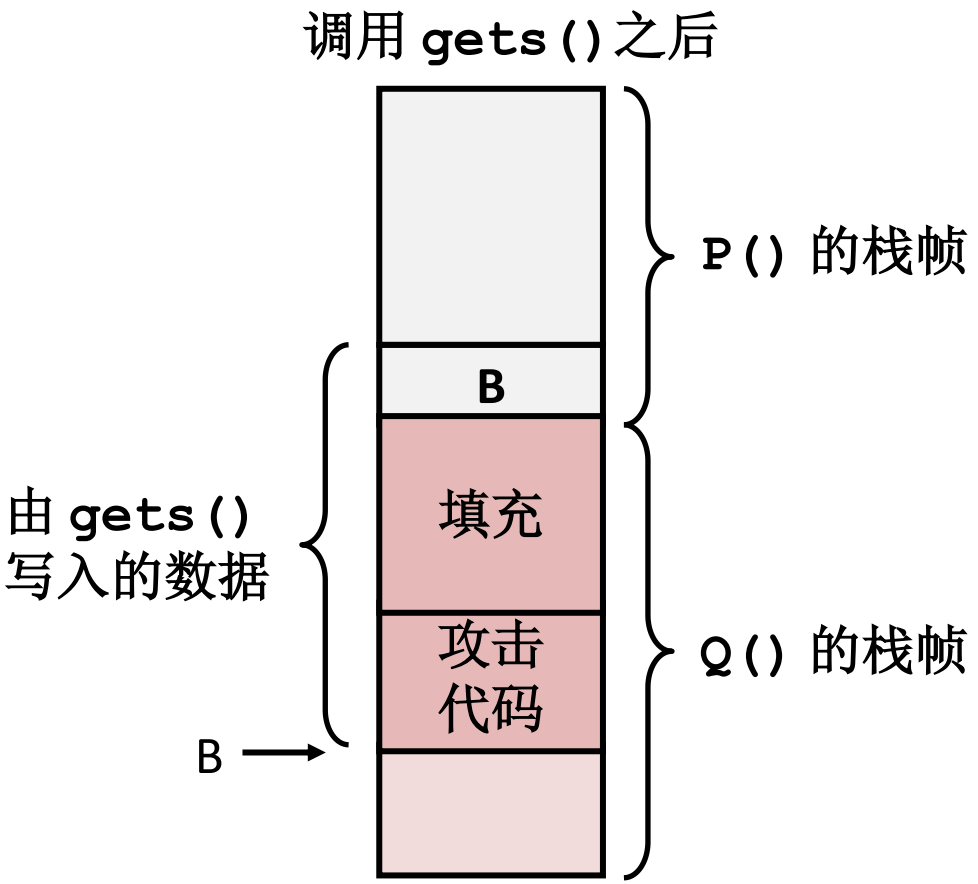
“返回”至不相关代码
执行了许多操作，但未改变关键的状态信息
最终执行 retq，返回到 main

