汇编与接口技术
第五章
子程序设计

子程序定义

- 使用过程定义伪指令PROC定义子程序

- 子程序名 PROC [类型]
- ……
- RET
- 子程序名 ENDP

- 在Win32汇编语言中，PROC后面还可以跟其他参数
- 子程序一定要在代码段中定义，在子程序结束时，要用RET指令返回主程序。

- 使用CALL指令来调用子程序，其中子程序名可以通过直接或间接方法给出
- 与子程序类型相对应，CALL指令分为有段内和段间调用
- 主程序和子程序可以在同一个代码段也可以在不同代码段。由于32位汇编程序的内存模式为FLAT，一个段长可达4GB，所以本节主要讨论段内调用。如果设计实模式程序，则应注意一个段长不能超过64KB

设计子程序时应注意的问题

- 1. 寄存器的保存与恢复
- 2. 保持堆栈平衡
- 3. 子程序说明

堆栈的用途

- 16位程序中以字为单位进行，32位程序中以双字为单位进行
- （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
- 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
- PUSH　Count
- ……
- POP　Count

将EAX中的内容转换为十进制字符串

- 对这个数连续除以10，直到所得的商为0结束。每次除法得到的商加上‘0’就可以转换为ASCII字符。
- 第1次除法所得的余数是最低位，应该放在最后面，作为字符串的最后一个字符；最后1次除法所得的余数是最高位，应该作为字符串的第一个字符。

```
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
```

返回地址作为一个双字压栈

- 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。

```
.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          ?
```

```
.code
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     start
```

参数传递

- 在主程序和子程序中传递参数，通常有3种方法：通过寄存器传递、通过数据区的变量传递、通过堆栈传递。
- 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。例如：MessageBox就属于Windows API函数，而printf、scanf属于C的库函数。

C函数常见的有5种参数传递方式

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

cdecl方式

- cdecl方式是C函数的默认方式，不加说明时，函数就使用cdecl调用规则。
- 设计一个通过堆栈传递函数参数的C程序。函数subproc( )有两个整型参数，参数名为a和b。函数的功能是计算a-b，减法的结果作为函数的返回值。

cdecl方式

- （1）使用堆栈传递参数。
- （2）主程序按从右向左的顺序将参数逐个压栈。最后一个参数先入栈。每一个参数压栈一次，在堆栈中占4字节。
- （3）在子程序中，使用[EBP+X]的方式来访问参数。X=8代表第1个参数；X=12代表第2个参数，依次类推。
- （4）子程序用RET指令返回。
- （5）由主程序执行“ADD ESP, N”指令调整ESP，达到堆栈平衡。N等于参数个数乘以4。每个参数在堆栈中占4字节。
- （6）子程序的返回值放在EAX中。

C程序

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

编译后的机器指令。

- subproc函数的地址为00401000H，main函数的地址为0040100BH。
- 主程序在调用subproc函数前，将20、30压栈。子程序通过[EBP+8]取得堆栈中的参数a，通过[EBP+0CH]取得堆栈中的参数b。子程序返回主程序后，主程序执行“ADD ESP, 8”，意味着30、20出栈。

- 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

stdcall方式

- stdcall方式的调用规则也是使用堆栈传递参数，使用从右向左的顺序将参数入栈。
- 与cdecl方式不同的是，堆栈的平衡是由子程序来完成的。子程序使用“RET n”指令，在返回主程序的同时平衡ESP。
- 子程序的返回值放在EAX中。
- Windows API采用的调用规则就是stdcall方式

```
int __stdcall subproc(int a, int b)        //两个下划线
{
        return a-b;
}
int r,s;
int main( )
{
        r=subproc(30, 20);
        s=subproc(r, -1);
}
```

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)

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。

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
- 如果不想让编译器生成这些进入代码和退出代码，而是要由编程者自行编写函数内的所有代码。这时，就可以使用naked调用规则。

