ICS 课程笔记

2016 秋季学期

笔记整理部分包括

3. Machine Prog: Procedures & Data

4. Machine Prog: Advanced

老师上课讲授知识点；教材中涉及的重要知识点；课外查到的补充资料

一些上课提到的有趣的内容

1.怎么把变量放在 寄存器/内存 里？

指定放在寄存器里？

Basically, you cannot. There is absolutely nothing in the C standard that gives you the control.

Using the register keyword is giving the compiler a hint that that the variable maybe stored into a register (i.e., allowed fastest possible access). Compiler is free to ignore it. Each compiler can have a different way of accepting/rejecting the hint.

Quoting C11, chapter §6.7.1, (emphasis mine)

A declaration of an identifier for an object with storage-class specifier register suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.

FWIW, most modern-day compilers can detect the mostly-used variables and allocate them in actual register, if required. Remember, CPU register is a scarce resource.

（摘自 http://stackoverflow.com/questions/36937570/how-do-make-sure-if-a-variable-defined-with-register-specifier-got-stored-in-c）

指定放在内存里？

上课的时候提到了一些方法：

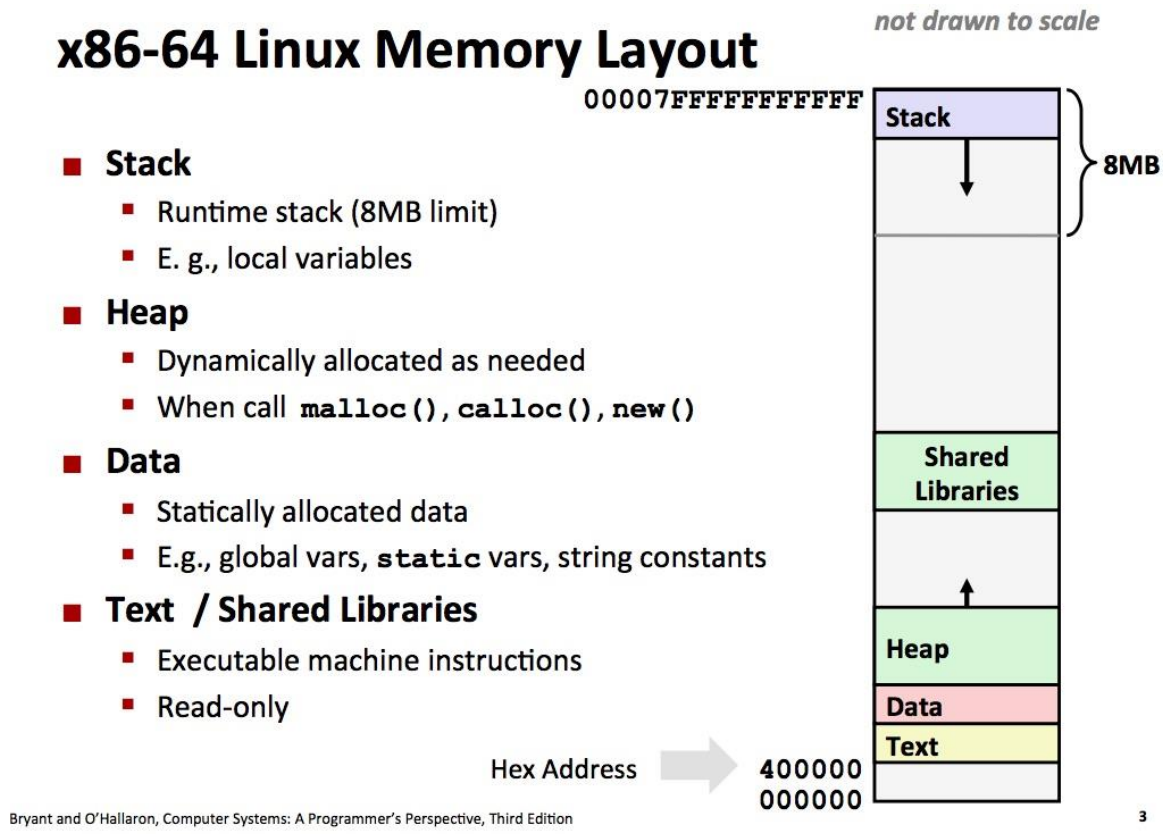
1.对某个变量加上取地址符&。（需要注意的是 the C Standard prohibits taking the address of a variable that is declared register, just as it does for bit fields in structs.）

