第五章 子程序设计

张华平 副教授 博士

Email: kevinzhang@bit.edu.cn

Website: http://www.nlpir.org/

@ICTCLAS张华平博士

大数据搜索挖掘实验室（wSMS@BIT）

（1）【重点讲解】子程序基本知识、参数传递
（2）【重点讲解】C语言程序的反汇编
（3）【一般性讲解】子程序特殊应用、模块化程序设计、混合编程

5.1.1、子程序定义

在汇编语言中用过程定义伪指令定义子程序。过程定义伪指令格式：

过程名　　PROC　属型

…

过程名　　ENDP

设计子程序时应注意的问题

1．寄存器的保存与恢复

为了保证调用程序的寄存器内容不被破坏，应在子程序开头保存它要用到的寄存器内容，返回前再恢复它们。

2. 注意堆栈状态

在设计含有子程序的程序时，要密切注意堆栈的变化。这包括要注意一切与堆栈有关的操作。例如CALL调用类型和子程序定义类型的一致性，PUSH和POP指令的匹配，通过堆栈传递参数时子程序返回使用RET n指令等，以确保堆栈平衡。

3. 子程序说明

为便于引用，子程序应在开头对其功能、调用参数和返回参数等予以说明，例如参数的类型、格式及存放位置等。

5.1.2 堆　　栈

- 所谓堆栈，就是供程序使用的一块连续的内存空间，一般用于保存和读取临时性的数据。

1. 堆栈特点

- 1. 临时性
- 2. 快速性
- 3. 动态扩展性

2. 堆栈用途

- 1. 保护和恢复调用现场
- 2. 用于变量之间的数据传递
- 3. 用做临时的数据区
- 4. 子程序的调用和返回

1. 保护和恢复调用现场

```
PUSH    EAX
PUSH    EBX
……
POP     EBX
POP     EAX
```

2. 用于变量之间的数据传递

```
PUSH        Var1
POP   Var2
```

下面的两条指令交换两个变量Var1和Var2的值。

```
PUSH        Var1
PUSH        Var2
POP   Var1   ;Var1中现在的值是原先Var2的值
POP   Var2          ;Var2中现在的值是原先Var1的值
```

3. 用做临时的数据区

例5.1 将EAX中的内容转换为十进制字符串

```
szStr     BYTE          10 DUP (0)
          MOV   EAX, 8192
          XOR   EDX, EDX
          XOR   ECX, ECX
          MOV   EBX, 10
a10:
          DIV   EBX                    ;EDX:EAX除以10
          PUSH  EDX              ;余数在EDX中, EDX压栈
          INC   ECX              ;ECX表示压栈的次数
          XOR   EDX, EDX         ;EDX:EAX=下一次除法的被除数
          CMP   EAX, EDX         ;被除数=0?
          JNZ   a10              ;如果被除数为0，不再循环
          MOV   EDI, OFFSET szStr
a20:
          POP   EAX              ;从堆栈中取出商
          ADD   AL, '0'          ;转换为ASCII码
          MOV   [EDI], AL        ;保存在szStr中
          INC   EDI
          LOOP  a20              ;循环处理
          MOV   BYTE PTR [EDI], 0
```

4. 子程序的调用和返回

1. 在调用子程序时，CALL指令自动在堆栈中保存其返回地址；
2. 从子程序返回时，RET指令从堆栈中取出返回地址。
3. 子程序中的局部变量也放在堆栈中。子程序执行过程中，这些局部变量是可用的；
4. 主程序还可以将参数压入堆栈，子程序从堆栈中取出参数。

5.1.3 子程序的返回地址

- 例. 段内调用和返回
- 设计两个子程序：第1个子程序AddProc1使用ESI和EDI作为加数，做完加法后把和放在EAX中；第2个子程序AddProc2使用X和Y作为加数，做完加法后把和放在Z中。主程序先后调用两个子程序，最后将结果显示出来。
- 在AddProc2中用到了EAX，所以要先将EAX保存在堆栈中，返回时再恢复EAX的值。否则EAX中的值会被破坏。
- 见程序PROG0501.ASM 。

```
.386
.model flat,stdcall
option casemap:none
includelib      msvcrt.lib
printf          PROTO C     :dword,:vararg
.data
szFmt           byte        '%d + %d=%d', 0ah, 0        ;输出结果格式字符串
x               dword       ?
y               dword       ?
z               dword       ?
```

```
AddProc1    proc    ;使用寄存器作为参数
        mov         eax, esi                ;EAX=ESI + EDI
            add             eax, edi
            ret
AddProc1    endp
AddProc2    proc                    ; 使用变量作为参数
            push    eax             ; C=A + B
            mov     eax, x
            add     eax, y
            mov     z, eax
        pop     eax                 ;恢复EAX的值
            ret
AddProc2    endp
```

```
start:
        mov   esi, 10
              mov   edi, 20                          ;为子程序准备参数
              call    AddProc1                 ;调用子程序
                                        ;结果在EAX中
              mov   x, 50
              mov   y, 60                      ;为子程序准备参数
              call    AddProc2                 ;调用子程序
                                        ;结果在Z中
              invoke          printf, offset szFmt,
              esi, edi, eax                    ;显示第1次加法结果
              invoke          printf, offset szFmt,
                    x, y, z                    ;显示第2次加法结果
        ret
end
```