汇编语言子程序的参数传递方式

- 汇编语言中，向子程序传递参数可以仿照C程序的方式来处理
- 子程序参数传递SubProc1采用cdecl方式，而SubProc2采用stdcall方式

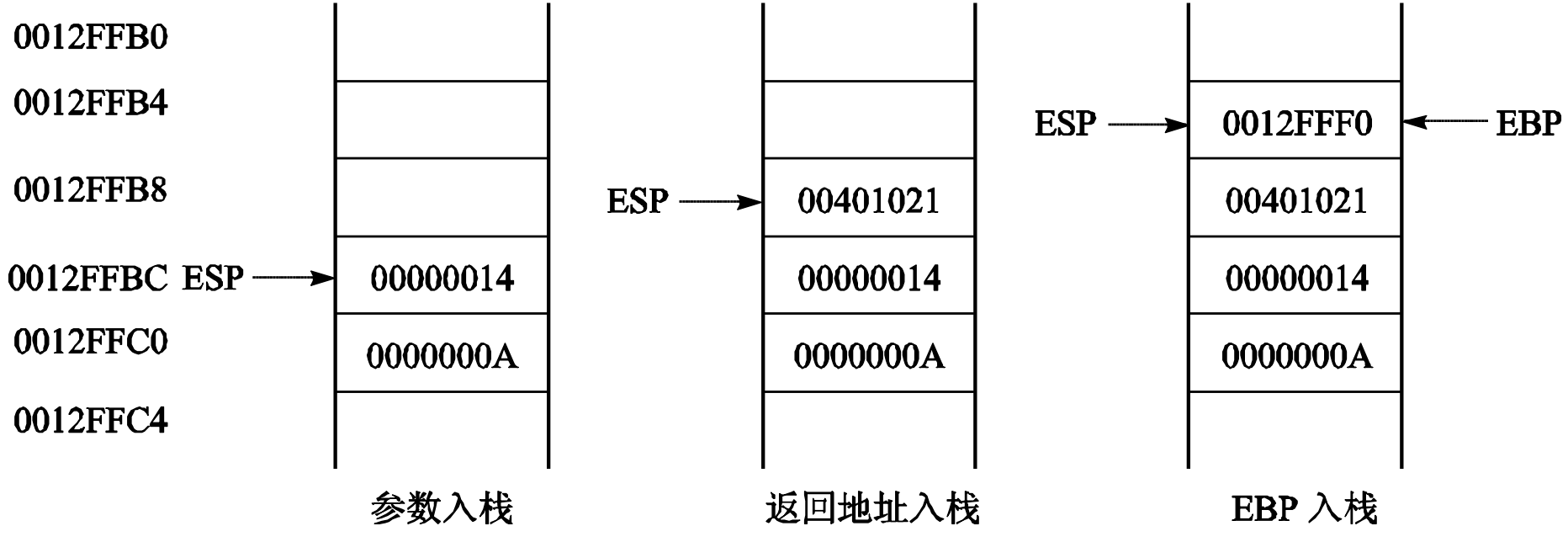
```
.386
.model flat,stdcall
.data
.code
SubProc1        proc                        ;使用堆栈传递参数
                push   ebp
                mov    ebp,esp
                mov    eax,dword ptr [ebp+8]   ;取出第1个参数
                sub     eax,dword ptr [ebp+12]         ;取出第2个参数
                pop    ebp
                ret
SubProc1        endp
```

```
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
```

```
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
```

分析

- 在调用“SubProc1”之前，主程序将10、20压入堆栈；执行“call SubProc1”之后，返回地址被压入堆栈；执行“push ebp”、“mov ebp, esp”之后，EBP被压入堆栈。此时，[EBP+8]的内容为20，即子程序的第1个参数，[EBP+12]的内容为10，是子程序的第2个参数。



带参数子程序的调用

- MASM提供了一个伪指令INVOKE来简化子程序的设计和调用。在定义子程序时，可以说明是使用cdecl规则还是stdcall规则，并指出各参数的名称。在调用子程序时，使用INVOKE伪指令，后面跟子程序名和各个参数即可，由编译软件在编译时完成将参数转换为［EBP+x］等工作。

- 重新前面的程序：定义SubProc1时，后面跟“C”，表示它使用cdecl调用规则（C语言默认的规则）。定义SubProc2时，后面跟“stdcall”，表示它使用stdcall调用规则。调用规则后面直接跟参数的名字和类型。
- 子程序中，不需要使用[EBP+8]、[EBP+12]等形式来指定参数，而直接使用a、b等形式参数即可。MASM自动地将a替换为[EBP+8]，将b替换为[EBP+12]。
- 子程序开始的地方也不再需要“PUSH EBP”、“MOV EBP, ESP”指令，结束时也不需要“POP EBP”指令。编译时，MASM自动在子程序开始的地方插入“PUSH EBP”、“MOV EBP, ESP”指令
- 如果使用了stdcall调用规则，MASM自动将“RET”指令替换为“RET n”指令。n等于参数个数乘以4。

```
.386
.model flat,stdcall
includelib      msvcrt.lib
printf          PROTO C :dword,:vararg
.data
szMsgOut        byte                '%d-%d=%d', 0ah, 0
.code
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
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
```