代码注入攻击

```
void P()
{
    Q();
    ...
}
```

返回地址 A

```
int Q()
{
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```



- 输入字符串为可执行代码的字符表示形式
- 将返回地址 A 改写为缓冲区 B 的地址
- 当 Q() 执行到 ret 指令时，将会转至攻击代码并开始执行

人类对缓冲区溢出的利用

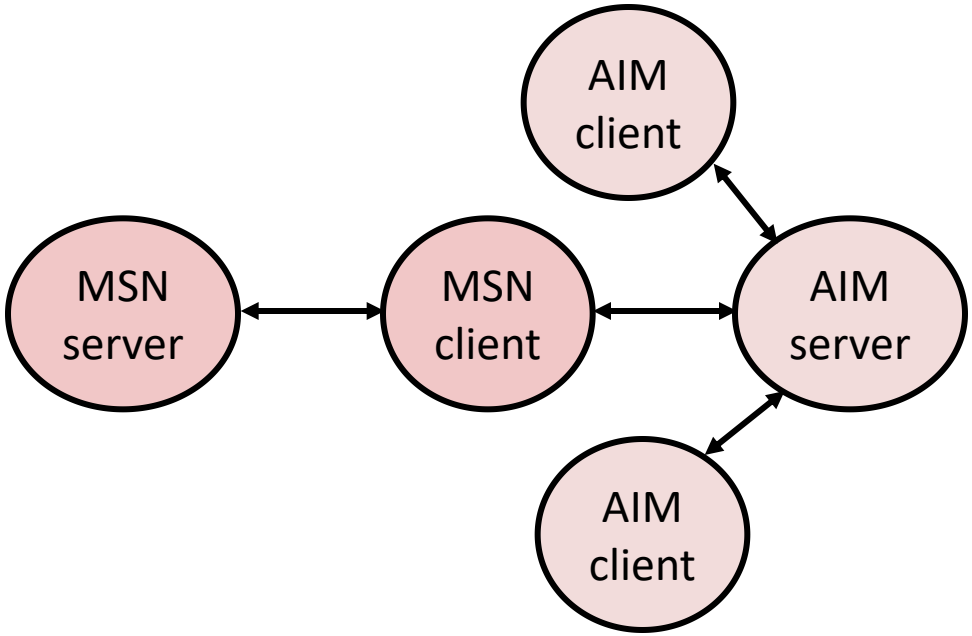
- 目标机中的缓冲区溢出漏洞可以让远程计算机在其上执行任意代码
- 在现实程序中大量存在
  - 程序员不停地犯同样的错误 ☹
  - 新近的措施使得这类攻击难度远高于此前
- 几十年间发生的案例
  - “互联网蠕虫”事件（1988）
  - 即时通讯战争IM wars（1999）
  - 针对任天堂 Wii 机的 Twilight Hack（00 年代）
  - …（数不胜数）
- attack lab实验用到此类技巧
  - 希望能让学习者有意识地在自己的程序中避免此类漏洞！！

案例 1：莫里斯蠕虫事件（1988）

- 利用若干漏洞实现传播
  - 早期的 finger 服务（fingerd）使用 `gets()` 读取客户发出的参数：
    - finger droh@cs.cmu.edu
  - “蠕虫”通过发送以下虚假参数攻击 fingerd 服务程序
    - finger "exploit-code padding new-return-address"
    - 攻击代码：在与攻击机有直接 TCP 连接的受害机上执行根 shell
- 每侵入一台机器，则扫描其它目标机，开始新一轮攻击
  - 几小时内即波及 6000 台机器（占当时互联网的 10% 哈哈哈）
    - 见《ACM 通讯》（*Comm. of the ACM*）1989 年 6 月文章
  - 作者 Robert Morris 被判 3 年缓刑，并因此负债 15 万美元
  - 计算机应急响应小组 CERT 由此成立，位于 CMU
  - 我国对应机构为国家互联网应急中心（CNCERT/CC）

案例 2：即时通讯战争

- 1999 年 7 月
  - 微软公司发布了 MSN Messenger（一款即时通讯系统）
  - Messenger 客户端能够访问当时流行的 AIM（AOL Instant Messaging Service）服务器



即时通讯战争（续）

- 1999 年 8 月
  - 不知为何，Messenger 客户端不能访问 AIM 服务器了
  - 一场即时通讯战争在微软和美国在线 AOL 之间打响：
    - AOL 对服务器作出更改，禁止 Messenger 客户使用
    - 微软随即也对客户端进行更改，令 AOL 的上述改变失效
    - 至少发生了 13 次类似的冲突
  - 到底发生了什么？
    - AOL 在自己的 AIM 客户端程序中发现了一个缓冲区溢出 bug
    - 他们利用该漏洞对微软进行甄别与封锁：攻击代码向服务器返回一个 4 字节的签名（来自 AIM 客户端中的某一位置）
    - 微软相应作出改变，以使自己的程序符合该签名，然后 AOL 又再度更换签名的位置

```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you
might find interesting because you are an Internet security expert with
experience in this area. I have also tried to contact AOL but received
no response.

I am a developer who has been working on a revolutionary new instant
messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself
this might not be the end of the world, as MS surely has had its share.
But AOL is now *exploiting their own buffer overrun bug* to help in
its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you
can use this information to help inform people that behind AOL's
friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

人们后来发现这封邮件
竟然是从微软内部流出的！

旁注：蠕虫与病毒

- 蠕虫：具有如下特征的程序
  - 可以自主运行
  - 能够将其自身完整地传播到其它计算机中
- 病毒：具有如下特征的代码
  - 能够将自身加入到其它程序内部（包括系统程序）
  - 无法自主运行
  - （口头上也可泛指蠕虫）
- 二者均以传播、制造混乱为目的

如何应对缓冲区溢出攻击

- 避免缓冲区溢出漏洞（应用层面）
- 实施系统级防护（OS 层面）
- 在编译器中采用“栈金丝雀”技术（编译层面）

- 下面逐一展开介绍…

1. 在代码中避免留下缓冲区溢出漏洞（!）

```
/* echo 一行字符 */
void echo()
{
    char buf[4];  /* 太小! */

    fgets(buf, 4, stdin);
    puts(buf);
}
```