2.数组，结构一定会放到内存里

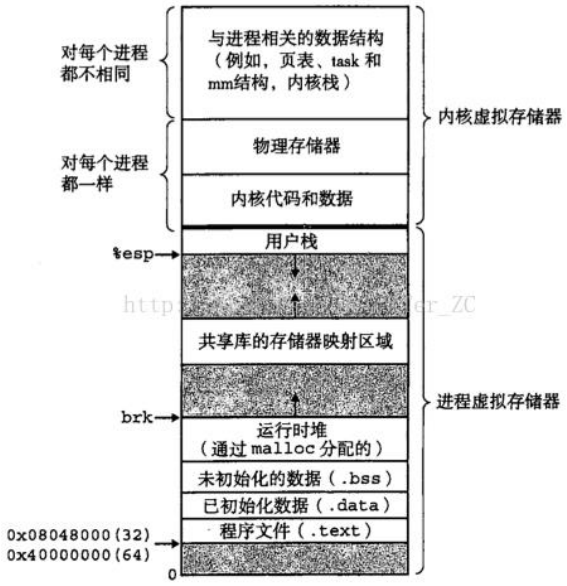
3.全局变量（虽然并不是放到栈里，而是在 data 段）

4. 使用 malloc，new（虽然也不是放到栈里，而是在 heap 段）

2.虚拟内存的结构



一个简单的示意图如上。



事实上会更加复杂一些。更具体一些我找了这么一张例子。

注意，stack 上面还有 os 相关的部分。data 部分企划书可以分为未初始化的数据部分 bss 和初始化的数据部分 data 两个部分。

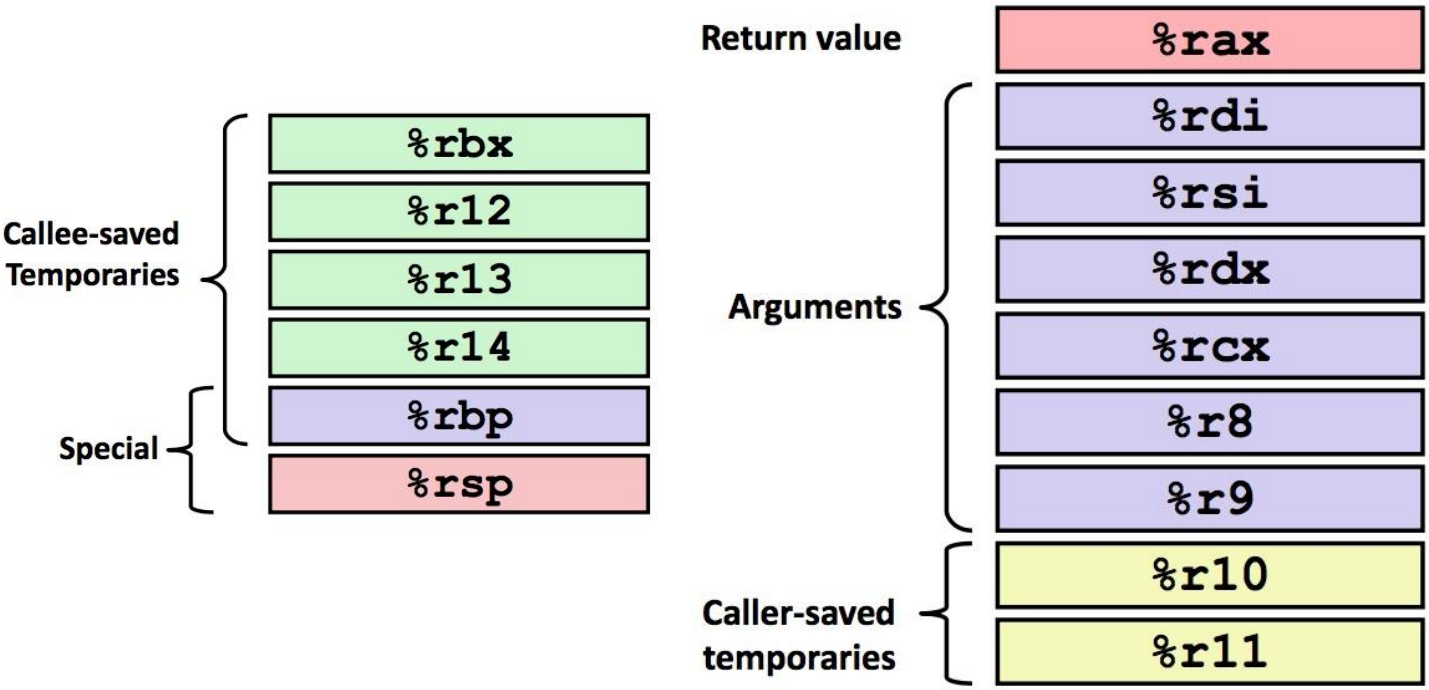
3.整数寄存器相关知识整理

以%rax 寄存器为例。64 位体系结构中，整数寄存器有八个 byte，对于它的各个部

```
|63..32|31..16|15-8|7-0|
                |AH.|AL.|
                |AX.....|
        |EAX.............|
|RAX.....................|
```

分，可以用不同的名字调用。

整数寄存器的功能



4.什么是 segmentation fault ？

课上提到这么两张图，提到了 segmentation fault

Recall: Memory Referencing Bug Example

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

- Result is system specific

页码：7/53

Explanation:

According to wikipedia:

A segmentation fault occurs when a program **attempts to access a memory location that it is not allowed to access**, or **attempts to access a memory location in a way that is not allowed** (for example, attempting to write to a read-only location, or to overwrite part of the operating system).

5.API 与 ABI 的比较

上课提到哪些寄存器是由 callee 保存的，哪些是由 caller 保存的。而这些规定是由 ABI 确定的。那么什么是 ABI？

One easy way to understand "ABI" is to compare it to "API".

You are already familiar with the concept of an API. If you want to use the features of, say, some library or your OS, you will use an API. **The API consists of data types/structures, constants, functions, etc that you can use in your code to access the functionality of that external component.**

An ABI is very similar. Think of it as the compiled version of an API (or as an API on the machine-language level). When you write source code, you access the library though an API. Once the code is compiled, your application accesses the binary data in the library through the ABI. **The ABI defines the structures and methods that your compiled application will use to access the external library** (just like the API did), only on a lower level.

ABIs are important when it comes to applications that use external libraries. If a program is built to use a particular library and that library is later updated, you don't want to have to re-compile that application (and from the end-user's standpoint, you may not have the source). If the updated library uses the same ABI, then your program will not need to change. The interface to the library (which is all your program really cares about) is the same even though the internal workings may have changed. Two versions of a library that have the same ABI are sometimes called "binary-compatible" since they have the same low-level interface (you should be able to replace the old version with the new one and not have any major problems).

（摘自 http://stackoverflow.com/questions/2171177/what-is-application-binary-interface-abi）

6.汇编语言常用指令整理

Address model

- **Most General Form**

  D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - D: Constant “displacement” 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

  (Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]]**

  D(Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]+D]**

  (Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]]**

lea （load effective address）

- **`leaq` *Src, Dst*** lea:计算下标，直接把这个下标给dst
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression

- **Example**

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax  # t = x+2*x
salq $2, %rax             # return t<<2
```

逻辑与运算

Two Operand Instructions:

Format	*Computation*		
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

One Operand Instructions

`incq`	*Dest*	*Dest = Dest + 1*
`decq`	*Dest*	*Dest = Dest – 1*
`negq`	*Dest*	*Dest = – Dest*
`notq`	*Dest*	*Dest = ~Dest*

jump 指令

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

课本内容： Machine Prog: Procedures & Data

一个过程调用包括将数据和控制从代码的一部分传递到另一部分。另外，它还必须在进入时为过程的局部变量分配空间，并在退出时释放这些空间。大多数机器，包括IA32，只提供转移控制到过程和从过程转移出控制这种简单的指令。数据传递、局部变量的分配和释放通过操纵程序栈来实现。

总结：1.转移控制由控制指令（call，ret）来实现

2.数据传递以及局部变量分配释放由操作程序栈来实现。

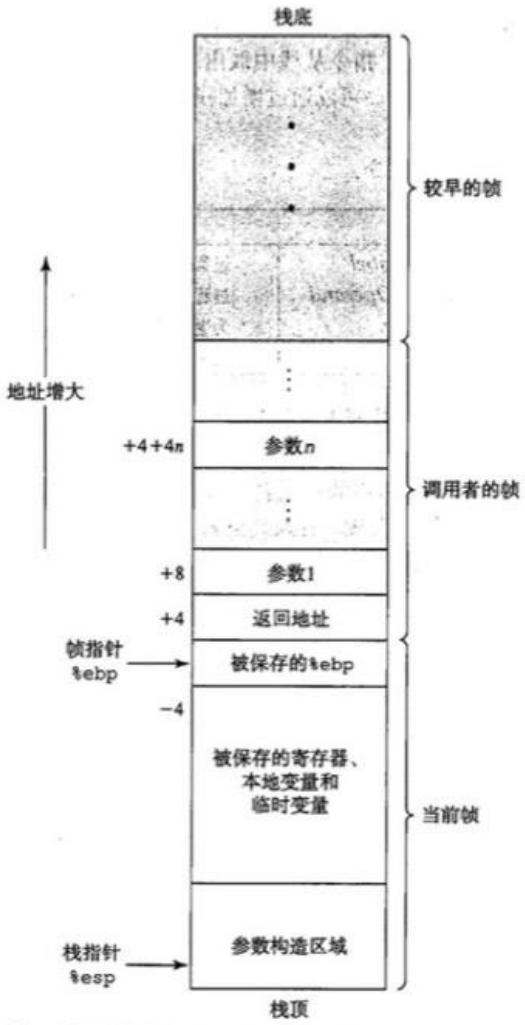


图 3-21　栈帧结构（栈用来传递参数、存储返回信息、保存寄存器，以及本地存储）

栈帧结构

机器用栈来传递过程参数、存储返回信息、保存寄存器用于以后回复，以及本地存储。为单个过程分配的那部分栈称为栈帧(stack frame)。

栈帧是由两个指针来定界的。

寄存器%rbp(bottom pointer)被称为帧指针，寄存器%rsp(stack pointer)被称为栈指针。

控制转移

支持过程转移和调用的指令如下：

指　令		描　述
call	*Label*	过程调用
call	**Operand*	过程调用
leave		为返回准备栈
ret		从过程调用中返回

call 指令，作用是将返回地址入栈，并跳转到被调用过程的起始处。返回地址是紧跟在 call 后面的那条指令的地址，这样当被调用过程返回时，执行会从此继续。

ret 指令，从栈中弹出地址，并跳转到那个位置。（ret = pop %rip）

一个例子

源代码

code/asm/swapadd.c

```
1    int swap_add(int *xp, int *yp)
2    {
3        int x = *xp;
4        int y = *yp;
5
6        *xp = y;
```

```
7         *yp = x;
8         return x + y;
9     }
10
11    int caller()
12    {
13        int arg1 = 534;
14        int arg2 = 1057;
```


```
        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
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
```

帧

帧栈的变化

caller

```
        Calling code in caller
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
```

注意到在调用之前 swap_add 之前，帧指针指向这个帧栈的底部，栈指针指向这个帧栈的头部。这个帧栈里存的数据是在这个函数中定义的两个 local 变量和他们的地址（因为要被用到。）

然后执行 call 指令，也就是将返回地址，也就是下一条要执行的代码的地址，压入栈，然后直接跳转到 call 的地方，执行被调用的函数。call 并不做其他多余的事情。

callee

callee 的代码主要分三个部分，初始化栈，执行，恢复栈的状态和返回。

