

# 程序的机器级表示V：高级主题

教师：吴锐

计算机科学与技术学院

哈尔滨工业大学

# 主要内容

- 内存布局
- 缓冲区溢出
  - 安全隐患
  - 防护
- 浮点数

# x86-64 Linux 内存布局

00007FFFFFFFH

未按比例绘制

## ■ 栈(Stack)

- 运行时栈 (8MB limit)
- 涉及局部变量

## ■ 堆(Heap)

- 按需动态分配
- 时机:调用 malloc(), calloc(), new()时

## ■ 数据(Data)

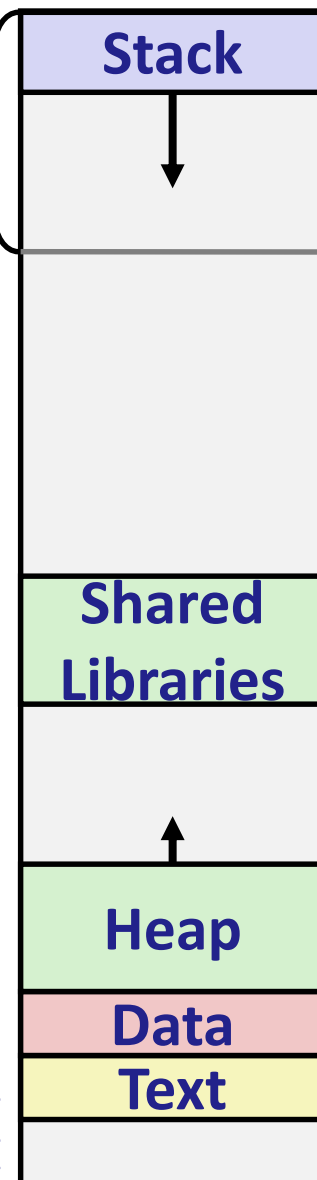
- 静态分配的内存中保存的数据
- 全局变量、static变量、字符串常量

## ■ 代码/共享库(Text / Shared Libraries)

- 只读的可执行的机器指令

8MB

400000H  
000000H



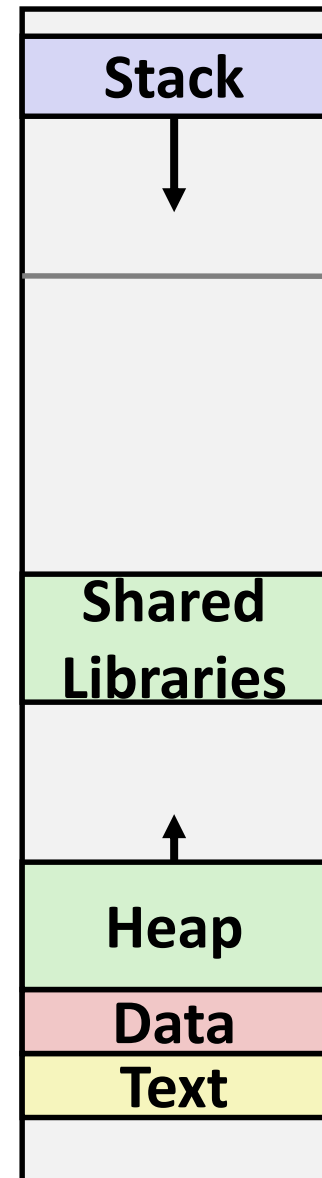
# 内存分配示例

```
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 */
    /* Some print statements ... */
}
```