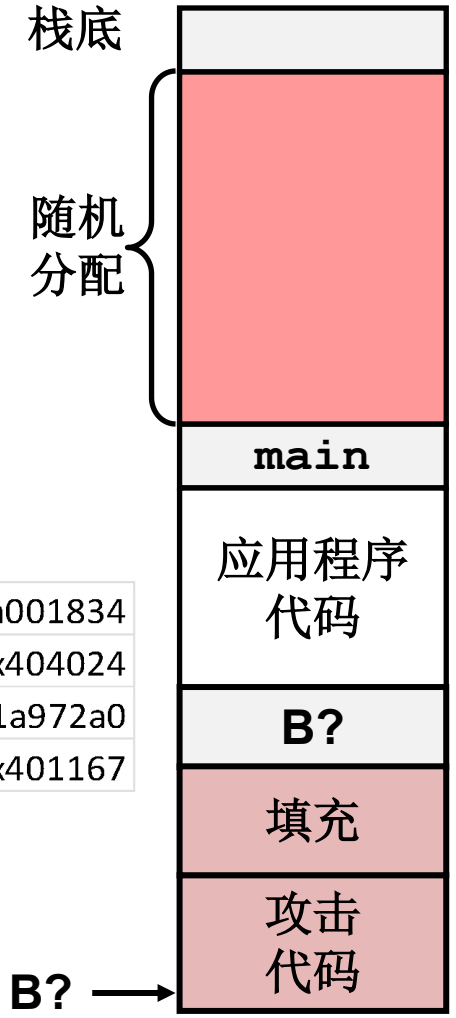
- 比如：调用能够限制字符串长度的库函数
  - 用 fgets 代替 gets
  - 用 strncpy 代替 strcpy
  - 不要在 scanf 中使用 %s 获取输入
    - 用 fgets 读取字符串
    - 或者用 %ns，注意整数 n 的取值大小要合适

2. 利用系统提供的防护

- 栈偏移随机化
  - 在程序执行伊始，分配一大小随机的栈空间
  - 整个程序所有的栈区存储单元地址被整体移动
  - 使黑客难以预测注入代码的起始地址
  - 例 1：分别执行 5 次内存分配代码

局部	0x7ffc9cb17214	0x7fff9580ce64	0x7fffacae5e74	0x7ffe61e4a3f4	0x7ffc4a001834
全局	0x404024	0x404024	0x404024	0x404024	0x404024
堆	0x23e92a0	0xd302a0	0x20f12a0	0xfa92a0	0x1a972a0
代码	0x401167	0x401167	0x401167	0x401167	0x401167

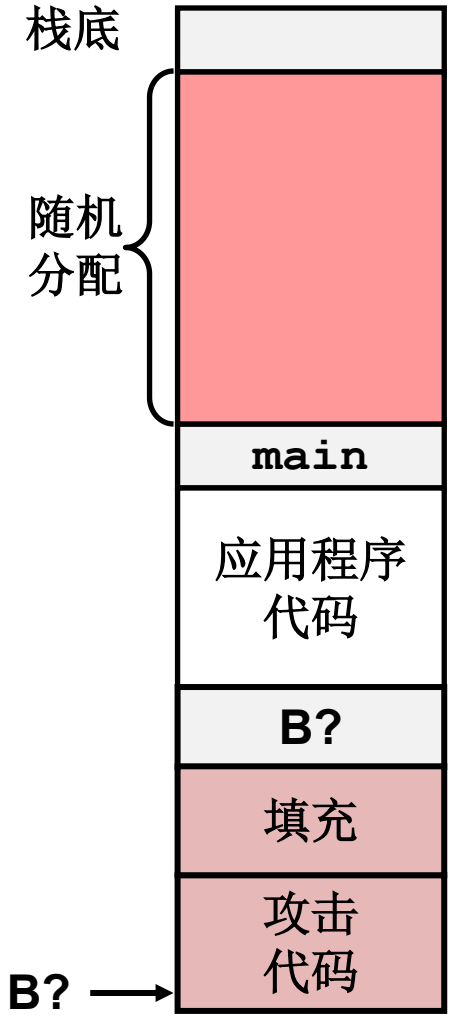
  - 堆栈每次都被重新定位
  - （源码见 chap3/adrs.c）
  - （全局变量地址、代码地址见第 7 章）
  - （堆区地址见第 9 章）



2. 利用系统提供的防护（续）

- **栈偏移随机化**
  - **例 2：栈区内存随机分配范围**
    - **C 源程序 stackAdr.c 见：3.10.4 → 栈随机化**
    - **执行该程序 10000 遍：**
    - **编辑脚本程序 stackAdrs.sh**

```
#!/bin/bash

i=0
while (( i++ < 10000 )); do
    ./stackAdr
done
```