可以通过给子程序传递参数使其更通用。常用的参数传递方法如下：

1. 通过寄存器传递；
2. 通过数据区的变量传递
3. 通过堆栈传递

5.2.1 C语言函数的参数传递方式

- 在C/C++以及其他高级语言中，函数的参数是通过堆栈来传递的。C语言中的库函数，以及Windows API等也都使用堆栈方式来传递参数。

C函数常见的有5种参数传递方式（调用规则）见下表。

调用规则	参数入栈顺序	参数出栈	说　明
cdecl方式	从右至左	主程序	参数个数可动态变化
stdcall方式	从右至左	子程序	Windows API常使用
fastcall方式	用ECX、EDX传递第1、2个参数，其余的参数同stdcall，从右至左	子程序	常用于内核程序
this方式	ECX等于this，从右至左	子程序	C++成员函数使用
naked方式	从右至左	子程序	自行编写进入/退出代码

```
//PROG0502.c
int subproc(int a, int b)
{
   return a-b;
}
int r,s;
int main( )
{
   r=subproc(30, 20);
   s=subproc(r, -1);
}
```

cdecl方式

```
00401000    PUSH    EBP
00401001    MOV     EBP,ESP
00401003    MOV     EAX,DWORD PTR [EBP+8]
00401006    SUB     EAX,DWORD PTR [EBP+0CH]
00401009    POP     EBP
0040100A    RET
```

```
0040100B    PUSH    EBP
0040100C    MOV     EBP,ESP
0040100E    PUSH    14H
00401010    PUSH    1EH
00401012    CALL    00401000
00401017    ADD     ESP,8
0040101A    MOV     [00405428],EAX
0040101F    PUSH    0FFFFFFFFH
00401021    MOV     EAX,[00405428]
00401026    PUSH    EAX
00401027    CALL    00401000
0040102C    ADD     ESP,8
0040102F    MOV     [0040542C],EAX
00401034    POP     EBP
00401035    RET
```

2. stdcall方式

堆栈的平衡是由子程序来完成的，子程序使用“RET n”指令

Windows API采用的调用规则就是stdcall方式。

```
WINBASEAPI int WINAPI lstrcmpA(LPCSTR lpStr1, LPCSTR lpStr2);
```

其中的WINAPI定义为：

```
#define WINAPI   __stdcall
```

将subproc（ ）设置为使用__stdcall调用规则：

```
int _stdcall subproc(int a, int b)
```

3. fastcall方式

这种方式和stdcall类似。区别是它使用ECX传递第1个参数，EDX传递第2个参数。其余的参数采用从右至左的顺序入栈，由子程序在返回时平衡堆栈。例如：

```
int _fastcall addproc(int a, int b, int c, int d)
```

4. this方式

这种方式和stdcall类似，在C++类的成员函数中使用。它使用ECX传递this指针，即指向对象。

naked方式

前面4种方式中，编译器自动为函数生成进入代码和退出代码。进入代码的形式为：

```
00401000    PUSH        EBP
00401001    MOV         EBP, ESP
```

退出代码的形式为：

```
00401009    POP         EBP
0040100A    RET         8
```

由编程者自行编写函数内的所有代码，可以使用naked调用规则。

例5.4　使用naked调用规则，使编译器不会为函数subproc生成进入代码和退出代码。

```
//PROG0503.c
__declspec(naked)  int subproc(int a, int b)
{
        _asm mov eax, [esp+4]
        _asm sub eax, [esp+8]
        _asm ret
}
```

5.2.2 汇编语言子程序的参数传递方式

```
SubProc1    proc                ;使用堆栈传递参数
            push    ebp
            mov     ebp,esp
            mov     eax,dword ptr [ebp+8]     ;取出第1个参数
            sub     eax,dword ptr [ebp+12]    ;取出第2个参数
            pop     ebp
            ret
SubProc1    endp
```

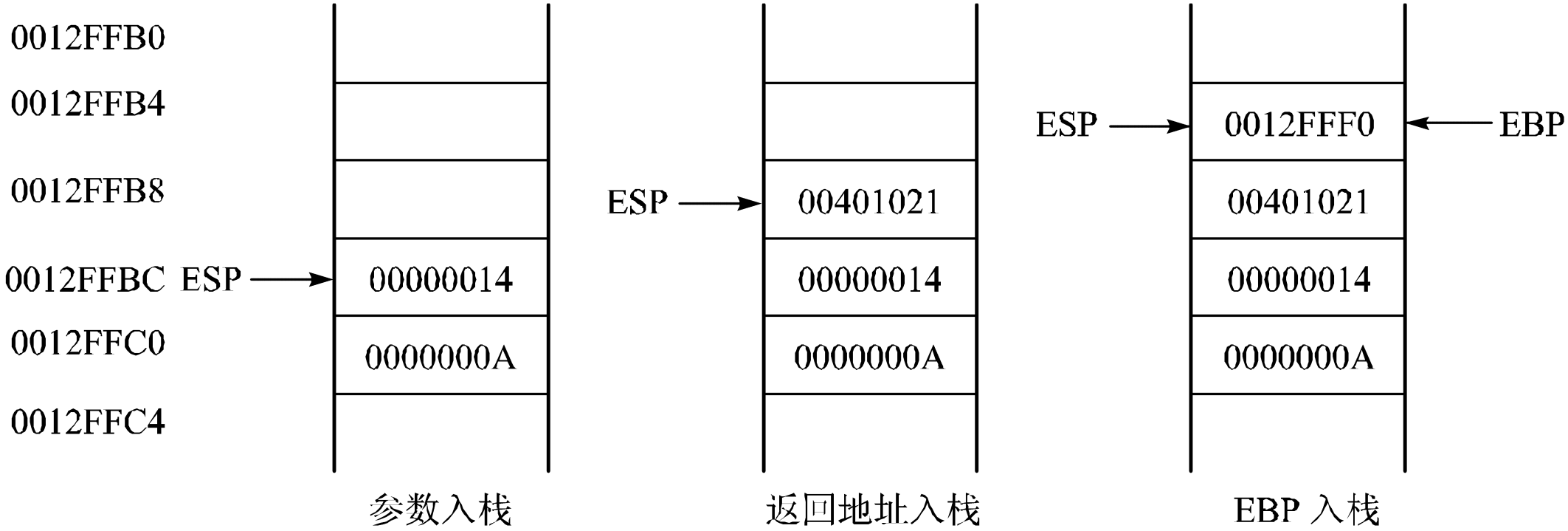
5.2.2 汇编语言子程序的参数传递方式

```
SubProc2    proc                                ;使用堆栈传递参数
            push    ebp
            mov     ebp,esp
            mov     eax,dword ptr [ebp+8]       ;取出第1个参数
            sub     eax,dword ptr [ebp+12]      ;取出第2个参数
            pop     ebp
            ret     8                           ;平衡主程序的堆栈
SubProc2    endp
```