```
        Setup code in swap_add
1     swap_add:
2       pushl %ebp                 Save old %ebp
3       movl %esp,%ebp             Set %ebp as frame pointer
4       pushl %ebx                 Save %ebx
```

初始化的时候，首先把 old %ebp 压栈，这样下次可以知道上一个帧栈的底部在哪里。然后第二部讲 ebp 指向 old %ebp，这样就确定了这个被调用的函数的帧栈的底部。然后由于 swap_add 需要用到%ebx 作为临时储存，而且%ebx 是一个被调用者保存的寄存器。所以要保存原有的%ebx 的值，就把它压栈，以便之后执行完后再恢复回去。

然后是执行。这部分没什么好说的。要注意的是结果会放到%eax 中去，是默认的 return value 储存用的寄存器。

最后是恢复栈的状态和返回。恢复的时候，先恢复%ebx。再将 esp 指向 ebp 指向的地方（虽然在这个特殊的例子里面是没有必要的，但是我猜想如果有跟多 local 变量在 callee 里面被创建，这一步是有必要的）。再恢复%ebp，这样%ebp 又指向了 caller

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
```

帧栈的栈底，同时%esp 指向了“返回地址”。最后在 ret 指令，相当于 pop ip，把返回地址，也就是下一条要执行的代码的地址出栈执行，同时，%esp 指向了 caller 的栈顶。这样就完成了一次函数的调用。

数组

C 语言的数组的基本原则：对于数据类型 T 和整型常数 N，声明如下： T A[N]

它有两个效果。

首先，它在存储器中分配一个 L*N 字节的连续区域；这里 L 是数据类型 T 的大小(单位为字节)。

其次，它引入了标识符 A，可以用来作为指向数组开头的指针。可以用从 0 到 N-1 之间的整数索引来访问数组元素。数组元素 i 会被存放在地址为 xa + L * i 的地方(xa 为指向数组开头的指针)

指针运算

C 语言允许对指针进行运算，而计算出来的值会根据该指针引用的数据类型的大小进行伸缩。也就是说，如果 p 是一个指向类型为 T 的数据的指针，p 的值为 xp，那么表达式 p+i 的值为 xp+L*i，这里 L 是数据类型 T 的大小。

多维数组

用递归的方式定义。

行	元素	地址
A[0]	A[0][0]	x_A
	A[0][1]	x_A+4
	A[0][2]	x_A+8
A[1]	A[1][0]	x_A+12
	A[1][1]	x_A+16
	A[1][2]	x_A+20
A[2]	A[2][0]	x_A+24
	A[2][1]	x_A+28
	A[2][2]	x_A+32
A[3]	A[3][0]	x_A+36
	A[3][1]	x_A+40
	A[3][2]	x_A+44
A[4]	A[4][0]	x_A+48
	A[4][1]	x_A+52
	A[4][2]	x_A+56

结构

创建一个数据类型，将可能不同类型的对象聚合到一个对象中。结构的各个组成部分用名字来引用。类似于数组的实现，结构的所有组成部分都存放在存储器中一段连续的区域内，而指向结构的指针就是结构第一个字节的地址。

联合

提供了一种方式，能够规避 C 语言的类型系统，允许以多种类型来引用一个对象。联合声明的语法与结构的语法一样，只不过语义相差比较大。它们是用不同的字段来引用相同的存储器块。

定义一个联合。

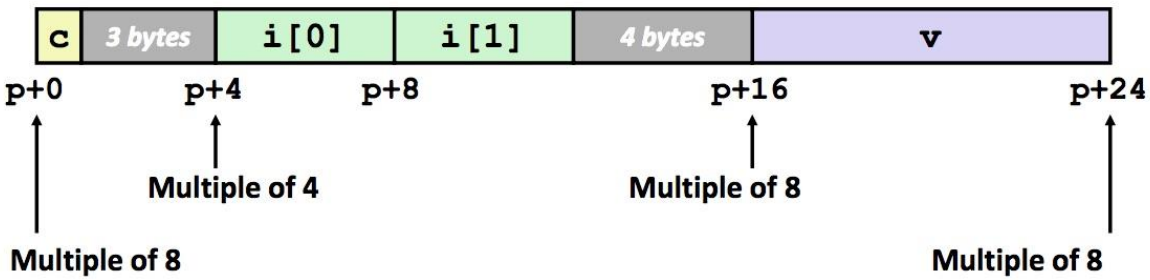
```
union U3 {
    char c;
    int i[2];
    double v;
};
```