注意

- 在子程序中不能随意改变EBP的值，因为子程序要依靠EBP来访问位于堆栈中的参数。
- invoke伪指令后面跟的参数不能像C语言那样灵活。在C语言中，参数本身可以是一个表达式，例如SubProc1(r*2, 30)。在汇编语言中，invoke伪指令后面跟的参数必须直接能够作为PUSH指令的源操作数，因此，下面这样的指令是不符合规则的，编译时会报错：
- invoke　　SubProc1, r*2, 30

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。

- MASM提供了LOCAL伪指令，可以在子程序中方便地定义局部变量。
- 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，可以使用多个LOCAL语句。变量类型可以是BYTE、WORD、DWORD等。还可以定义一个局部的结构变量，此时可以用结构的名称当作类型。在子程序中还可以定义一个局部数组

- LOCAL　TEMP[3]:DWORD
- TEMP数组有3个元素。每个元素占4字节，TEMP数组在堆栈中占12字节。在程序中使用TEMP[0]代表第0个元素，TEMP[4]代表第1个元素，TEMP[8]代表第2个元素。
- 如果在子程序中定义了局部变量，而在INVOKE语句中使用这个局部变量的地址，就需要用到ADDR伪操作符，而不能使用OFFSET伪操作符。OFFSET后面只能跟全局变量（即在数据区中定义的变量）和程序中的标号，不能跟局部变量。

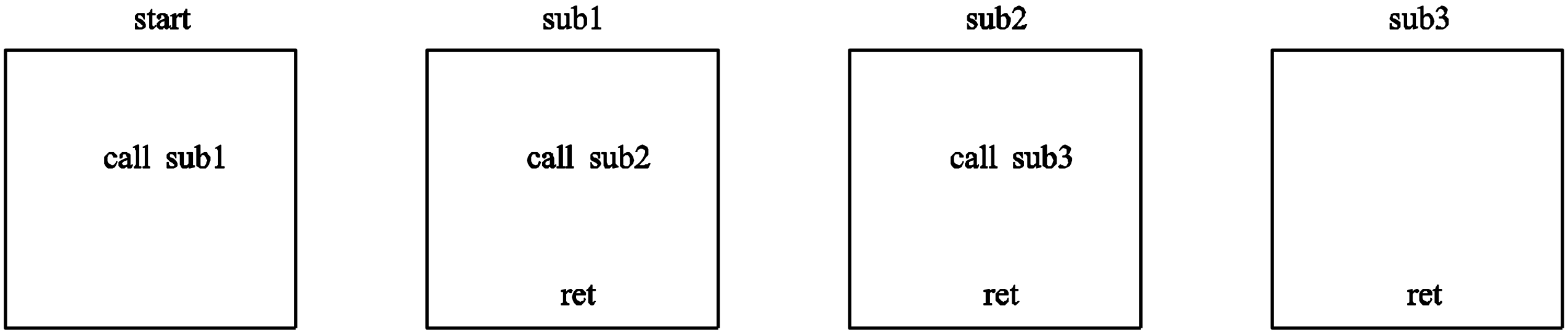
- LOCAL　TEMP1, TEMP2:DWORD
- MASM将TEMP1作为[EBP-4]，将TEMP2作为[EBP-8]。但子程序中直接使用TEMP1、TEMP2，而不必使用[EBP-4]、[EBP-8]的形式。
- SWAP的两个入口参数a和b是两个指针，所以a和b的类型用“PTR DWORD”说明。

```
.386
.model flat, stdcall
includelib  msvcrt.lib
printf        PROTO c :dword,:vararg
.data
r               dword  10
s               dword  20
szMsgOut            byte   'r=%d s=%d', 0ah, 0
```

```
.code
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
```

```
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
```

子程序嵌套（不举例）


start
call sub1
sub1
call sub2
ret
sub2
call sub3
ret
sub3
ret

子程序递归

- 计算n!（n的阶乘）
- n!=n×(n-1)!　　（若n>1）
- n!=1　　（若n=0,1）
- 分析：子程序factorial将n作为参数，结果n!放置在EAX中。子程序中首先判断n是否小于等于1，若是，返回1即可；否则调用它自己求出(n-1)!，调用它自身时，以n-1作为子程序的参数，求出(n-1)!后，再将它乘以n放置在EAX中，作为子程序的返回值。

```
.386
.model flat,stdcall
includelib          msvcrt.lib
printf              PROTO C:dword,:vararg
.data
szOut           byte            'n=%d (n!)=%d', 0AH, 0
```

```
.code
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
start   endp
end     start
```

缓冲区溢出

- 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。
- 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。