5.2.2 汇编语言子程序的参数传递方式

```
start:
        push    10              ;第2个参数入栈
        push    20              ;第1个参数入栈
        call    SubProc1        ;调用子程序
        add     esp, 8
        push    100             ;第2个参数入栈
        push    200             ;第1个参数入栈
        call    SubProc2        ;调用子程序
        ret
end       start
```

5.2.2 汇编语言子程序的参数传递方式



5.2.3 带参数子程序的调用

```
SubProc1    proc    C  a:dword, b:dword                ; 使用C规则
        mov         eax, a                          ; 取出第1个参数
        sub         eax, b                          ; 取出第2个参数
        ret                                 ; 返回值=a-b
SubProc1    endp
SubProc2    proc    stdcall a:dword, b:dword            ; 使用stdcall规则
        mov         eax, a                          ; 取出第1个参数
        sub     eax, b                              ; 取出第2个参数
        ret                                 ; 返回值=a-b
SubProc2    endp
```

5.2.3 带参数子程序的调用

```
start:
        invoke   SubProc1, 20, 10
         invoke        printf, offset szMsgOut, 20, 10, eax

        invoke   SubProc2, 200, 100
        invoke   printf, offset szMsgOut, 200, 100, eax
        ret
        end      start
```

~~invoke SubProc1, r*2, 30~~

5.2.4 子程序中的局部变量

局部变量只供子程序内部使用，使用局部变量能提高程序的模块化程度，节约内存空间。局部变量也被称为自动变量。

在高级语言中，局部变量的实现原理如下。

（1）在进入子程序的时候，通过修改堆栈指针ESP来预留出需要的空间。用SUB ESP，x指令预留空间，x为该子程序中所有局部变量使用的空间。

（2）在返回主程序之前，通过恢复ESP来释放这些空间，在堆栈中不再为子程序的局部变量保留空间。

5.2.4 子程序中的局部变量

LOCAL伪指令的格式为：

LOCAL变量名1[重复数量][:类型], 变量名2[重复数量][:类型]……

LOCAL伪指令必须紧接在子程序定义的伪指令PROC之后

```
LOCAL   TEMP[3]:DWORD
LOCAL   TEMP1, TEMP2:DWORD
```

5.2.4 子程序中的局部变量

```
swap    proc    C  a:ptr dword, b:ptr dword        ;使用堆栈传递参数
        local   temp1,temp2:dword
        mov     eax, a
        mov     ecx, [eax]
        mov     temp1, ecx              ;temp1=*a
        mov     ebx, b
        mov     edx, [ebx]
        mov     temp2, edx              ;temp2=*b
        mov     ecx, temp2
        mov     eax, a
        mov     [eax], ecx              ;*a=temp2
        mov     ebx, b
        mov     edx, temp1
        mov     [ebx], edx      ;*b=temp1
        ret
swap    endp
```