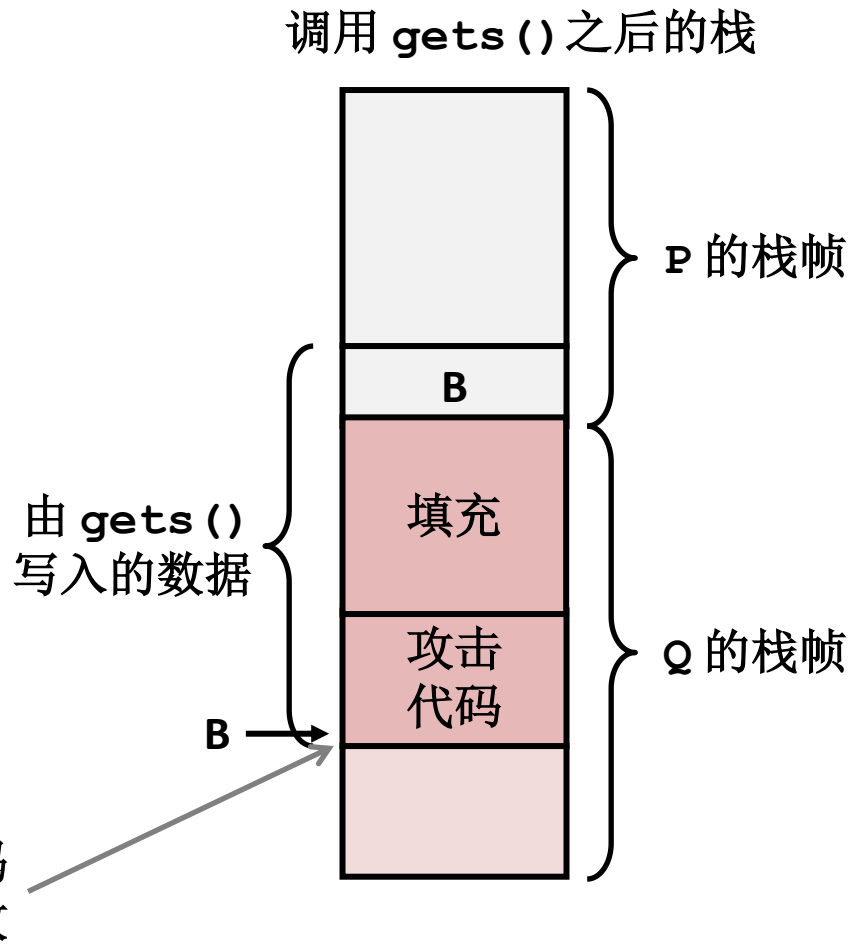


- 执行该脚本，查看 local 的地址变化范围

```
openEuler> chmod u+x  ./stackAdrs.sh        # 赋权
openEuler> ./stackAdrs.sh | sort –d –k 3 ¥   # 管道
> > stackAdrs.txt                  # 输出重定向
```

2. 利用系统提供的防护（续）

- 令代码不可执行
  - 传统 x86 机可将内存某区域标记为只读或者可写
    - 只要可读便可执行
  - x86-64 机专门增加了对执行权限的控制
  - 栈区被标记为：不可执行



3. 利用栈金丝雀技术

- 基本思想
  - 紧邻着缓冲区边界之外存储一个特殊数值（金丝雀值canary）
  - 退出函数之前检测该值是否被污染
- GCC 实现
  - -fstack-protector
  - 现已成为默认选项
  - 如欲关闭，加 -fno-stack-protector（见此前 bufdemo-nsp）

```
unix> ./bufdemo-sp
Type a string: 0123456
0123456
```

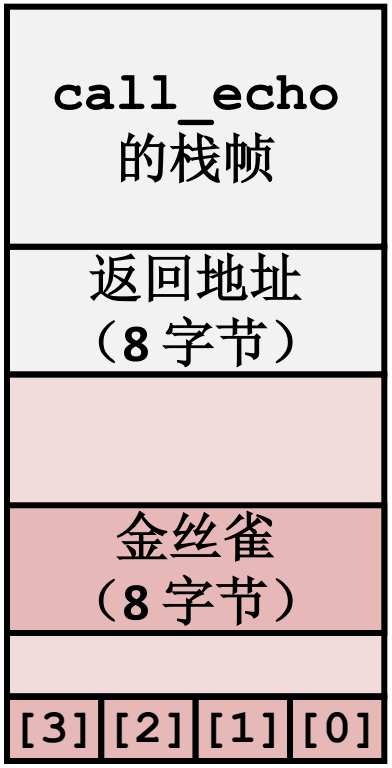
```
unix> ./bufdemo-sp
Type a string: 01234567
*** stack smashing detected ***
```