模块化程序设计

- 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。
- 通过对系统功能的分析，采取“分而治之”的办法，将一个大的系统分解为小的模块，每一个模块都可以采取不同的编程语言。各个模块的开发可以由多个开发人员并行完成，最后，将所有模块组合成为一个完整的系统。

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。

外部引用伪指令

- 格式：EXTRN/EXTERN 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。
- 说明：这里的名字一般是变量名，变量是在另一模块中定义的。类型可以是BYTE、WORD、DWORD，与另一模块对该变量的定义要一致。EXTRN伪指令应该出现在程序引用该名字之前，一般放在程序的开头。

全局符号说明伪指令

- 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。

子程序声明伪指令

- 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。在模块化程序设计中，若子程序位于另一模块，则在主程序模块中，就需要用PROTO伪指令对子程序的名字、调用方式和参数类型予以说明。

- 设计由两个模块组成的程序，模块名分别为PROG0509.ASM和PROG0510.ASM。其中主模块调用子模块中的SubProc子程序实现减法功能。

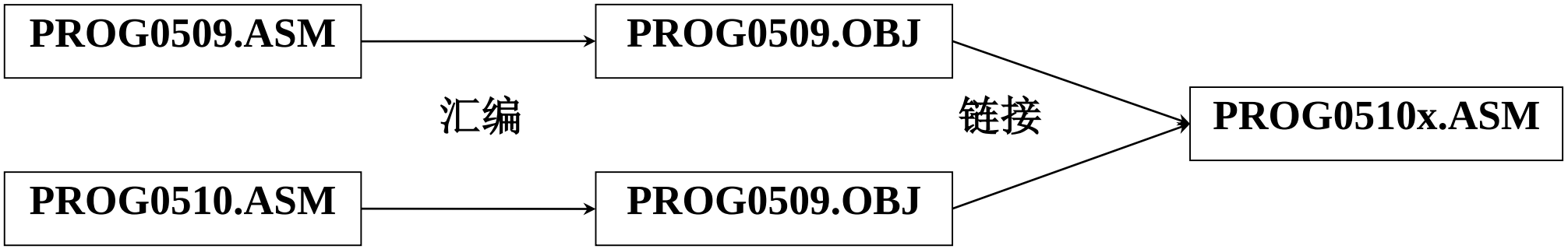
```
;PROG0509.asm
.386
.model      flat,stdcall
option      casemap:none
includelib          msvcrt.lib
printf      PROTO C :dword,:vararg
SubProc     PROTO stdcall :dword, :dword    ; SubProc位于其他模块中
public      result                          ;允许其他模块使用result
.data
szOutputFmtStr    byte    '%d-%d=%d', 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
```

- 使用ML分别编译模块PROG0509.ASM和PROG0510.ASM，分别得到PROG0509.OBJ和PROG0510.OBJ。最后，再使用LINK将两个.OBJ文件连接生成一个.EXE文件，使用/out选项来指定产生的.EXE文件名。



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

C语言模块的反汇编

- C基本框架
- 1: #include "stdio.h"
- 2: int main( )
- 3: {
- 4: return 0;
- 5: }
- 在C基本框架中，需要对部分寄存器初始化，并为局部变量在栈上开辟40h的空间，初始化为0CCh。

反汇编码-栈初始化过程

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 C0	lea	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]
00401038 33 C0	xor	eax,eax

反汇编码-栈初恢复过程

- 0040103A 5F　　pop　　edi
- 0040103B 5E　　pop　　esi
- 0040103C 5B　　pop　　ebx
- 0040103D 8B E5　　mov　　esp,ebp
- 0040103F 5D　　pop　　ebp
- 00401040 C3　　ret

C选择结构

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

- 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
- ;输出printf("i is nonnegative!");

- 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
- ;输出printf("i is negative!");

C循环结构-for

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

- 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)

C-while循环

- 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 83 7D FC 0A cmp dword ptr [ebp-4],0Ah
- ;while(i<=10)
- 00401043 7F 0B jg main+30h (00401050)
- 00401045 8B 45 FC mov eax,dword ptr [ebp-4]
- 00401048 83 C0 01 add eax,1
- 0040104B 89 45 FC mov dword ptr [ebp-4],eax
- 0040104E EB EF jmp main+1Fh (0040103f)

