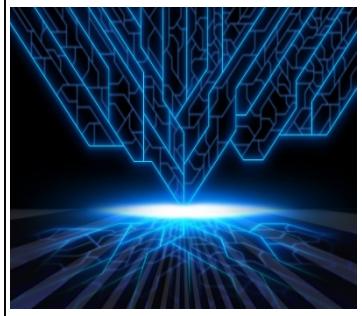
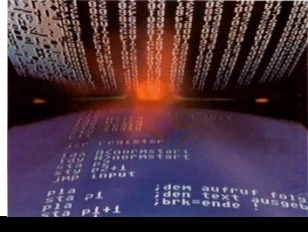
第10讲
过程



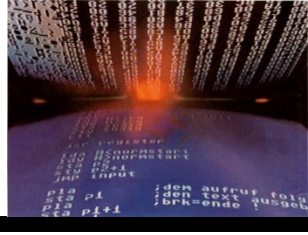
吴海军
南京大学计算机科学与技术系

主要内容



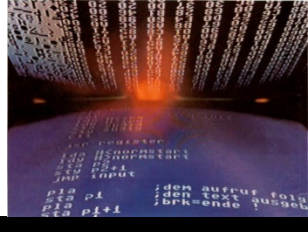
- 栈
- 过程
- 递归

过程



- 过程是一种软件抽象，一种代码封装方式。
- 用一组指定的参数和返回值实现了某种功能。
- 形式多样： 函数、方法、子例程、过程等等。
- 过程调用包括
  - 传递控制，返回地址的保护与恢复。
  - 传递数据，传入参数，返回结果。
  - 内存分配，局部变量内存空间的申请与释放
- C语言过程调用中关键特性-栈数据结构
- x86-64提供通过寄存器传递参数的机制，如果参数少于6个，可以不需要栈帧