受保护的缓冲区代码反汇编

echo:

```
40072f: sub    $0x18,%rsp
400733: mov    %fs:0x28,%rax
40073c: mov    %rax,0x8(%rsp)
400741: xor    %eax,%eax
400743: mov    %rsp,%rdi
400746: callq  4006e0 <gets>
40074b: mov    %rsp,%rdi
40074e: callq  400570 <puts@plt>
400753: mov    0x8(%rsp),%rax
400758: xor    %fs:0x28,%rax
400761: je     400768 <echo+0x39>
400763: callq  400580 <__stack_chk_fail@plt>
400768: add    $0x18,%rsp
40076c: retq
```

安放金丝雀

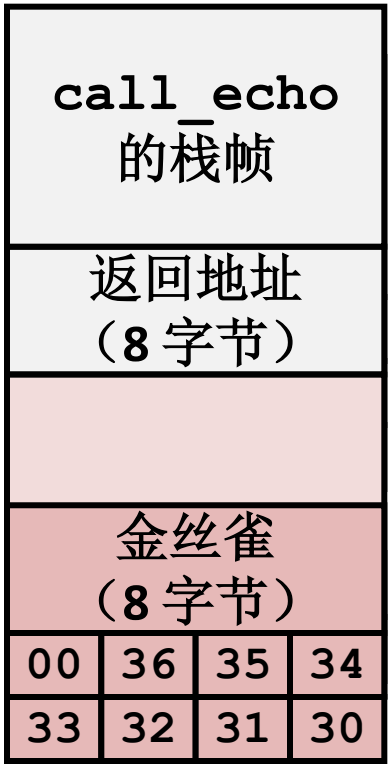


调用 gets 之前

```
/* echo 一行字符 */
void echo()
{
    char buf[4];     /* 太小！ */

    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax    # 取金丝雀值
    movq    %rax, 8(%rsp)   # 将其存入栈中
    xorl    %eax, %eax      # %eax清零
    . . .
```

检测金丝雀值



调用 gets 之后

```
/* echo 一行字符 */
void echo()
{
    char buf[4];  /* 太小！ */

    gets(buf);
    puts(buf);
}
```

输入：*0123456*

```
echo:
    . . .
    movq    8(%rsp), %rax       # 从栈中取回该值
    xorq    %fs:40, %rax        # 与原金丝雀作比较
    je      .L6                 # 若相等则通过
    call    __stack_chk_fail    # 检测失败
.L6:        . . .
```

ROP 攻击（Return-Oriented Programming）

- （给黑客）带来的挑战
  - 由于采用栈随机化技术，导致难以预测缓冲区地址
  - 将栈区标记为不可执行，导致难以注入二进制代码
- 替代策略
  - 使用已有代码
    - 例：stdlib 中的库函数代码
  - 将片段连成一体，实现想要的结果
  - 依旧无法克服栈金丝雀
- 由若干现有程序小段gadget“缝合”出完整的攻击代码
  - 以 `ret` 结束的指令序列
    - 对应二进制码为单字节的 `0xc3`
  - 代码位置每次执行都相同
  - 代码是可执行的

Gadget 示例 #1

```
long ab_plus_c(long a, long b, long c)
{
    return a*b + c;
}
```

```
0000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

rax ← rdi + rdx

Gadget 地址 = 0x4004d4

- **借用已有函数的结尾部分**

Gadget 示例 #2