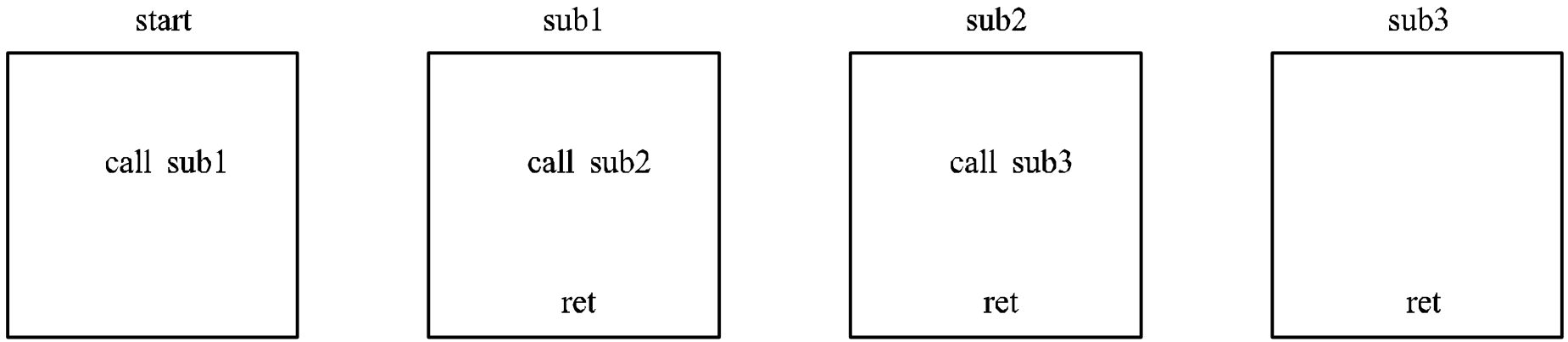
5.2.4 子程序中的局部变量

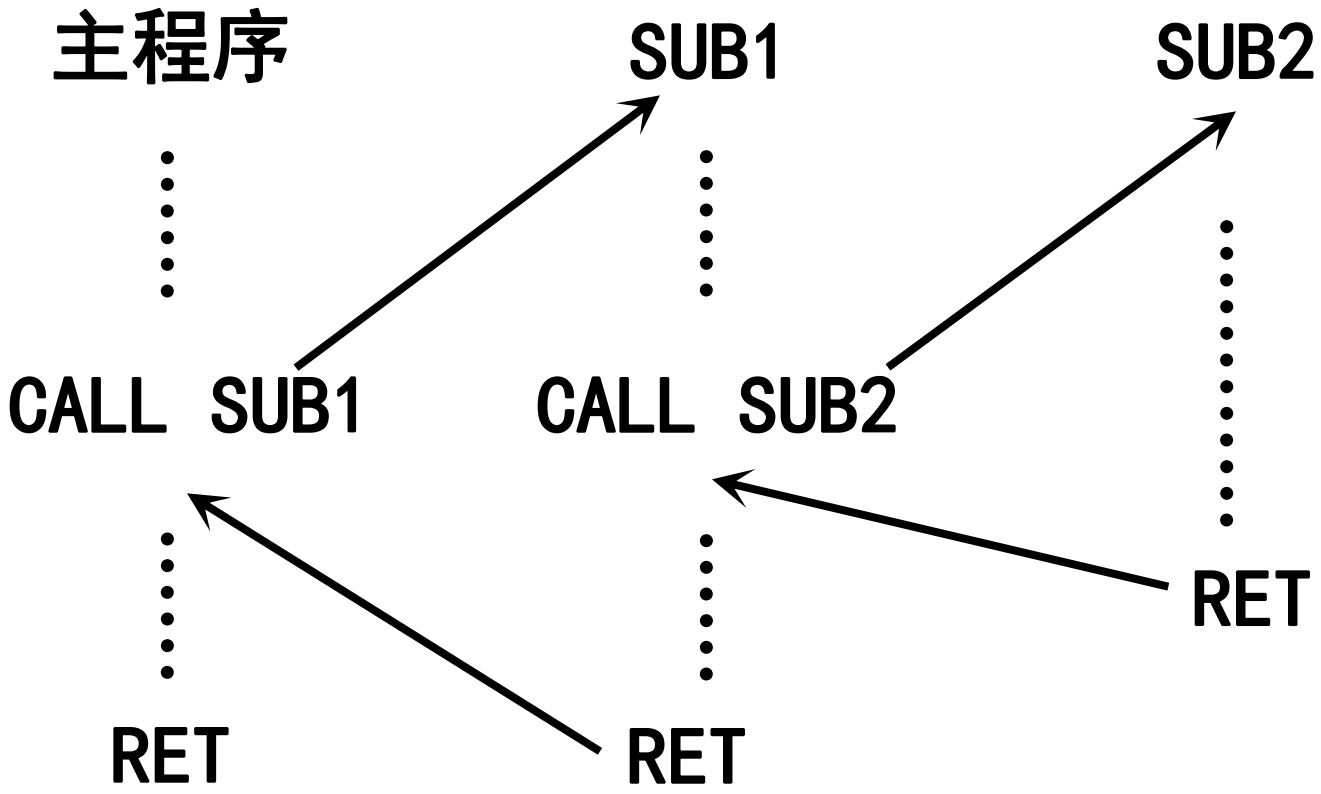
```
start       proc
            invoke printf, offset szMsgOut, r, s
            invoke swap, offset r, offset s
            invoke printf, offset szMsgOut, r, s
            ret
start       endp
end         start
```

5.3.1、子程序嵌套

在汇编语言中，允许子程序作为调用程序去调用另一子程序，把这种关系称为子程序嵌套。

3层子程序嵌套示意图。嵌套的层数没什么限制，其层数称为嵌套深度。





子程序嵌套示意图

返回

由于子程序嵌套对堆栈的使用很频繁，因此还要确保堆栈有足够空间，并要注意堆栈的正确状态，这包括CALL、RET、RET N、PUSH、POP、INT、IRET等与堆栈操作有关指令的正确使用。

5.3.2、子程序递归

```
factorial       proc    C  n:dword
        cmp     n, 1
        jbe     exitrecurse
        mov     ebx, n                  ;EBX=n
        dec     ebx             ;EBX=n–1
        invoke  factorial, ebx          ;EAX=(n–1)!
        imul    n               ;EAX=EAX * n
        ret                     ;=(n–1)! * n=n!
exitrecurse:
        mov     eax, 1                  ;n=1时, n!=1
            ret
factorial       endp
```

```
start       proc
        local   n,f:dword
        mov     n, 5
        invoke  factorial,n;EAX=n!
        mov     f, eax
        invoke  printf, offset szOut, n, f
        ret
```

5.3.3 缓冲区溢出

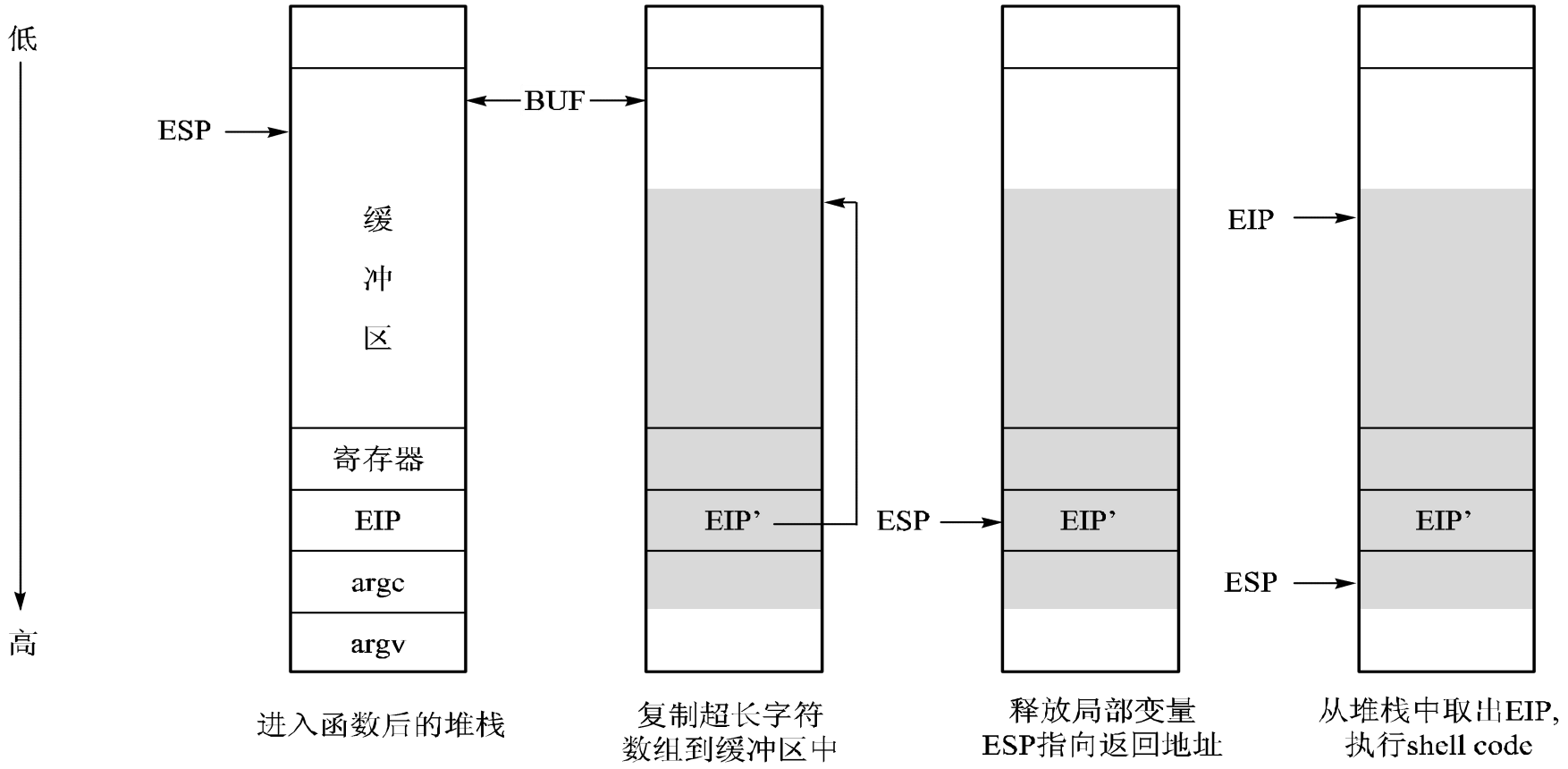
- 缓冲区溢出是目前最常见的一种安全问题，操作系统以及应用程序一般都存在缓冲区溢出漏洞。缓冲区溢出是由编程错误引起的，当程序向缓冲区内写入的数据超过了缓冲区的容量，就发生了缓冲区溢出，缓冲区之外的内存单元被程序“非法”修改。
- 一般情况下，缓冲区溢出会导致应用程序的错误或者运行中止，但是，攻击者利用程序中的漏洞，精心设计出一段入侵程序代码，覆盖缓冲区之外的内存单元，这些程序代码就可以被CPU所执行，从而获取系统的控制权。