对于 union U3 *的指针 p，p->c , p->i[0] , p->v 都是数据结构的起始位置，还要注意，一个联合的总的大小等于它最大域 的大小。

对齐

数据对齐的原则：

如果某数据类型需要 kbytes，那么他的起始地址必须是 k 的整数倍。

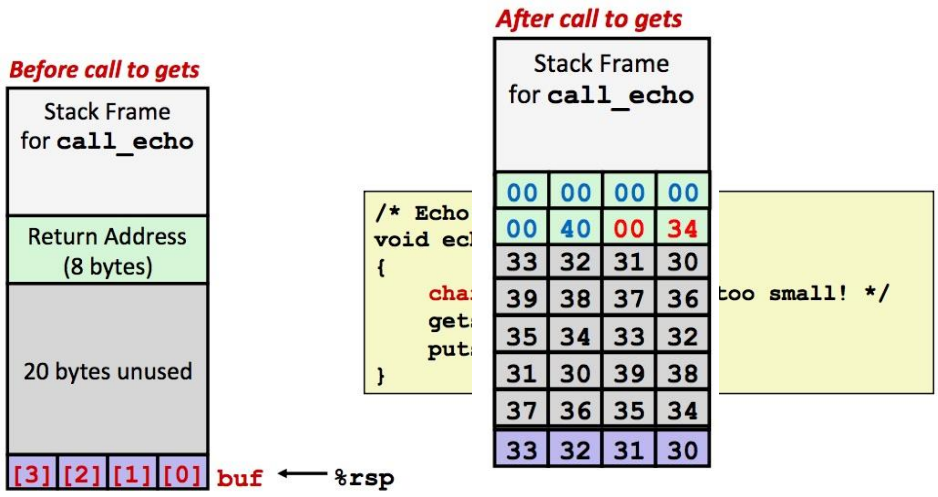


对齐的动机在于，内存在读取的时候是 4 或 8bytes 一起取的（与具体的机器相关），对其之后同一个 byte 数据不会需要取两次。

课本内容： Machine Prog: Advanced

缓存区溢出攻击

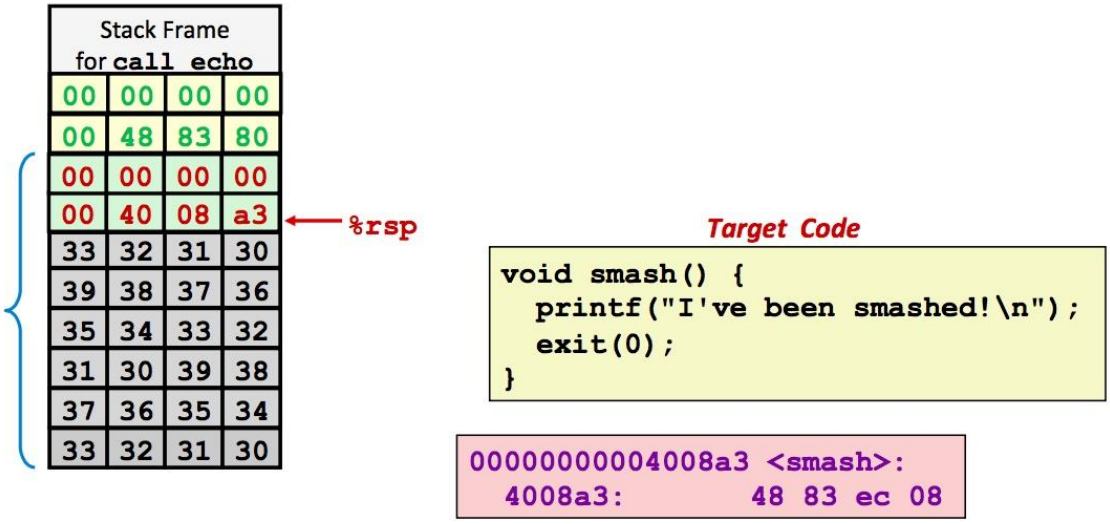
C 对于数组引用不进行任何边界检查，而局部变量和状态信息，都存放在栈中。这两种情况结合到一起就可能导致严重的程序错误，对越界的数组元素的写操作会破坏存储在栈中的状态信息。当程序使用这个被破坏的状态，试图重新加载寄存器或执行 ret 指令时，就会出现很严重的错误。