**程序中各个部分都在哪里?**

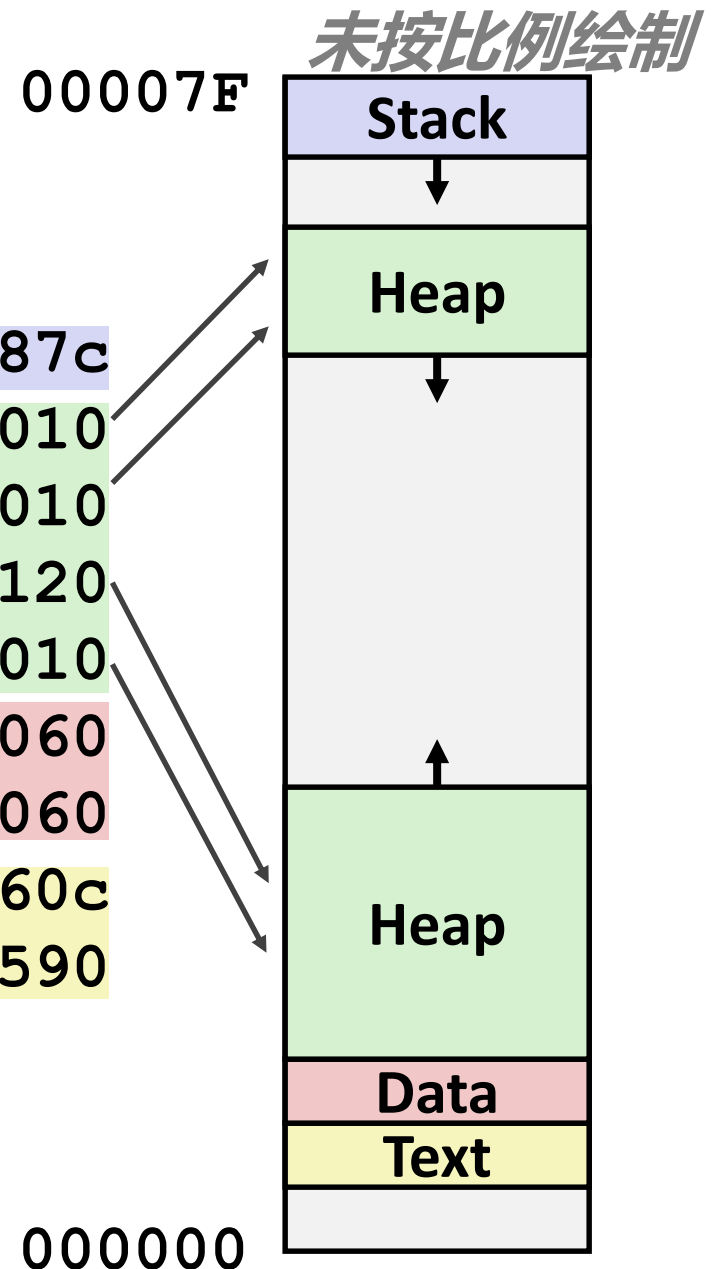
未按比例绘制



# x86-64 例子的地址

地址范围  $\sim 2^{47}$

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



# 主要内容

- 内存布局
- 缓冲区溢出
  - 安全隐患
  - 防护
- 浮点数

# 回忆: 内存引用的Bug示例

```
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; /* Possibly out of bounds */  
    return s.d;  
}
```

**fun (1) → 3.14**

**fun (2) → 3.13999998664856**

**fun (3) → 2.000000061035156**

**fun (4) → 3.14**

**fun (6) → Segmentation fault**

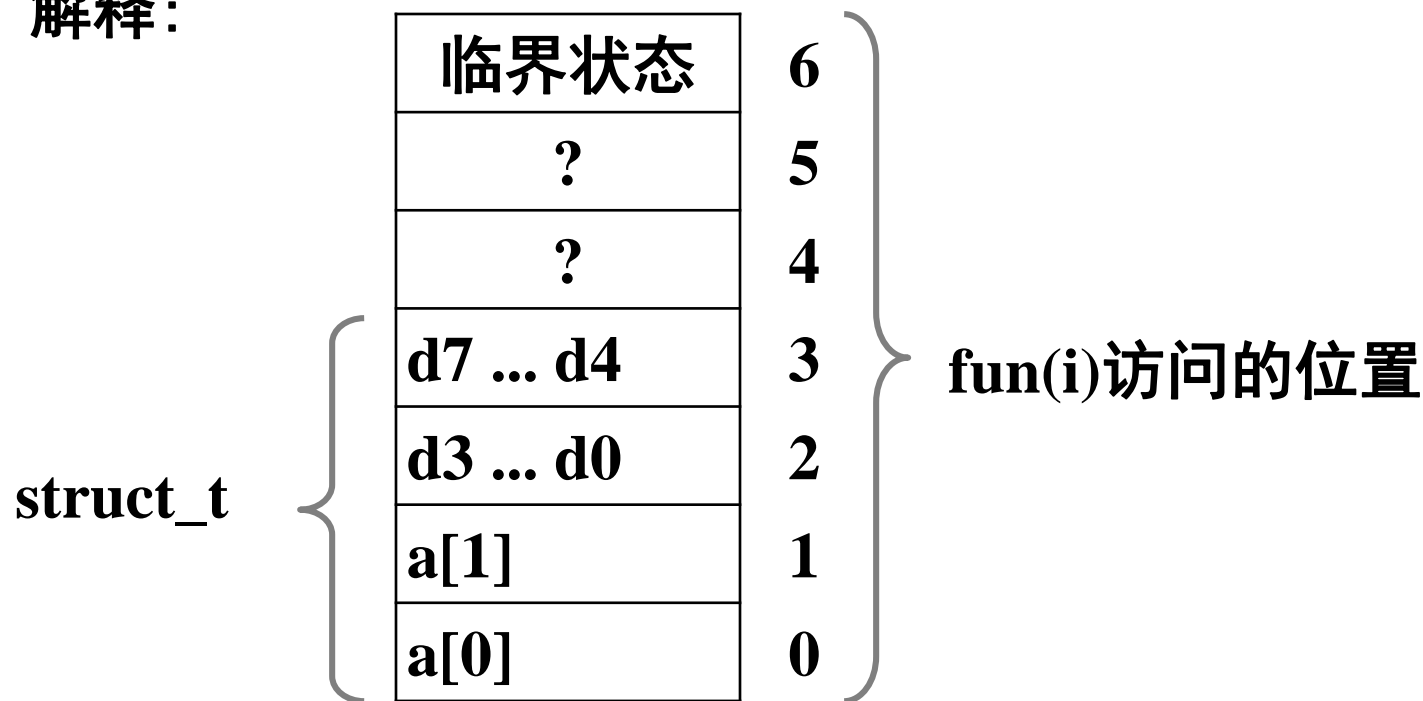
运行结果与系统有关

# 内存引用的Bug示例

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

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

解释：





# 这是个 **大** 问题。

## ■ 一般称为“缓冲区溢出”

- 当超出数组分配的内存大小（范围）

## ■ 为何是大问题？

- 示例 #1 安全隐患的技术原因
  - 示例#1总的原因是用户无知

## ■ 更一般的形式

- 字符串输入不检查长度
- 特别是堆栈上的有界字符数组
  - 有时称为堆栈粉碎(stack smashing)

# 字符串库的代码

## ■ Unix函数gets () 的实现

```
/* Get string from stdin */  
char *gets(char *dest){  
    int c = getchar();  
    char *p = dest;  
    while (c != EOF && c != '\n') {  
        *p++ = c;  
        c = getchar();  
    }  
    *p = '\0';  
    return dest;  
}
```

- 无法设定读入字符串的长度限制
- 其他库函数也有类似问题
  - strcpy, strcat: 任意长度字符串的拷贝
  - scanf, fscanf, sscanf, 使用 %s 转换符时

# 存在安全隐患的缓冲区代码

```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    gets(buf);  
    puts(buf);  
}
```

← btw, 多大才足够?