1　堆栈溢出

在一个程序中，会声明各种变量。静态全局变量位于数据段并且在程序开始运行时被初始化，而局部变量则在堆栈中分配，只在该函数内部有效。

如果局部变量使用不当，会造成缓冲区溢出漏洞。

```
int main(int argc, char **argv)
{
   char buf  [80];
   strcpy(buf, argv[1]);
}
```



2　数据区溢出

当变量或数组位于数据区时，由于程序对变量、数组的过度使用而导致对其他数据单元的覆盖，也可能导致程序执行错误。

```
;PROG0508.asm
.386
.model flat,stdcall
includelib          msvcrt.lib
printf              PROTO C:dword,:vararg
scanf     PROTO C:dword,:vararg
.data
szMsg               byte      'f is called. buf=%s', 0ah, 0
szFormat            byte      '%s', 0
buf       byte      40 dup (0)
fn        dword     offset f
.code
f         proc
          invoke    printf, offset szMsg, offset buf
          ret
f         endp
start:
          invoke    scanf, offset szFormat, offset buf
          call      dword ptr [fn]
invalidarg:
          ret
end       start
```

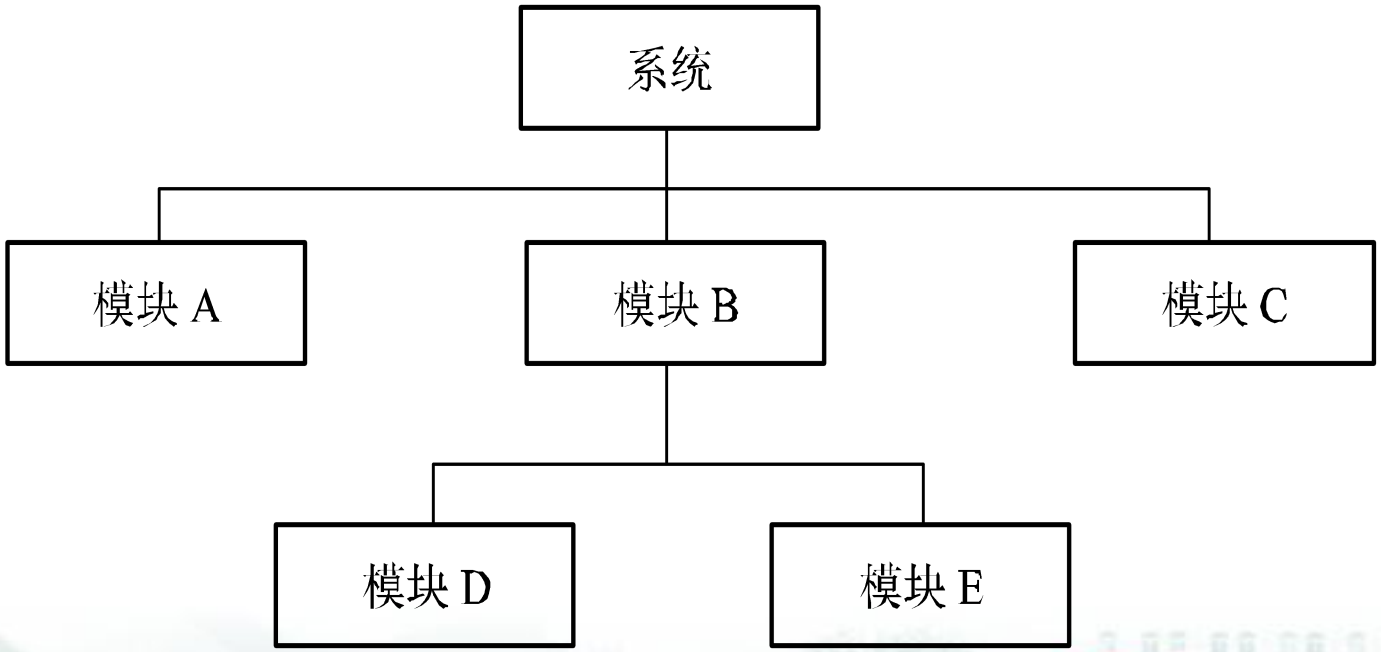
当输入的字符长度超过40个字节以后，后面的fn就被覆盖。

执行“call dword ptr [fn]”指令时，从fn单元中取出的内容就不再是子程序f的地址

5.4 模块化程序设计

- 如果有多个源程序文件，或者需要使用C/C++、汇编等多种语言混合编程，就需要对这些源程序分别编译，最后连接构成一个可执行文件。

如图所示，系统由模块A、模块B、模块C组成，而模块B中的部分功能又可以进一步分解成为模块D、模块E，整个系统包括了5个模块。模块中的代码设计为子程序，能够相互进行调用。



- 在子程序设计中，主程序和子程序之间可以通过全局变量、寄存器、堆栈等方式传递数据，这种技术在模块化程序设计中同样适用。

5.4.2 模块间的通信

- 由于各个模块需要单独汇编，于是就会出现当一个模块通过名字调用另一模块中的子程序或使用其数据时，这些名字对于调用者来讲是未定义的，因此在汇编过程中就会出现符号未定义错误。可以通过伪指令EXTRN、PUBLIC等来解决。

1. 外部引用伪指令EXTRN

- 格式：EXTRN　变量名:类型 [,...]