```
void setval(unsigned *p)
{
    *p = 3347663060u;       /* 0xc78948d4 */
}
```

编码指令 `movq %rax, %rdi`

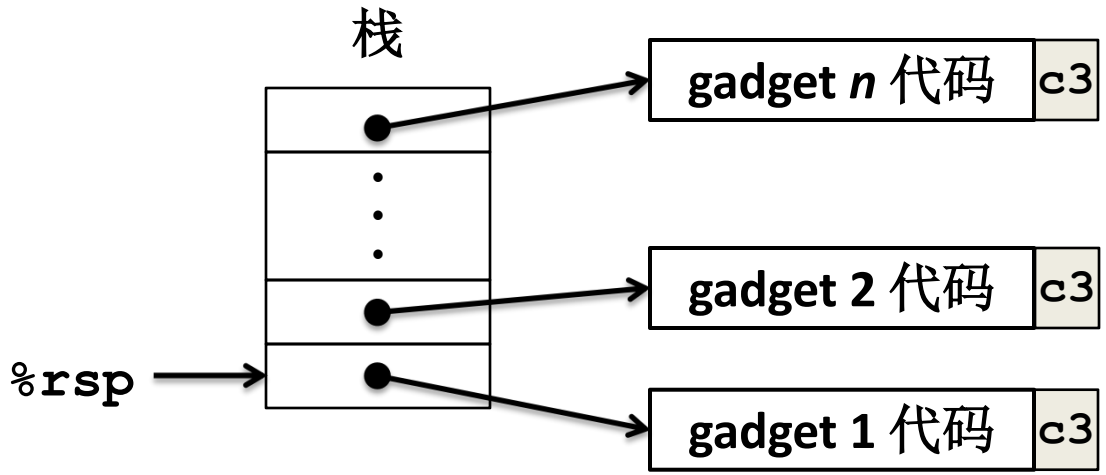
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl   $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

rdi ← rax

gadget 地址 = 0x4004dc

- 以不同方式重新解析字节码

ROP 执行



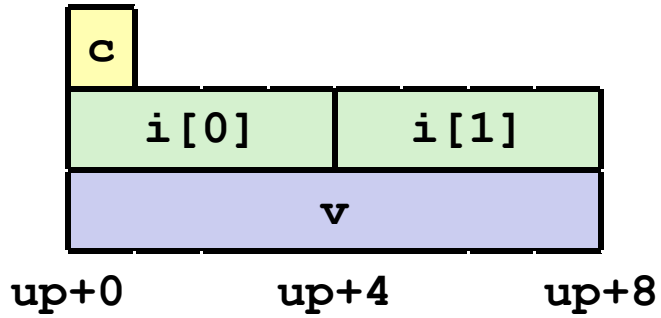
- 以 `ret` 指令触发
  - 将会开始执行 gadget 1
- 每个 gadget 结尾的 `ret` 将会启动下一个 gadget 的执行

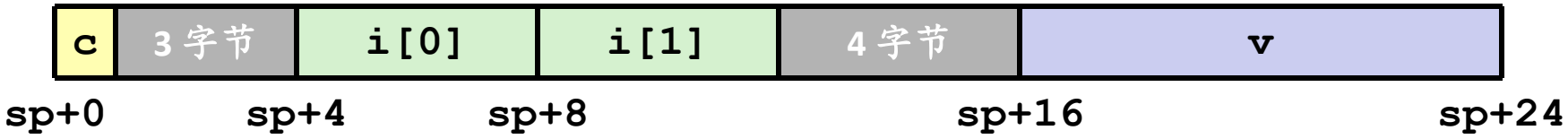
共用体union的内存分配

- 按最大的成员（域）进行分配
- 一次只能使用一个域（常用于互斥的场合，见课本 3.9.2 节）

```
union U1 {
    char c;
    int i[2];
    double v;
} *up;
```

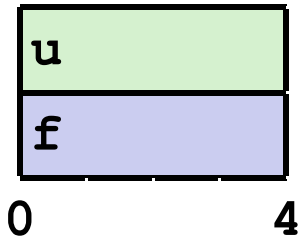
```
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```





使用共用体访问同一位串

```
typedef union {
    float f;
    unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
    bit_float_t arg;

    arg.u = u;
    return arg.f;
}
```

```
unsigned float2bit(float f)
{
    bit_float_t arg;

    arg.f = f;
    return arg.u;
}
```

是否等同于(float) u?

是否等同于 (unsigned) f?

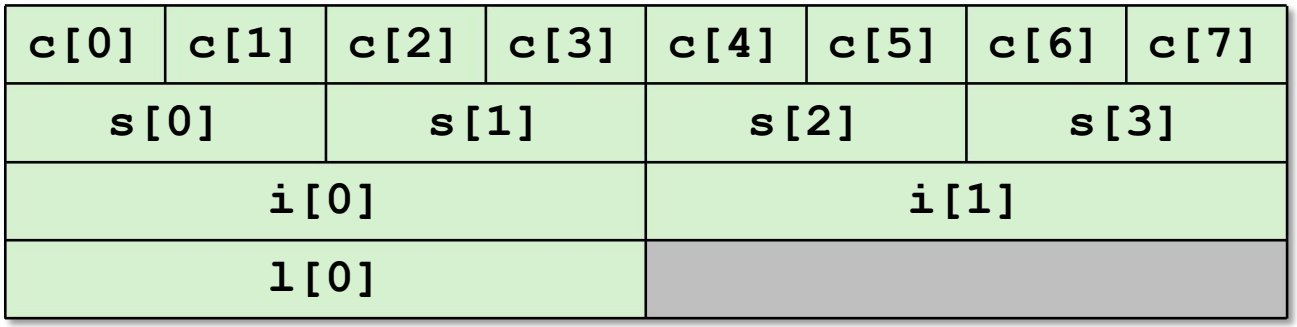
字节序回顾

- 基本思想
  - 16位、32位、64位数据，存于内存 2、4、8 个连续字节单元中
  - 哪一个作为最高（低）字节？
  - 数据在不同机器之间传输时，可能造成错误
- 大端序
  - 最高字节（MSB）置于最低地址
  - Sparc 机器
- 小端序
  - 最低字节（LSB）置于最低地址
  - Intel x86、ARM Android、iOS
- 双端序
  - 可被配置成两种字节序中的一种
  - ARM 硬件