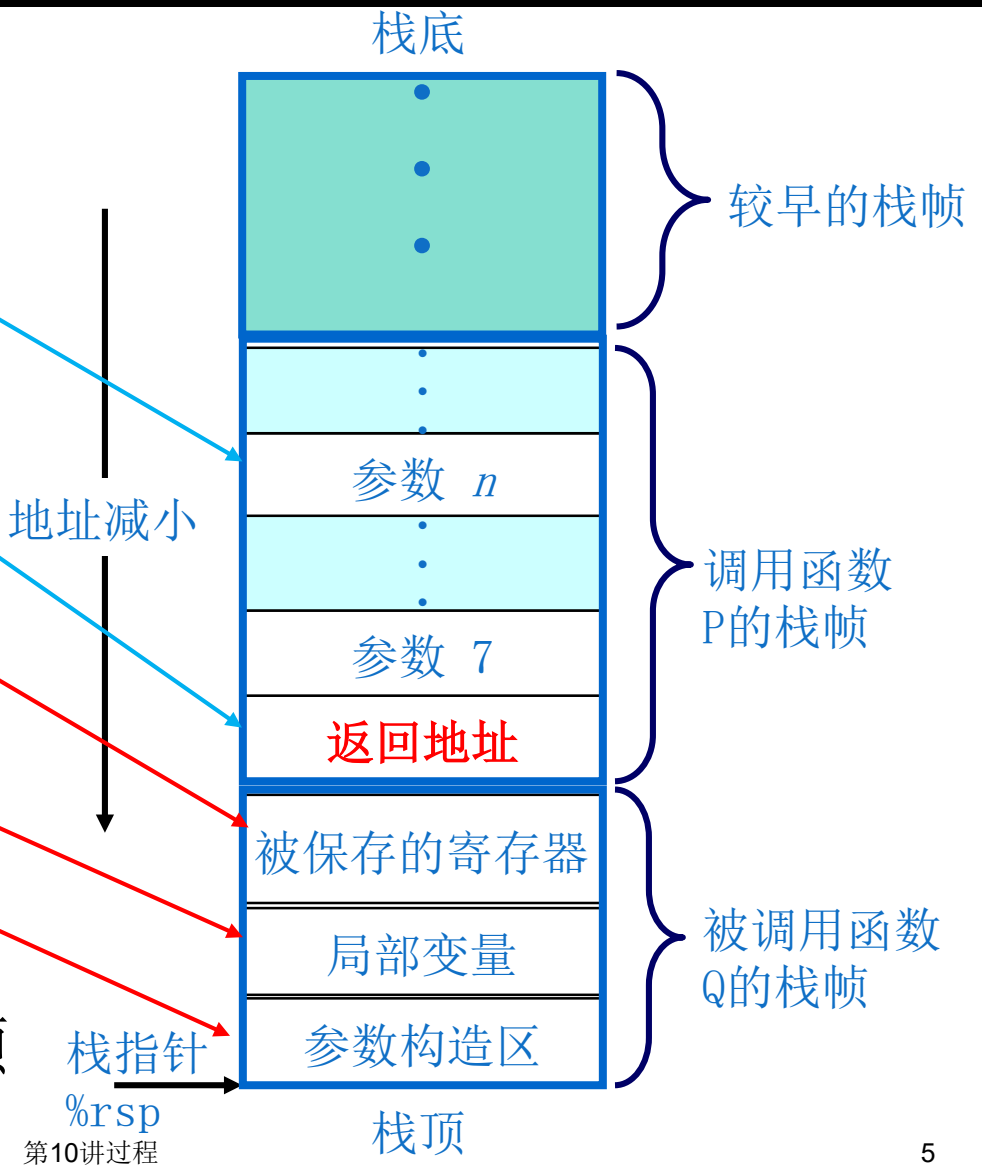
栈规则



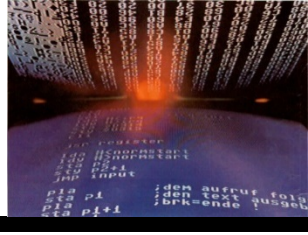
- 支持递归的语言
  - 例如：C, Pascal, Java
  - 代码必须是“*可重入*”的
    - 同时存在单个过程的多个运行实例
  - 需要存储每个实例状态的空间
    - 参数、局部变量、返回指针
- 栈规则
  - 给定过程的状态，只在有限的时间内需要
    - 从被调用开始，到返回时结束
  - 被调用者先于调用者返回
- 栈按照帧（Frame）进行分配
  - 单一过程实例的状态

栈空间结构

- 调用者P栈帧
  - 第7参数及以后
  - 返回地址
    - 通过**call指令进栈**
- 当前Q的栈帧
  - 被保护的寄存器内容
  - 局部变量
  - 被调用函数的参数区
    - 参数建立
- 指针
  - 栈指针%rsp 指示栈顶



转移控制



- 在函数P中执行call Q语句，CPU控制权将从函数P转移到函数Q。
- 在函数Q中，执行ret语句后，继续执行函数P中的后续代码。
- 过程调用:`call label`
  - **返回地址进栈**;
  - 跳转到*label处*
- 过程返回:`ret`
  - **返回地址出栈**;
  - 跳转到地址处

pushq 返回地址

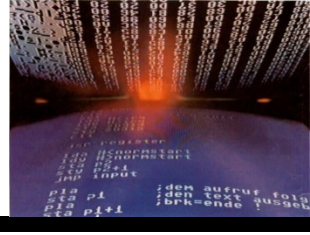
%rip指令寄存器=被调用起始地址

返回地址：Call语句后面一条指令的地址

popq 返回地址;

%rip指令寄存器=返回地址

过程调用示例

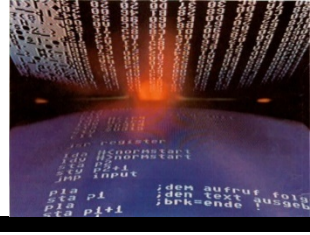


```
    Beginning of function multstore
1   0000000000400540 <multstore>:
2     400540:   53                          push   %rbx
3     400541:   48 89 d3                    mov    %rdx,%rbx
      . . .
    Return from function multstore
4     40054d:   c3                          retq

  Call to multstore from main
5     400563:   e8 d8 ff ff ff              callq  400540 <multstore>
6     400568:   48 8b 54 24 08              mov    0x8(%rsp),%rdx
```

- 执行call前：%rip=0x400563
- 执行call后：%rip=0x400540，(%rsp)=0x400568,%rsp-=8
- 执行retq后：%rip=(%rsp)=0x400568，%rsp+=8

过程调用示例



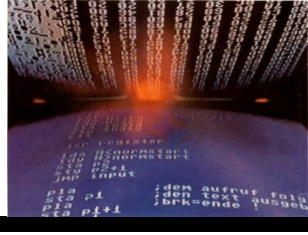
```
    Disassembly of leaf(long y)
    y in %rdi
1   0000000000400540 <leaf>:
2     400540:  48 8d 47 02        lea    0x2(%rdi),%rax  L1: y+2
3     400544:  c3                 retq                   L2: Return

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
```

执行M1后：%rip=0x400545，(%rsp)=0x400560,%rsp-=8

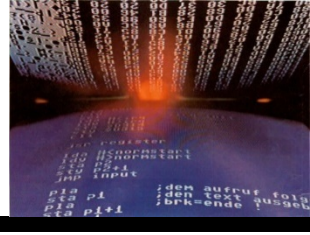
```
     Call to top from function main
9     40055b:  e8 e5 ff ff ff     callq  400545 <top>    M1: Call top(100)
10    400560:  48 89 c2           mov    %rax,%rdx       M2: Resume
```