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

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

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

# 缓冲区溢出的反汇编

echo:

00000000004006cf <echo>:

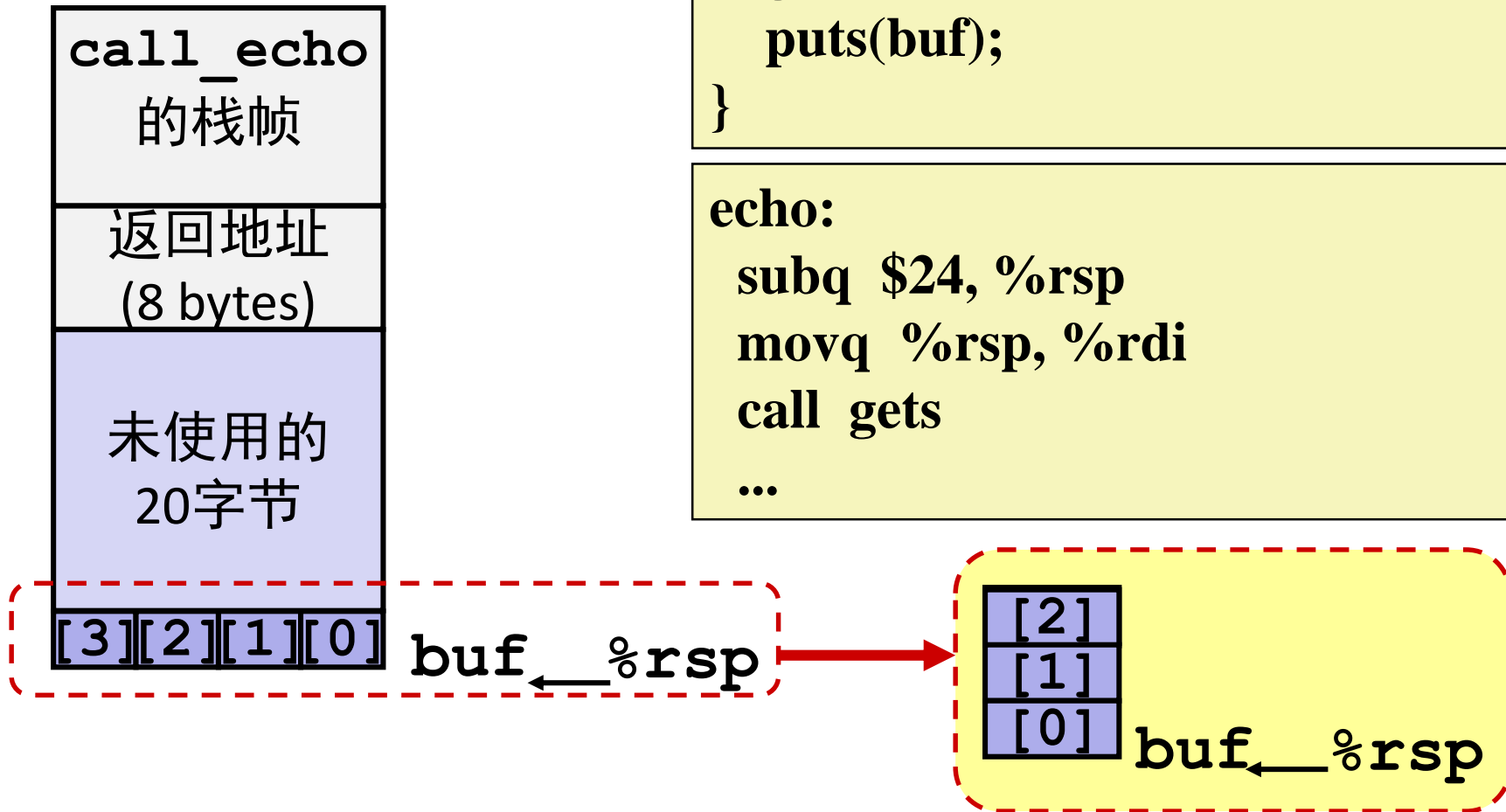
4006cf:	48 83 ec 18	sub	<b>\$0x18</b> ,%rsp
4006d3:	48 89 e7	mov	<b>%rsp,%rdi</b>
4006d6:	e8 a5 ff ff ff	callq	400680 <gets>
4006db:	48 89 e7	mov	%rsp,%rdi
4006de:	e8 3d fe ff ff	callq	400520 <puts@plt>
4006e3:	48 83 c4 18	add	\$0x18,%rsp
4006e7:	c3	retq	

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>
<b>4006f6:</b>	<b>48 83 c4 08</b>	add	<b>\$0x8,%rsp</b>
4006fa:	c3	retq	

# 缓冲区溢出的栈示

*调用gets之前*



# 缓冲区溢出的栈示例

*调用gets之前*



buf ← %rsp

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

call\_echo:

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

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

# 缓冲区溢出的栈示例 #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
...
```

缓冲区溢出，  
但没有破坏状态

```
unix> ./bufdemo-nsp
Type a string: 012345678901234567
8901201234567890123456789012
```

# 缓冲区溢出的栈示例 #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
...
```

溢出的缓冲区,返回  
地址被破坏

```
unix> ./bufdemo-nsp
Type a string: 0123456789012345678
901234
Segmentation Fault
```



# 缓冲区溢出的栈示例 #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);
    ...
}
```

call\_echo:

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

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

溢出的缓冲区,破坏了  
返回地址,但程序看  
起来能工作

```
unix> ./bufdemo-ns
Type a string: 01234567890123456
7890123012345678901234567890123
```

# 缓冲区溢出的栈示例 #3 —— 解读

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:

```

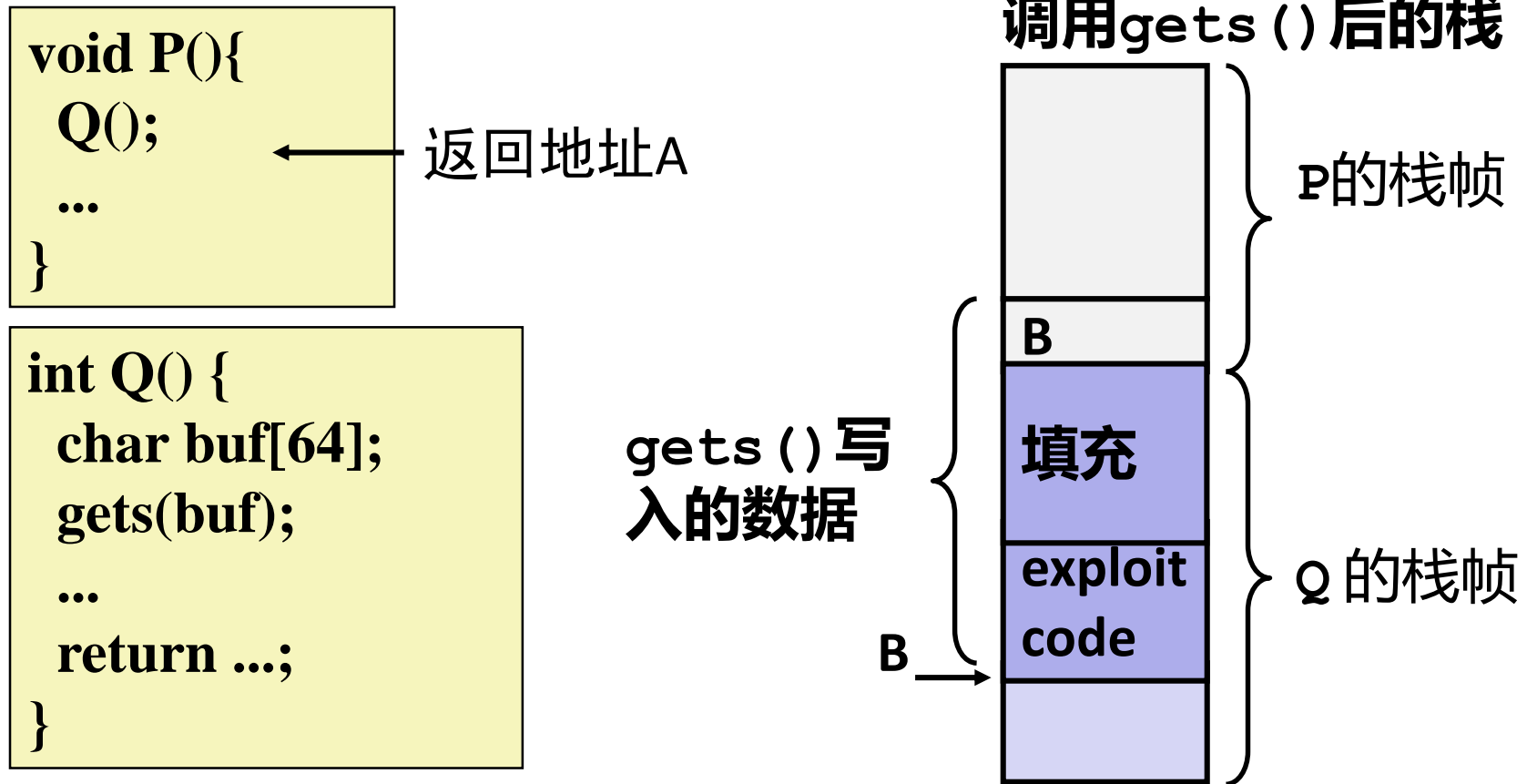
...
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq

```

返回到无关的代码

大多数情况不会修改临界状态，最终执行retq 返回主程序

# 代码注入攻击(Code Injection Attacks)



- 输入字符串包含可执行代码的字节序列!
- 将返回地址 A 用缓冲区 B 的地址替换
- 当 Q 执行 ret 后，将跳转到 B 处，执行漏洞利用程序(exploit code)