C变量定义

- 局部变量保存在栈中，通过[ebp-n]的方式访问。全局变量和静态全局变量保存在内存区域，通过逻辑地址访问。

```
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

C指针-内存地址

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

- 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。

C函数

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

子程序subproc的反汇编码-栈的初始化

- 00401010 55 push ebp
- 00401011 8B EC mov ebp,esp
- 00401013 83 EC 40 sub esp,40h
- 00401016 53 push ebx
- 00401017 56 push esi
- 00401018 57 push edi
- 00401019 8D 7D C0 lea edi,[ebp-40h]
- 0040101C B9 10 00 00 00 mov ecx,10h
- 00401021 B8 CC CC CC CC mov eax,0CCCCCCCCh
- 00401026 F3 AB rep stos dword ptr [edi]

返回值-保存在eax中

- 00401028 8B 45 08　　mov　　eax,dword ptr [ebp+8]
- 0040102B 0F AF 45 0C　　imul　　eax,dword ptr [ebp+0Ch]
- ;eax = a*b

栈的恢复

- 0040102F 5F　　pop　　edi
- 00401030 5E　　pop　　esi
- 00401031 5B　　pop　　ebx
- 00401032 8B E5　　mov　　esp,ebp
- 00401034 5D　　pop　　ebp
- 00401035 C3　　ret

主程序main反汇编码-第一次函数调用

- 00401005 E9 66 A4 00 00　　jmp　main (0040b470)
- 0040100A E9 01 00 00 00　　jmp　subproc (00401010)
- ；----------（堆栈初始化略）----------
- 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

第二次函数调用

- 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

输出结果

- 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
- ; ----------（堆栈恢复略）----------
- 0040B4CD C3 ret

C语言和汇编语言的混合编程

- 程序的主体部分用高级语言编写，以便缩短开发周期，而程序的关键部分及高级语言不能胜任的部分用汇编语言编写。
- 在C程序中直接嵌入汇编代码
- 由C语言主程序调用汇编子程序。

直接嵌入

- 要在汇编语句前用关键字_asm说明，其格式为：
- _asm　汇编语句
- 内嵌汇编语句的操作码必须是有效的80x86指令。只可嵌入指令，不能是数据。可以使用OFFSET、TYPE、SIZE、LENGTH等汇编语言操作符
- 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
- 　　...
- }

- 操作数可以是寄存器、局部变量、全局变量以及函数参数、结构成员。
- 在内嵌汇编中结构成员可以直接用“结构变量名.成员名”表示。如果结构的地址已放入寄存器中，可以用“[寄存器].成员名”表示。程序中有多个结构使用同一个成员名时，用“[寄存器]结构名.成员名”表示该成员

C程序调用汇编子程序

- C模块可以调用汇编模块中的子程序，还可以使用汇编模块中定义的全局变量。反过来，汇编模块可以调用C模块中的函数，也可以使用C模块中定义的全局变量。
- 如果C程序需要使用汇编模块中的变量，在汇编模块中的变量名必须以下划线开头。同时，在汇编模块中，用PUBLIC语句允许外部模块来访问这些变量。
- 在C模块中，用extern表明这些变量是来自于外部模块，同时说明这些变量的类型

32位模式下的C模块和汇编模块的主要等价变量类型

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

- public _a, _b
- _a sdword 3
- _b sdword 4
- extern int a, b;

汇编模块使用C模块中的变量

- 在汇编模块中要使用C模块中定义的全局变量时，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。
- extern int z;
- int z
- 在编译时，变量的名字前会自动加一个下划线。在汇编模块中，要使用这些变量，需要EXTRN加以说明。
- extrn　　_z:sdword
- mov　　_z, esi

C模块调用汇编模块中的子程序

- 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
- 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);
- 在汇编子程序中，如果用到了EBX、ESI、EDI这3个寄存器，就需要在子程序的开头将这些寄存器保存在堆栈中，在子程序结束时恢复这些寄存器。因为C语言的编译程序总是假设这些寄存器的值在调用子程序的过程中保持不变。

//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;
}
```

;PROG0516.asm

```
.386
.model flat
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                ;子程序中用到EBX, ESI, EDI时
                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
```

- 对C模块和汇编模块分别进行编译，生成各自的.OBJ文件。最后，再将这些.OBJ文件连接成一个可执行文件。

汇编调用C函数

- 汇编模块可以作为主程序，而将C模块中的函数作为子程序，供汇编模块调用。从汇编模块的角度看，这种方式与调用C库函数及Windows API没有什么区别。
- 在汇编模块中，使用PROTO说明C函数的名称、调用方式、参数类型等，就可以调用C函数了。

- input　　PROTO C px:ptr sdword, py:ptr sdword
- output　　PROTO C x:dword, y:dword
- 在C模块中，实现上面两个函数，并用EXTERN说明这些函数可以被外部模块所调用。

;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);
}
```