功能：说明在本模块中用到的变量是在另一个模块中定义的，同时指出变量的类型。

2. 全局符号说明伪指令PUBLIC

- 格式：PUBLIC 名字 [,...]

功能：告诉汇编程序本模块中定义的名字可以被其他模块使用。这里的名字可以是变量名，也可以是子程序名。

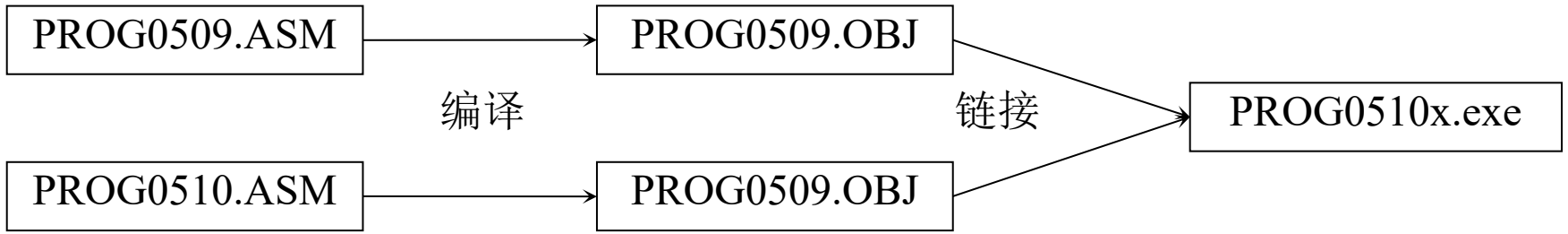
3. 子程序声明伪指令PROTO

- 格式：子程序名 PROTO [C | stdcall] :[第一个参数类型] [,:后续参数类型]

功能：说明子程序的名字和参数类型，供主程序调用。在前面的程序中，已经多次使用这种方式调用C语言的库函数及Windows的API。

```
;PROG0509.asm
.386
.model          flat,stdcall
option          casemap:none
includelib      msvcrt.lib
printf          PROTO C :dword,:vararg
SubProc         PROTO stdcall :dword, :dword   ; SubProc位于其他模块中
public          result                                  ;允许其他模块使用result
.data
szOutputFmtStr  byte    '%d?%d=%d', 0ah, 0      ;输出结果
oprd1           dword   70                      ;被减数
oprd2           dword           40                      ;减数
result          dword           ?                       ;差
.code
main            proc            C argc, argv
        invoke          SubProc, oprd1, oprd2           ;调用其他模块中的函数
        invoke          printf, offset szOutputFmtStr, \        ;输出结果
                        oprd1, \
                        oprd2, \
                        result                          ;result由SubProc设置
                ret
main    endp
        end
```

```
;PROG0510.asm
.386
.model flat,stdcall
public          SubProc                     ;允许其他模块调用SubProc
extrn           result:dword                ;result位于其他模块中
.data
.code
SubProc         proc    stdcall a, b        ;减法函数, stdcall调用方式
                mov     eax, a          ;参数为a,b
                  sub   eax, b              ;EAX=a-b
                mov     result, eax         ;减法的结果保存在result中
                ret     8                               ;返回a-b
SubProc         endp
                  end
```



ml /c /coff prog0509.asm
ml /c /coff prog0510.asm
link prog0509.obj prog0510.obj /out:prog0510x.exe /subsystem:console

5.5 C语言模块的反汇编

- 汇编语言的一个重要应用就是程序的底层分析，学会阅读反汇编程序，通过逆向分析将反汇编程序写成高级语言如C语言等代码格式，在实际工程应用中具有重要意义。

- 对高级语言底层实现细节的分析，能够了解程序的实现机理，对编写高效率的程序也有很大帮助。

5.5.1 基本框架

```
//PROG0511.c
 1:     #include "stdio.h"
 2:     int main( )
 3:     {
 4:     return 0;
 5:     }
```

```
00401020   55                push    ebp
00401021   8B EC             mov     ebp,esp
00401023   83 EC 40 sub              esp,40h
00401026   53                push    ebx
00401027   56                push    esi
00401028   57                push    edi
00401029   8D 7D C0lea               edi,[ebp-40h]
0040102C   B9 10 00 00 00            mov     ecx,10h
00401031   B8 CC CC CC CC            mov     eax,0CCCCCCCCh
00401036   F3 AB     rep stos dword ptr [edi];以上为栈初始化过程。
00401038   33 C0     xor     eax,eax; 返回值0保存在eax中。
0040103A   5F                pop     edi
0040103B   5E                pop     esi
0040103C   5B                pop     ebx
0040103D   8B E5             mov     esp,ebp
0040103F   5D                pop     ebp;以上为栈初恢复过程。
00401040   C3                ret
```