代码执行过程



指令	%RIP	%rsp	*%rsp	%rdi	%rax
M1	0x40055b	%rsp	-	100	-
T1	0x400555	%rsp-8	0x500560	100	-
T2	0x400559	%rsp	0x500560	95	-
L1	0x400540	%rsp-8	0x50054e	95	-
L2	0x400544	%rsp	0x50054e		97
T3	0x40054e	%rsp+8	0x500560		97
T4	0x400551	%rsp	0x500560		194
M2	0x400560	%rsp+8	-		194

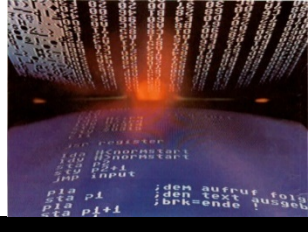
数据传送



- 过程调用的参数传递
  - 通过寄存器传送参数
  - 最多可有**6个整型或指针型参数通过寄存器**传递
  - 超过6个入口参数时，后面的通过栈来传递
  - 在栈中传递的参数若是基本类型，则都被分配8个字节

操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

使用栈帧的局部存储

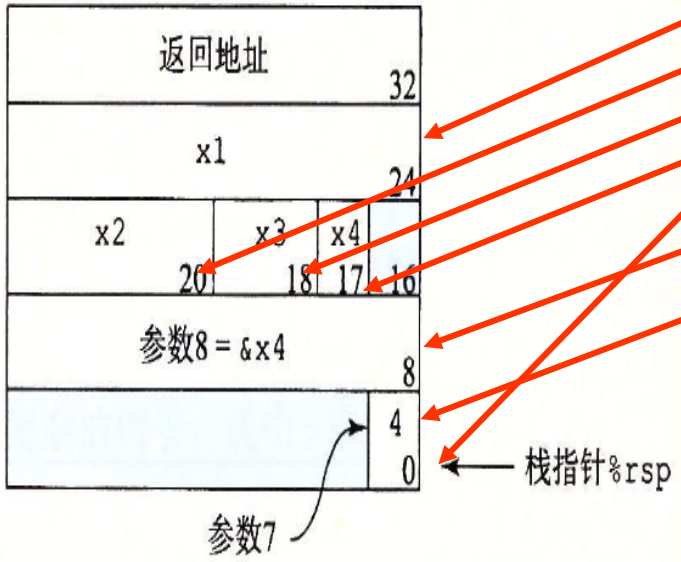


- 局部数据必须存放在内存中的原因：
  - 局部变量太多，寄存器不够存放。
  - 对一个局部变量使用地址运算符&，必须获取该局部变量的地址。
  - 某些局部变量是数组或结构，必须通过数组或结构引用来访问。

局部存储

- 在调用过程proc前：建立栈帧，保存局部变量和参数，参数加载寄存器。

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

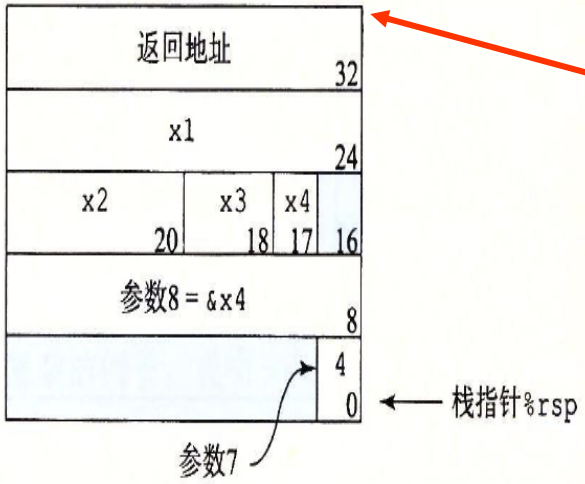



```
long call_proc()
1   call_proc:
      Set up arguments to proc
2       subq    $32, %rsp
3       movq    $1, 24(%rsp)
4       movl    $2, 20(%rsp)
5       movw    $3, 18(%rsp)
6       movb    $4, 17(%rsp)
7       leaq    17(%rsp), %rax
8       movq    %rax, 8(%rsp)
9       movl    $4, (%rsp)
10      leaq    18(%rsp), %r9
11      movl    $3, %r8d
12      leaq    20(%rsp), %rcx
13      movl    $2, %edx
14      leaq    24(%rsp), %rsi
15      movl    $1, %edi
      Call proc
```