# 基于缓冲区溢出的漏洞利用程序

- **缓冲区溢出错误允许远程机器在受害者机器上执行任意代码。**
- **在程序中常见，令人不安**
  - 程序员持续犯相同的错误 ☹
  - 最近的措施使这些攻击更加困难。
- **经典案例**
  - 原始"互联网蠕虫"(Internet worm), 1988
  - 即时通讯战争"IM wars", 1999
  - Twilight hack on Wii, 2000s(不改动硬件，直接在Wii上运行自制程序)
- **在相应的实验中会学到一些技巧**
  - 希望能说服你永远不要在程序中留下这样的漏洞！！

# 例子: 原始互联网蠕虫 (1988)

## ■ 利用漏洞传播

- 指服务器(finger server)的早期版本用 `gets()` 读取客户机发来的参数:
  - `finger droh@cs.cmu.edu`
- 蠕虫利用发送假参数的方法攻击指服务器:
  - `finger "exploit-code padding new-return-address"`
  - 利用程序:用直接和攻击者相连的TCP链接, 在受害者机器上执行根用户shell

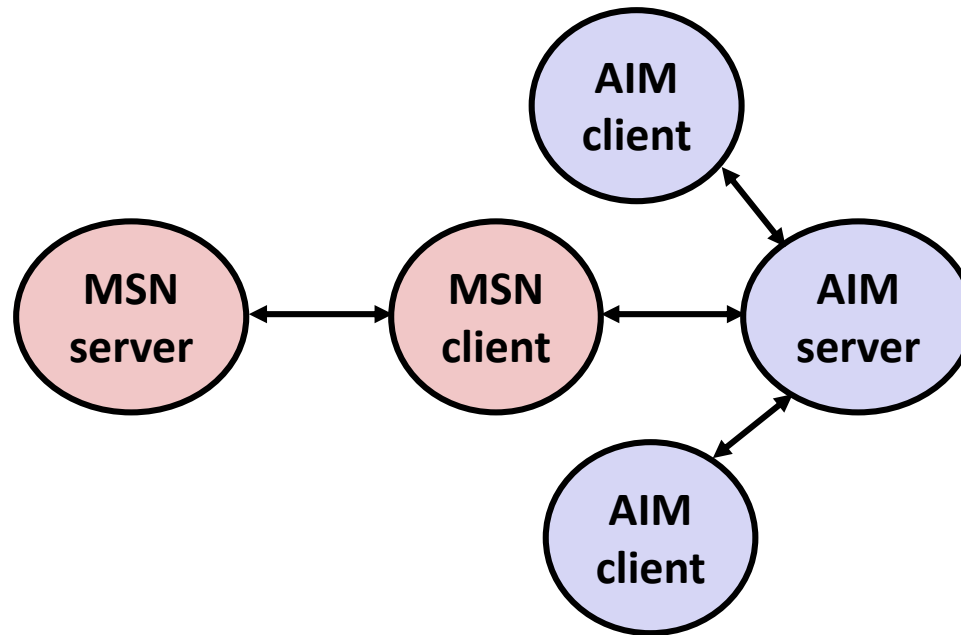
# 例子: 原始互联网蠕虫 (1988)

- 一旦进到机器上, 就扫描其他机器攻击
- 几小时内侵入了大概6000台 (互联网机器总数的10% )
  - 参考*Comm. of the ACM*在1989年6月的文章
  - 年轻的蠕虫作者被起诉.....
  - 计算机安全应急响应组(Computer Emergency Response Team)成立, ...现仍在CMU

# 例2:即时通讯战争

## ■ 1999年7月

- 微软发布了即时通讯系统MSN Messenger
- Messenger 的客户端能获取流行的美国在线(American Online, AOL)即时通讯服务 (AIM) 服务器



## 例2:即时通讯战争 (续...)

### ■ 1999年8月

- 很神秘, Messenger客户端无法再AIM服务器
- 微软和AOL 开始了即时通讯战争
  - AOL 变动服务器不允许Messenger客户端连接
  - 微软对客户进行更改以挫败AOL的变动
  - 至少有13个这样的小冲突
- 真正发生的到底是什么?
  - AOL 在他们自己的AIM服务器中发现了缓冲区溢出的漏洞
  - 他们利用这个bug检测并阻塞微软: 漏洞利用程序返回一个4字节的签名(在AIM客户端的某些位置存储的)到服务器
  - 当微软改变签名匹配程序时, AOL改变签名的位置



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