5.5.1 基本框架

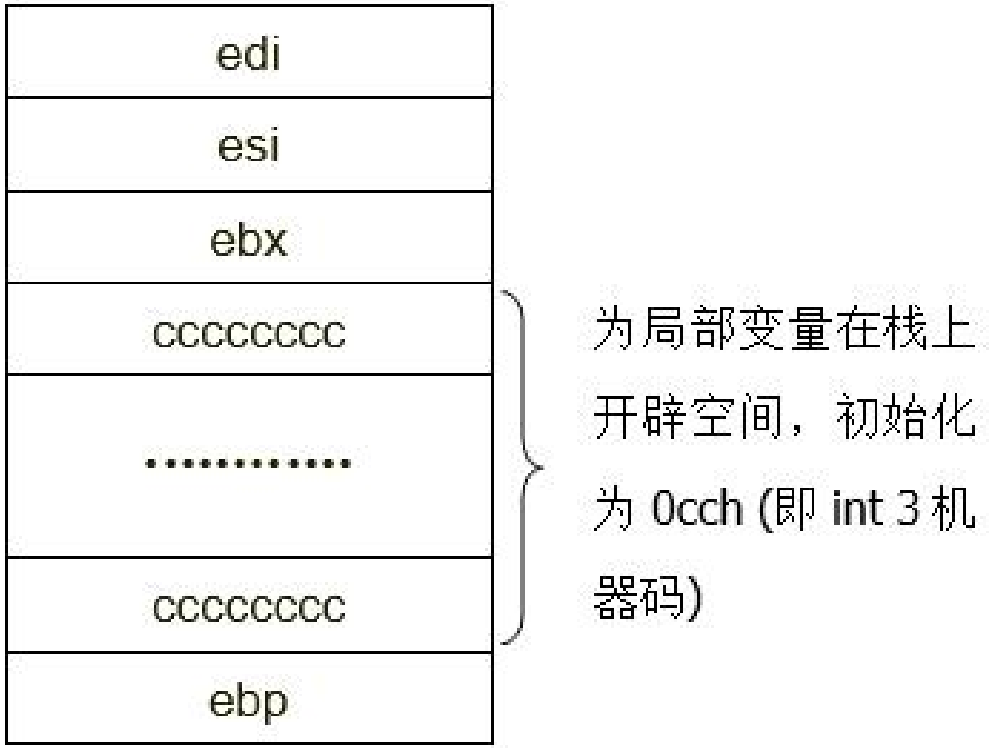


图 5-6 局部变量存储空间

```
00401049 83 7D FC 00    cmp     dword ptr [ebp-4],0
0040104D 7C 0F          jl      main+3Eh (0040105e)
0040104F 68 84 0F 42 00 push    offset string "i is nonnegative!" (00420f84)
00401054 E8 87 00 00 00 call    printf (004010e0)
00401059 83 C4 04               add     esp,4
0040105C EB 0D          jmp     main+4Bh (0040106b)
0040105E 68 74 0F 42 00 push    offset string "i is negative!" (00420f74)
00401063 E8 78 00 00 00 call    printf (004010e0)
00401068 83 C4 04               add     esp,4
```

```
1:      int i;
2:      if(i>=0)
3:          printf("i is nonnegative!");
4:      else
5:          printf("i is negative!");
```

5.5.3 循环结构

```
00401038      C7 45 FC 01 00 00 00      mov       dword ptr [ebp-4],1
;局部变量i保存在栈中，通过[ebp-4]的方式访问。
0040103F      EB 09                     jmp       main+2Ah (0040104a)
00401041      8B 45 FC                  mov       eax,dword ptr [ebp-4]
00401044      83 C0 01                  add       eax,1
00401047      89 45 FC                  mov       dword ptr [ebp-4],eax
0040104A      83 7D FC 0A               cmp       dword ptr [ebp-4],0Ah
0040104E      7F 02                     jg        main+32h (00401052)
00401050      EB EF                     jmp       main+21h (00401041)
```

```
1:      int i;
2:      for(i=1;i<=10;i++)
3:              ;
```

5.5.4 变量定义

```
//PROG0512.c
1:      #include "stdio.h"
2:      int i1;  //全局变量
3:      static int i2;//静态
全局变量
4:      int main( )
5:      {
6:      int i3;//局部变量
7:      i1=0;
8:      i2=0;
9:      i3=0;
10:     return 1;
11:     }
```

```
00401028 C7 05 B8 27 42 00 00
        mov     dword ptr [_i1 (004227b8)],0;全局变量i1
00401032 C7 05 D8 25 42 00 00
        mov     dword ptr [i2 (004225d8)],0;静态变量i2
0040103C C7 45 FC 00 00 00 00
        mov     dword ptr [ebp-4],0;局部变量i3
00401043 B8 01 00 00 00
        mov     eax,1;返回值保存在eax
0040104E C3                 ret
```

```
//PROG0513.c
1:      #include "stdio.h"
2:      int main( )
3:      {
4:      int *p,a;
5:      a=10;
6:      p=&a;
7:      }
```

反汇编码如下所示。

```
00401028      C7 45 F8 0A 00 00 00
          mov           dword ptr [ebp-8],0Ah
          ;a=10，a为局部变量，通过[ebp-n]的方式访问。
0040102F      8D 45 F8
          lea           eax,[ebp-8]
00401032      89 45 FC
          mov           dword ptr [ebp-4],eax
; p=&a，p为局部变量，p中保存着a的地址。
```

```
//PROG0514.c
1:      #include "stdio.h"
2:      int subproc(int a, int b)
3:      {
4:          return a*b;
5:      }
```

子程序subproc的反汇编码如下所示。

```
;以上为栈的初始化略
00401028 8B 45 08 mov       eax,dword ptr [ebp+8]
0040102B 0F AF 45 0C imul   eax,dword ptr [ebp+0Ch]
;eax = a*b;返回值保存在eax中
;以上为栈的恢复，略
00401035 C3                 ret
```

5.5.6 函数

```
6:      int main( )
7:      {
8:          int r,s;
9:          r=subproc(10, 8);
10:         s=subproc(r, -1);
11:printf("r=%d,s=%d",r,s);
12:     }
```