局部存储

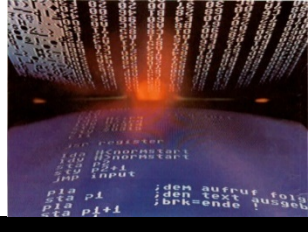
- proc返回后：执行后续代码，取出局部变量，修改栈指针，释放栈帧。

```
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);
}
```




```
                Call proc
16      call    proc
                Retrieve changes to memory
17      movslq  20(%rsp), %rdx
18      addq    24(%rsp), %rdx
19      movswl  18(%rsp), %eax
20      movsbl  17(%rsp), %ecx
21      subl    %ecx, %eax
22      cltq
23      imulq   %rdx, %rax
24      addq    $32, %rsp
25      ret
```

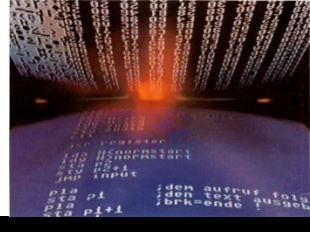
寄存器中的局部存储空间



- 寄存器是唯一被所有过程共享的资源。
- 在某一时刻只有一个过程是活动的
- X86-64制定了寄存器使用惯例

功能	寄存器名称	说明
被调用者保存寄存器	%rbx,%rbp,%r12-%r15	Q使用前，须先保存
调用者保存寄存器	%r10，%r11	P须先保存，Q随意使用
传递参数	%rdi,%rsi,%rdx,%rcx,%r8,%r9	P传递参数给Q
返回值	%rax	Q返回结果给P

寄存器中的局部存储空间



- 函数P两次调用Q，先保护寄存器，再保存参数x，保护返回值

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

```
long P(long x, long y)
x in %rdi, y in %rsi
P:
    pushq   %rbp
    pushq   %rbx
    subq    $8, %rsp
    movq    %rdi, %rbp
    movq    %rsi, %rdi
    call    Q
    movq    %rax, %rbx
    movq    %rbp, %rdi
    call    Q
    addq    %rbx, %rax
    addq    $8, %rsp
    popq    %rbx
    popq    %rbp
    ret
```

保护寄存器

建立栈帧
保护参数x
y保存到第1参数寄存器
保护返回值

释放栈帧

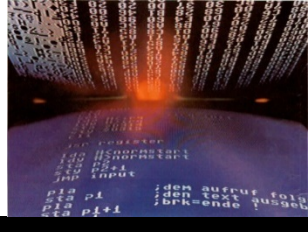
递归过程

- 过程调用在栈中有私有空间，保存返回地址，局部变量，须保护的寄存器等
- x86-64 过程能够递归地调用它们自身。

```
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

```
long rfact(long n)
n in %rdi
1   rfact:
2     pushq   %rbx
3     movq    %rdi, %rbx
4     movl    $1, %eax
5     cmpq    $1, %rdi
6     jle     .L35
7     leaq    -1(%rdi), %rdi
8     call    rfact
9     imulq   %rbx, %rax
10  .L35:
11    popq    %rbx
12    ret
```

有关“过程调用”的讨论



- 从上述代码可以看出，对于double *p=(double *)&a，只是把a的地址直接传送到p所存放的空间，然后把p中的内容，也就是a的地址送到了EAX中，随后用指令“fldl (%eax)”将a的地址处开始的8个字节的机器数（xx…x0000000AH）直接加载到ST(0)中，其中前4个字节xx…x表示R[esp]+0x28，在Linux系统中它应该是一个很大的数，如BFFF…，然后再用指令“fstpl 0x4(%esp)”把ST(0)中的内容（即xx…x0000000AH）作为printf函数的参数送到R[esp]+4的位置，printf(“%lf\n,*p)函数将其作为double类型（%lf）的数打印出来。显然，这个打印的值不会是10.000000，而是一个负数。
- 因为Linux和Windows两种系统所设置的栈底所在地址不同，所以ESP寄存器中的内容不同，因而打印出来的值也肯定不同。通常，Linux中栈底在靠近C0000000H的位置，而在Windows中栈的大致位置是0012FFxxH。因此，可以判断出题目中给出的结果应该是在Windows中执行的结果，打印的值应该是0012 FFxx 0000 000AH 或者 0000 000A 0012 FFxxH对应的double类型的值，前者值为+1.0010….1010×2^{-1022}，后者为+0.0…1… ×2^{-1023}，显然都是接近0的值，正如题目中程序注释所示，结果为0.000000。
- 对于printf(“%lf\n, (double)a) 函数，使用的指令为fildl，该指令先将a作为int型变量（值为10）等值转换为double类型，再加载到ST(0)中。这样再作为double类型（%lf）的数打印时，打印的值就是10.000000。