注意到，由于不检查边界，输入过多的东西之后，Return Address 都有可能会改变。

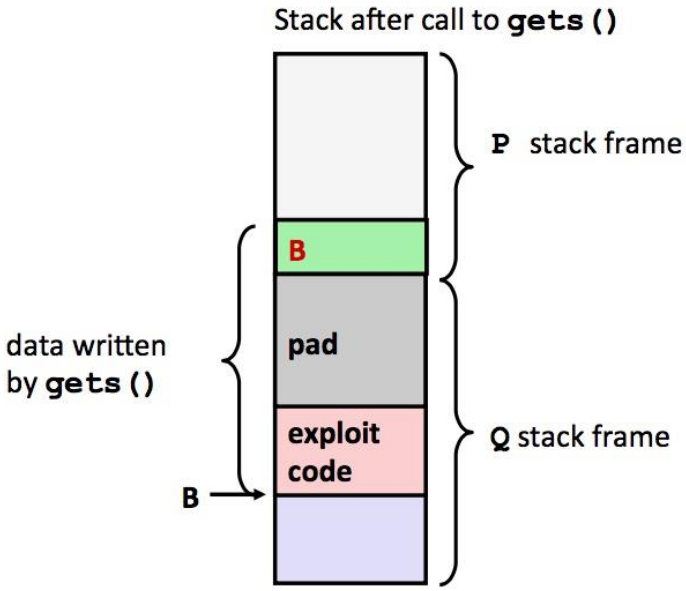
有两种具体的实供给的方式。

Smashing Attack 利用缓存区溢出攻击，改写 return address，从而当函数执行完，再



ret 的时候，跳转到另一段目标代码。图示如下。

Code Injerction Attack 则是直接跳转到刚刚输入的地方，执行刚刚写的那段代码。图示如下。



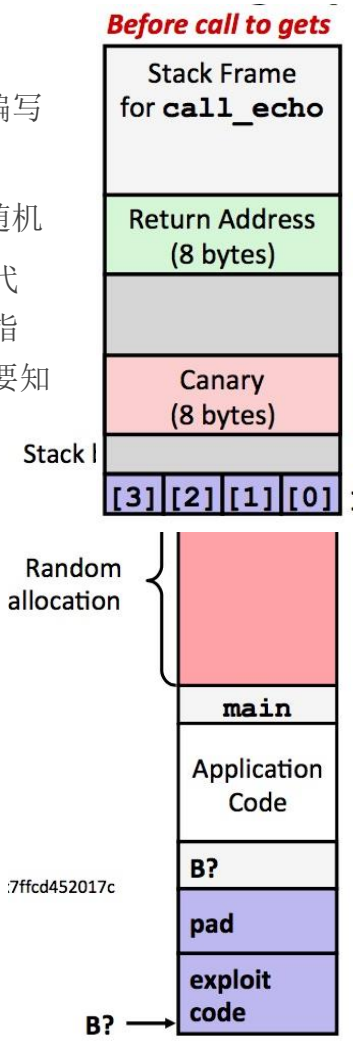
防止缓存区溢出攻击

1.程序员级别的保护：编写 代码的时候注意栈溢出（如用 fgets 取代 gets）

2.系统级别的保护：栈随机 化

为了在系统中插入攻击代 码，攻击者不但要插入代码，还需要插入指向这段代码的指 针，这个指针也是攻击字符串的一部分。产生这个指针需要知 道这个字符串放置的栈地址。

栈随机化的思想使得 栈的位置在程序每次运行时都有变化。



3.系统级别的保护：限制可执行代码区域

消除攻击者向系统中插入可执行代码的能力，只有保存编译器产生的代码的那部分内存才是可执行的，其他部分可以被限制为只允许读和写

4.系统级别的保护：栈破坏检测

保护者(stack protector)机制，用来检测缓冲区越界。其思想是在栈帧中任何局部缓冲区与栈状态之间存储一个特殊的金丝雀(canary)值，也称为哨兵值(guard value)，如下图所示。这个值是在程序每次运行时随机产生的，因此，攻击者没有简单的办法能够知道它是什么。在恢复寄存器状态和从函数返回之前，程序检查这个金丝雀值是否被该函数的某个操作或者该函数调用的某个函数的某个操作改变了。如果是，那么程序异常终止。