**后来确定这封电子邮件来源于  
微软内部！**

# 旁白：蠕虫和病毒

- **蠕虫(Worm):程序**
  - 可以自行运行
  - 可以将自己的完整版本传播到其他计算机上
- **病毒(Virus): 代码**
  - 将自己添加到别的程序中
  - 不独立运行
- **两者通常都能在计算机之间传播并造成破坏。**

# 针对缓冲区溢出攻击，怎么做？

- 避免溢出漏洞
- 使用系统级的防护
- 编译器使用“栈金丝雀” (stack canaries)

# 1. 代码中避免溢出漏洞(!)

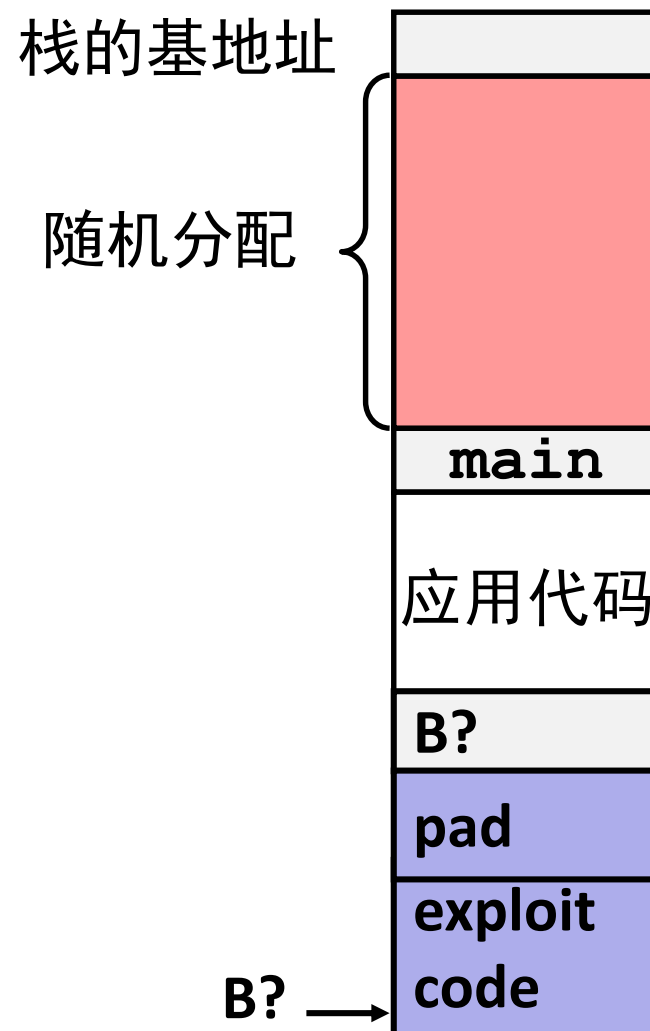
```
/* Echo Line */  
void echo() {  
    char buf[4]; /* Way too small! */  
    fgets(buf, 4, stdin);  
    puts(buf);  
}
```

- 例如，使用限制字符串长度的库例程
  - fgets 代替 gets
  - strncpy 代替 strcpy
  - 在 scanf 函数中别用 %s
    - 用 fgets 读入字符串
    - 或用 %ns 代替 %s, 其中 n 是一个合适的整数

## 2. 系统级防护

### ■ 随机的栈偏移

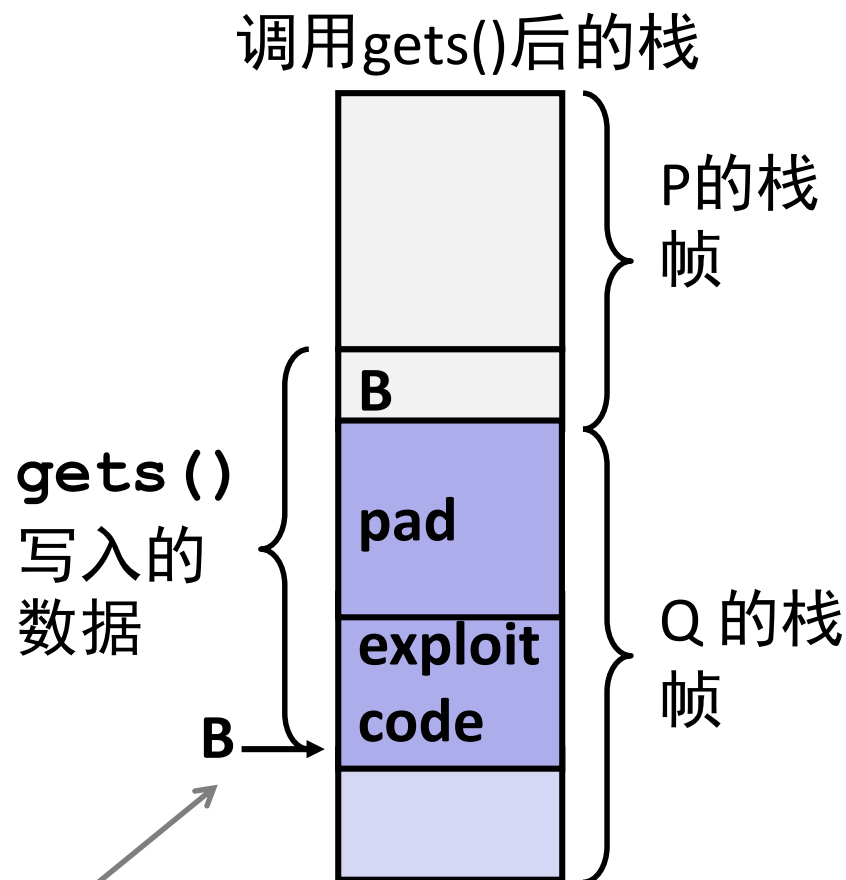
- 程序启动后，在栈中分配随机数量的空间
- 将移动整个程序使用的栈空间地址
- 黑客很难预测插入代码的起始地址
- 例如：执行5次内存申请代码
  - 每次程序执行，栈都重新定位



## 2. 系统级防护

### ■ 非可执行代码段

- 在传统的x86中，可以标记存储区为“只读”或“可写的”
  - 可以执行任何可读的操作
- x86-64添加显式“执行”权限
- 将stack标记为不可执行



所有执行该代码的尝试都将失败

### 3. 栈金丝雀(Stack Canaries)

#### ■ 想法

- 在栈中buffer之后的位置放置特殊的值——金丝雀 (“canary”)
- 退出函数之前，检查是否被破坏

#### ■ 用GCC 实现

- `-fstack-protector`
- 该选项现在是默认开启的(早期默认关闭)

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

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

# 保护缓冲区反汇编

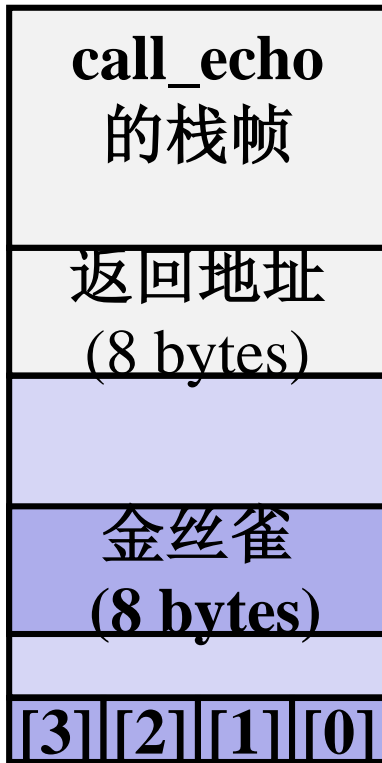
echo:

```
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
```



# 设立金丝雀(Canary)

**调用gets之前**



buf ← %rsp

```
/* Echo Line */
void echo(){
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    ...
    movq    %fs:40, %rax # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax  # Erase canary
    ...
```

# 核对金丝雀

调用gets后

call_echo 的栈帧			
返回地址 (8 bytes)			
金丝雀 (8 bytes)			
00	36	35	34
33	32	31	30

```
/* Echo Line */
void echo(){
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    ...
    movq    8(%rsp), %rax    # Retrieve from stack
    xorq    %fs:40, %rax    # Compare to canary
    je      .L6              # If same, OK
    call    __stack_chk_fail # FAIL
.L6:
    ...
```

buf ← %rsp

Input: 0123456

# 面向返回的编程攻击

## ■ 挑战(对黑客)

- 栈随机化使缓冲区位置难以预测。
- 标记栈为不可执行，很难插入二进制代码

## ■ 替代策略

- 使用已有代码
  - 例如：stdlib的库代码
- 将片段串在一起以获得总体期望的结果。
- 不用克服栈金丝雀

## ■ 从小工具构建攻击程序

- 以ret结尾的指令序列
  - 单字节编码为0xc3
- 每次运行，代码的位置固定
- 代码可执行

# 小工具例子 #1

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

00000000004004d0 <ab\_plus\_c>:

4004d0: 48 0f af fe      imul %rsi,%rdi

4004d4: 48 8d 04 17      lea (%rdi,%rdx,1),%rax

4004d8: c3      retq

$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

小工具地址 = 0x4004d4

## ■ 使用现有功能的尾部

## 小工具例子 #2

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

<setval>:

4004d9: c7 07 d4 **48 89 c7** movl \$0xc78948d4, (%rdi)

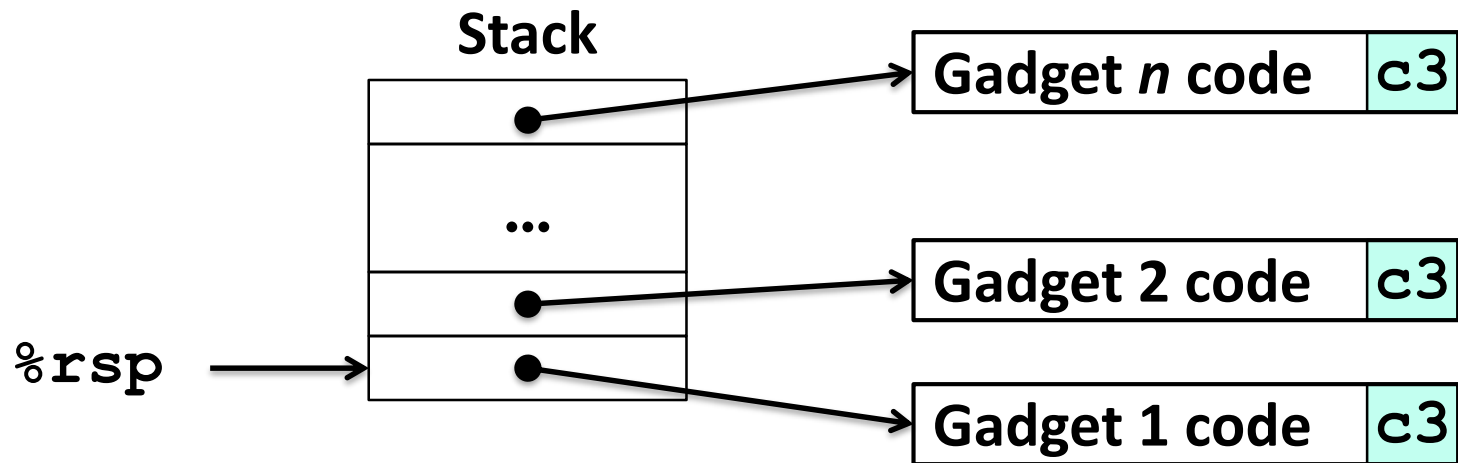
4004df: **c3** retq

movq %rax, %rdi 的编码

### ■ 改变字节码的用途

rdi ← rax  
小工具地址 = 0x4004dc

# 面向返回编程(ROP)的 执行



- **ret 指令触发**
  - 将开始运行 Gadget 1
- 每个小工具最终的 **ret** 将启动下一个小工具
- 通过小工具序列的运行，达到攻击目的。

# 主要内容

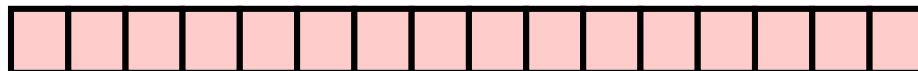
- 内存布局
- 缓冲区溢出
  - 安全隐患
  - 防护
- 浮点数

# 用SSE3编程

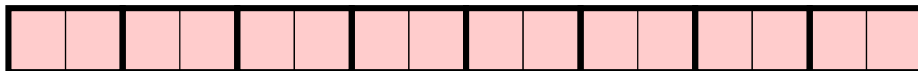
## XMM 寄存器

■ 共16 个 16字节的寄存器

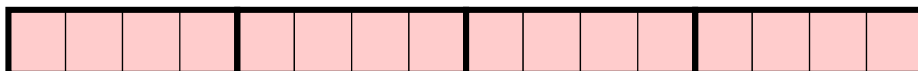
■ 16个单字整数



■ 8个16位整数



■ 4个32位整数



■ 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

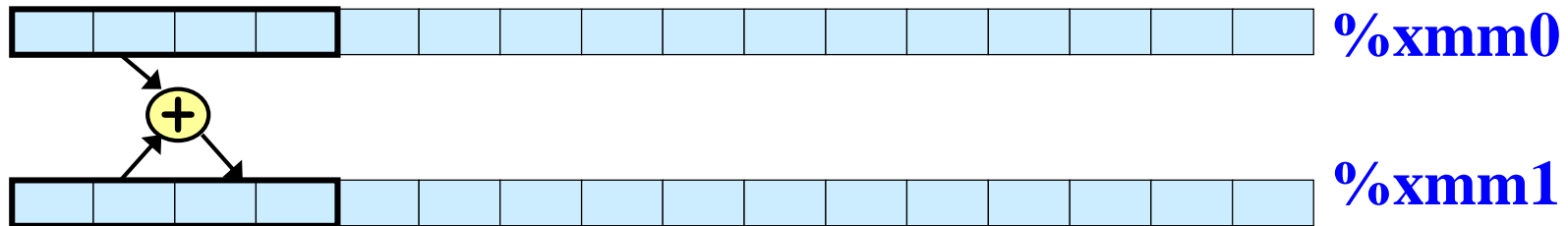




# 标量和SIMD操作

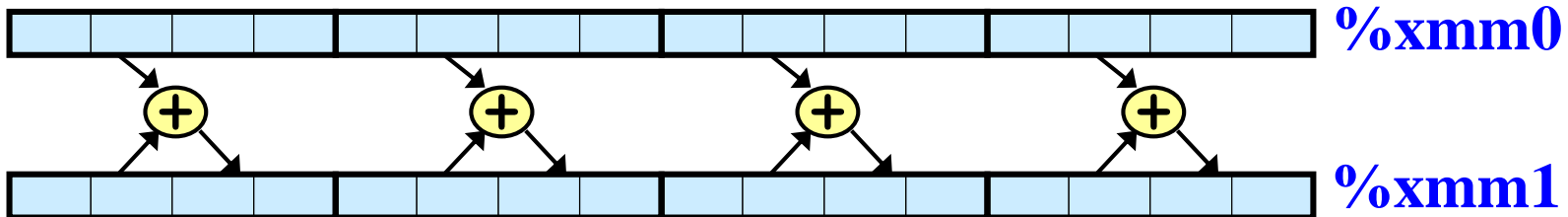
## ■ 标量操作:单精度

`addss %xmm0,%xmm1`



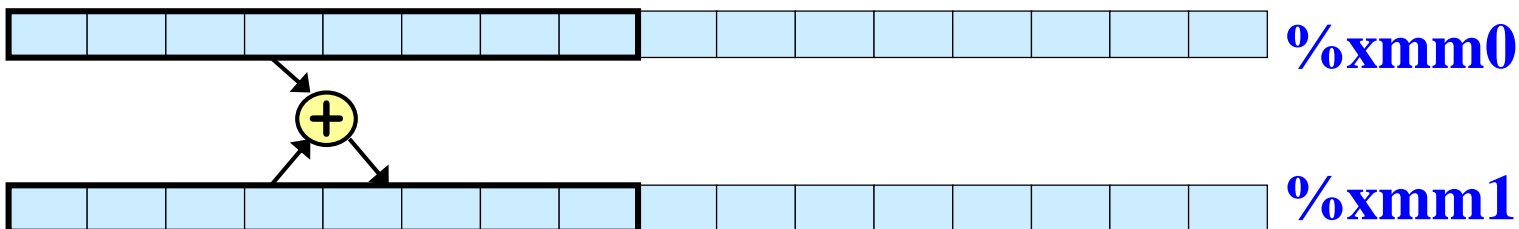
## ■ SIMD 操作: 单精度

`addps %xmm0,%xmm1`



## ■ 标量操作:双精度

`addsd %xmm0,%xmm1`



# 浮点基础

- 参数传递使用： `%xmm0`, `%xmm1`, ...
- 返回结果保存： `%xmm0`
- 所有XMM 寄存器都是调用者保存

```
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
```

# 浮点数的内存引用

- 单数传递：整数型 (包括指针) 参数用通用寄存器
- 单数传递：浮点型参数用XMM 寄存器
- 使用不同的mov指令在XMM 寄存器之间、或者内存和 XMM 寄存器之间传送数值

```
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
```

# 浮点数编程

## ■ 指令多

- 不同的操作、格式...

## ■ 浮点数比较

- `ucomiss` 和 `ucomisd`
- 设置条件码: CF, ZF和PF

## ■ 常量数值的使用

- 寄存器XMM0 清零:  
`xorpd %xmm0, %xmm0`
- 其他: 从内存载入