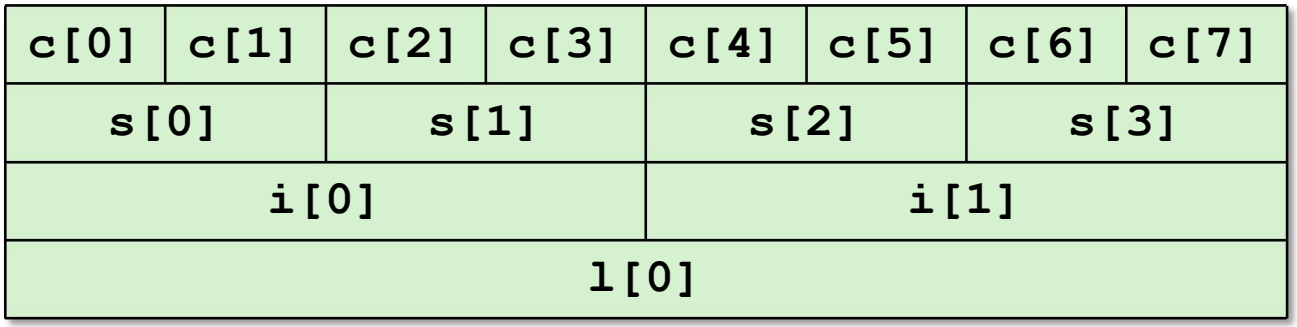
字节序举例

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

32-bit



64-bit

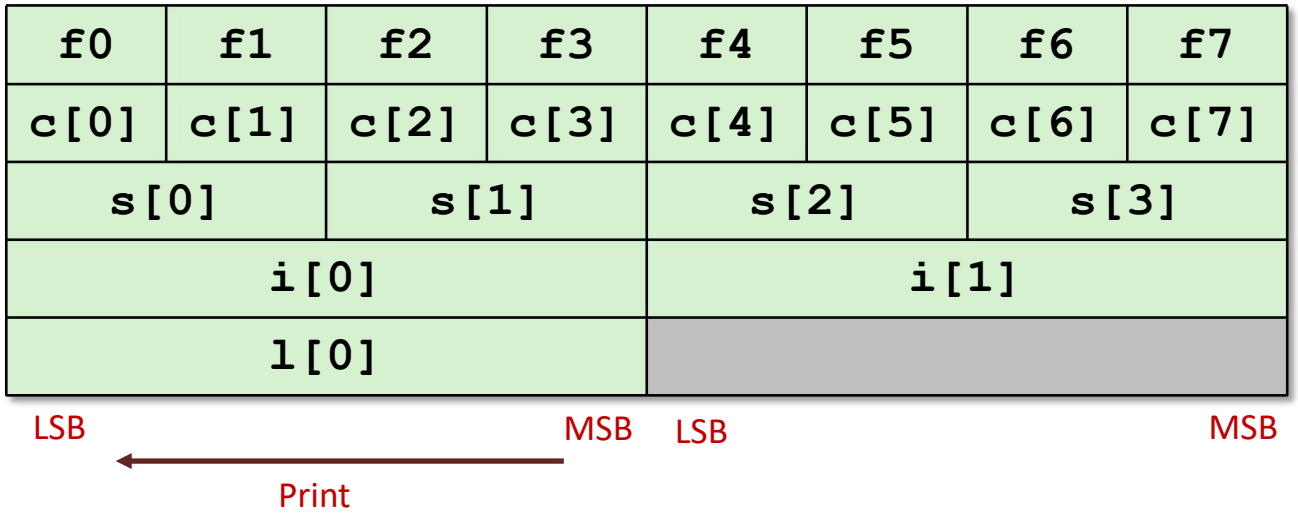


字节序举例（续）

```
int j;
for (j = 0; j < 8; j++)
   dw.c[j] = 0xf0 + j;

printf("Characters 0-7 =="
   "[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
      dw.c[0], dw.c[1], dw.c[2], dw.c[3],
      dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
   dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
   dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
   dw.l[0]);
```