主程序main反汇编码如下所示。

```
00401005    E9 66 A4 00 00    jmp          main (0040b470)
0040100A    E9 01 00 00 00    jmp          subproc (00401010)
……栈初始化（略）……
0040B488    6A 08             push         8
0040B48A    6A 0A             push         0Ah
0040B48C    E8 79 5B FF FF    call         @ILT+5(_subproc) (0040100a)
0040B491    83 C4 08          add          esp,8
;第一次函数调用
0040B494    89 45 FC          mov          dword ptr [ebp-4],eax
0040B497    6A FF             push         0FFh
0040B499    8B 45 FC          mov          eax,dword ptr [ebp-4]
0040B49C    50          push              eax
0040B49D    E8 68 5B FF FF    call         @ILT+5(_subproc) (0040100a)
0040B4A2    83 C4 08          add          esp,8
;第二次函数调用
```

```
11:printf("r=%d,s=%d",r,s);

    0040B4A5    89 45 F8          mov         dword ptr [ebp-8],eax
    0040B4A8    8B 4D F8          mov         ecx,dword ptr [ebp-8]
    0040B4AB    51                push        ecx
    0040B4AC    8B 55 FC          mov    edx,dword ptr [ebp-4]
    0040B4AF    52            push          edx
    0040B4B0     68 50 FE 41 00    push    offset string "r=%d,s=%d"
(0041fe50)
    0040B4B5    E8 76 02 00 00    call        printf (0040b730)
    0040B4BA    83 C4 0C          add         esp,0Ch
    ;输出结果
```

5.6 C语言和汇编语言的混合编程

要求执行速度快、占用空间小、要求直接控制硬件等场合，仍然要用到汇编语言程序，在这种情况下，使用汇编语言编程是程序设计人员的最好选择

在C程序中直接嵌入汇编代码，或者由C语言主程序调用汇编子程序。

其格式为：

```
_asm      汇编语句
```

对于连续的多个汇编语句，可以采用下面的形式：

```
_asm {
        汇编语句
        汇编语句
        …
}
```

内嵌汇编语句的操作码必须是有效的80x86指令。不能使用BYTE、WORD、DWORD等语句定义数据。

5.6.2 C程序调用汇编子程序

1.C模块使用汇编模块中的变量

C变量类型	汇编变量类型	大小
Char	SBYTE	1字节
short	SWORD	2字节
int	SDWORD	4字节
long	SDWORD	4字节
unsigned char	BYTE	1字节
unsigned short	WORD	2字节
unsigned int	DWORD	4字节
unsigned long	DWORD	4字节
指针	DWORD	4字节

在汇编模块中，用**PUBLIC**语句允许外部模块来访问这些变量。例如：

```
        public  _a, _b
_a  sdword  3
_b   sdword  4
```

```
extern int a, b;
```

2. 汇编模块使用C模块中的变量

在C模块中应该用extern来指明这些变量可以由外部模块所使用。例如：

```
extern int z;
int z;
```

在汇编模块中，要使用这些变量，需要EXTRN加以说明。之后，在汇编模块中就可以访问C模块中的变量了。即：

```
extrn _z:sdword
mov       _z, esi
```

3. C模块调用汇编模块中的子程序

汇编模块中的语句以子程序的形式编写，相当于C语言的一个函数。

在C模块中，使用extern表明这个函数来自于外部模块，同时说明它的参数类型及返回值类型，例如：

```
extern int CalcAXBY(int x, int y);
```

之后，就可以在C模块中调用汇编模块中的子程序：

```
int r=CalcAXBY(x, y);
```

CalcAXBY函数把返回值存入EAX中。

5.6.2 C程序调用汇编子程序

```
//PROG0515.c
#include "stdio.h"
extern int a, b;
extern int CalcAXBY(int x, int y);
extern int z;
int z;
int x=10, y=20;
int main()
{
        int r=CalcAXBY(x, y);
        printf("%d*%d+%d*%d=%d, r=%d\n", a, x, b, y, z, r);
        return 0;
}
```

5.6.2 C程序调用汇编子程序

```
;PROG0516.asm
public  _a, _b                          ;允许a,b被C模块所使用
extrn   _z:sdword                               ;_z在C模块中
.data   _a          sdword   3          _b          sdword   4
.code
CalcAXBY        proc                    C x:sdword, y:sdword
            push    esi                 push    edi                         ;必须保存在堆栈中
            mov     eax, x                      ;x在堆栈中
            mul     _a                  ;a*x → EAX
            mov             esi, eax                    ;a*x → ESI
            mov             eax, y              ;y在堆栈中
            mul     _b                  ;b*y → EAX
            mov             edi, eax                    ;a*x+b*y → ECX
            add     esi, edi                    ;a*x+b*y → ECX
            mov     _z, esi                     ;a*x+b*y → _z
            mov     eax, 0              ;函数返回值设为0
            pop     edi                 ;恢复EDI
            pop     esi                         ;恢复ESI
            ret
CalcAXBY        endp
            end
```

5.6.2 C程序调用汇编子程序

```
cl /c prog0515.c
ml /c /coff prog0516.asm
link prog0515.obj prog0516.obj /out:prog0516.exe /subsystem:console
```

5.6.3 汇编调用C函数

```
;PROG0517.asm
.386
.model flat
input       PROTO C px:ptr sdword, py:ptr sdword
output      PROTO C x:dword, y:dword
.data
x           dword   ?
y           dword   ?
.code
main          proc    C
            invoke  input, offset x, offset y
            invoke  output, x, y
            ret
main          endp
end
```

```
//PROG0518.c
#include "stdio.h"
extern void input(int *px, int *py);
extern void output(int x, int y);
void input(int *px, int *py)
{
        printf("input x y: ");
        scanf("%d %d", px, py);
}
void output(int x, int y)
{
        printf("%d*%d+%d*%d=%d\n", x, x, y, y, x*x+y*y);
}
```

5.6.4 C++与汇编的联合编程

对于C++与汇编的联合编程，在汇编模块一方并没有特殊的要求。
在C++一方，则应将与汇编模块共享的变量、函数等用extern “C”的形式说明。
例如：在汇编模块中实现了_ArraySum2、_ArraySum3子程序，要在C++模块中调用，就要使用以下两个语句来说明：

```
extern "C" int _cdecl ArraySum2(int array[ ], int count);
extern "C" int _stdcall ArraySum3(int array[ ], int count);
```

如果汇编模块要使用C++模块的initvals数组，同样需要用extern “C”说明：

```
extern "C" int initvals[ ];
```

感谢关注聆听！

张华平
Email: kevinzhang@bit.edu.cn
微博：@ICTCLAS张华平博士
实验室官网：
http://www.nlpir.org

大数据千人会