IA32 的字节序

小端序

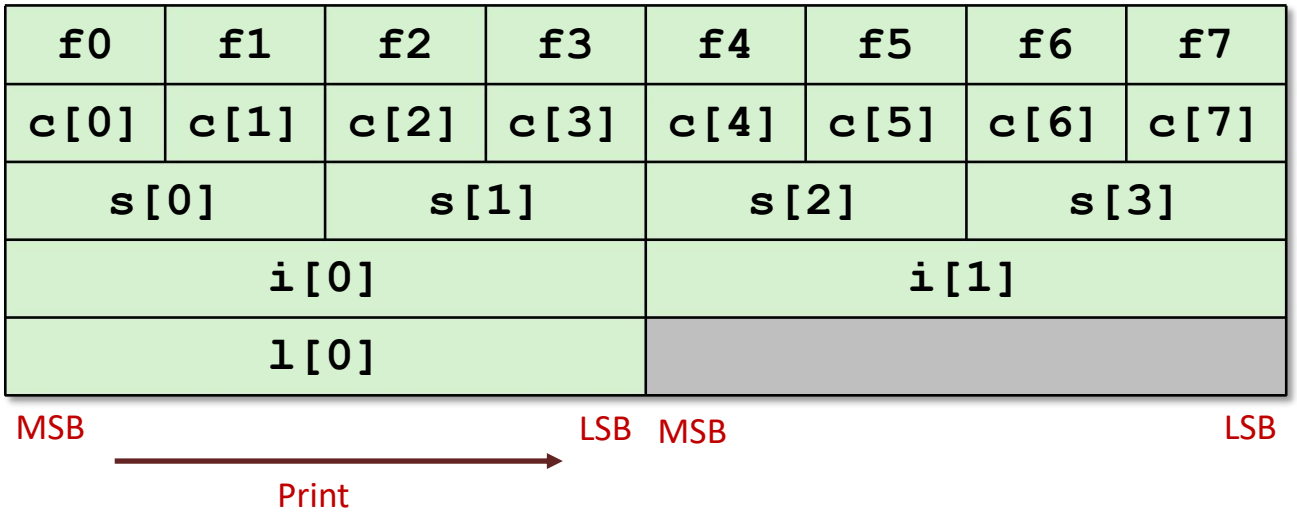


IA32 机器上的输出结果：

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

SUN 的字节序

大端序

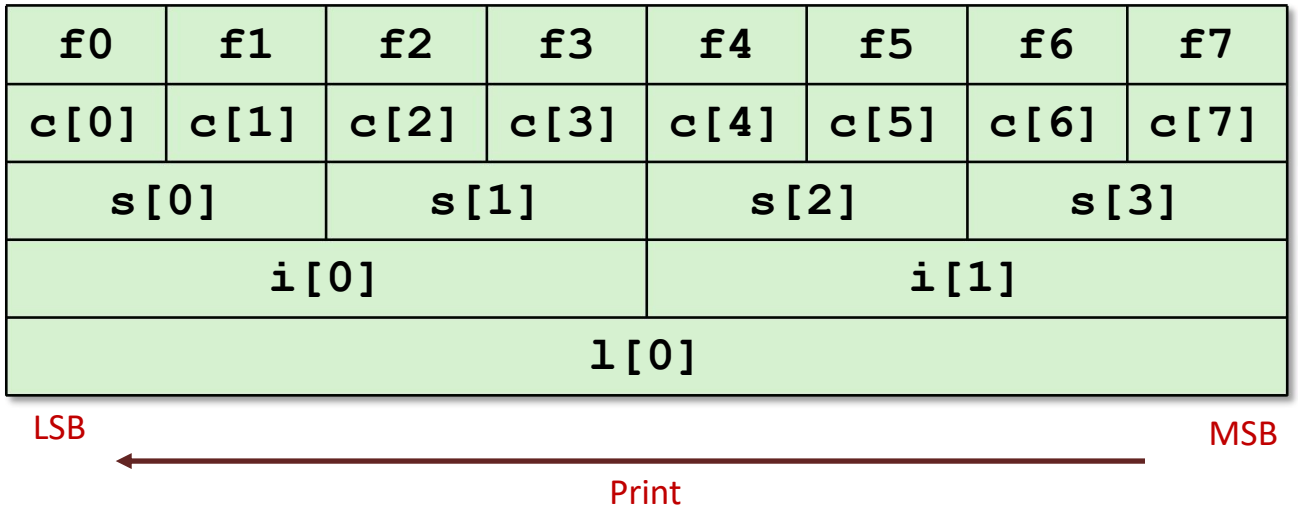


SUN 机器上的输出结果：

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

x86-64 的字节序

小端序



x86-64 机器上的输出结果：

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

C 组合数据类型小结

- **数组arrays**
  - 分配连续内存
  - 满足每个元素的对齐要求
  - 数组名为首元素的地址（常量）
  - 无边界检查bounds checking
- **结构体structures**
  - 按照声明的成员顺序分配内存
  - 各成员间、最后一个成员之后加上填充字节，以满足对齐要求
- **联合体unions**
  - 若干声明相互重叠
  - 可借此绕开 C 的类型机制