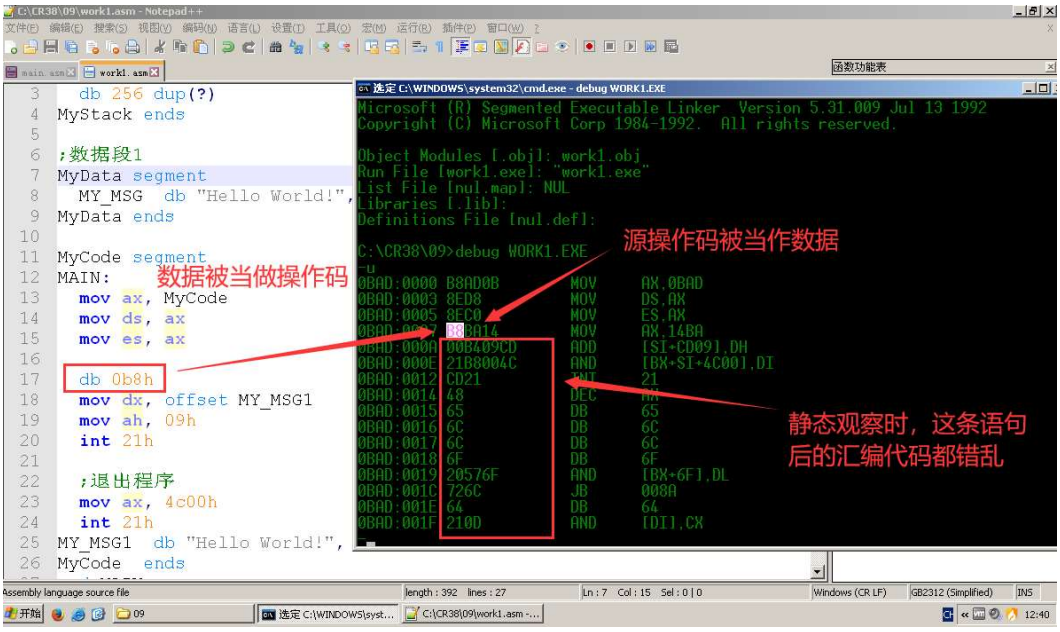
2021/01/06_16位汇编_第8课_花指令、汇编子程序的设计、中断指令

笔记本： 16位汇编
创建时间： 2021/1/6 星期三 10:30
作者： ileemi

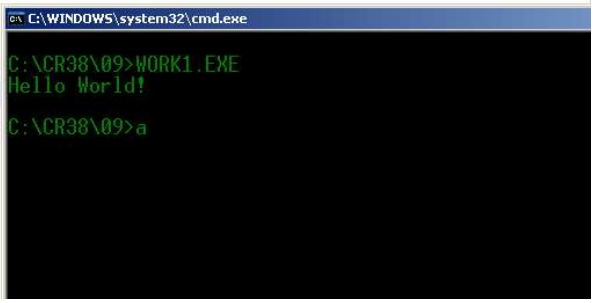
花指令

代码段可以存放数据（需要将数据放置到代码段末尾，同时需要将数据段切换成代码段）。这种方式可用来进行代码加密，在代码段的前面或者中间添加数据代码会被当作操作码（db 0b8h（寄存器）） -- **花指令（成本低）**。



需要保证代码可以正常执行（跳转等），汇编代码就做到了加密，示例如下：

```
11  MyCode segment
12  MAIN:
13    mov ax, MyCode
14    mov ds, ax
15    mov es, ax
16
17    jmp LAEL1
18    db 0b8h
19  LAEL1:
20    mov dx, offset MY_MSG1
21    mov ah, 09h
22    int 21h
23
24    ;退出程序
25    mov ax, 4c00h
26    int 21h
```



> jmp LAEL1: ;反汇编引擎可以检测到
> db 0b8h ;对CPU来说不会被执行
> LAEL1:
> xxx

代码执行结果没有问题，汇编代码错乱。但是上面的方法反汇编引擎可以检测到（反汇编引擎遇到跳转指令将指令到跳转目的地址间的代码当做数据，不将其当做指令），所以也可以使用下面的方法：

改进版本：

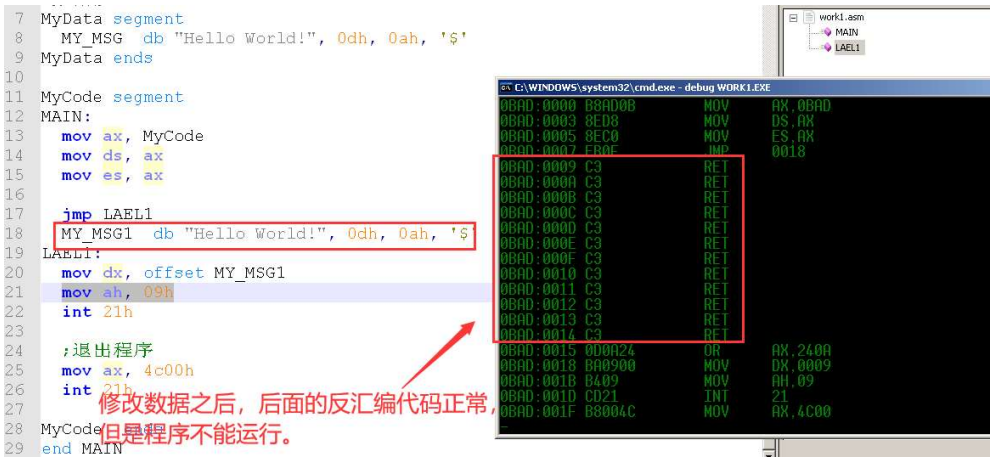
> mov ax, 1
> mov bx, 1
> sub ax, bx
> jz LAEL1 ;**概率跳转**
> db 0b8h ;数据被当作操作码解释
> LAEL1:
> xxx

有些反汇编引擎会模仿一堆乱算指令后的结果，判断其会不会进行跳转。后来发展到读去文件指定字节数据、接收服务器发送数据来保证一定进行跳转（反汇编引擎就不好判断），这样就做到了低成本代码加密。

升级版本，在跳转指令中添加可以输出数据的代码，有使用的代码：

> jmp LAEL1
> MY_MSG1 db "Hello World!", 0dh, 0ah, '$'
> LAEL1:
> mov dx, offset MY_MSG1
> mov ah, 09h

int 21h



可以做个备份，修改后的文件进行反汇编分析。

花指令解法

- 使用花指令的程序在进行调试的时候没有效果（只适合静态观察，执行的时候和正常程序一样）。代码量多的时候，不方便调试。
- 静态观察使用花指令的程序，发现使用花指令的位置，使用 "-u 地址"，之后的代码就显示正常，可进行调试。
- 通用使用静态观察，观察使用花指令处的反汇编代码，观察使用花指令的内存前后数据位置，使用二进制工具就该对应的字节数（将其修改为：C3，一个字节的指令（就不在影响后面的反汇编代码））

汇编代码重用

子程序指令

> 子程序是完成特定功能的一段程序，当主程序（调用程序）需要执行这个功能时，采用CALL调用指令转移到该子程序的起始处执行，当运行完子程序功能后，采用RET返回指令回到主程序继续执行。

方法1：

```
    ;不通用，代码冗余
    jmp  SHOW_HELLO1
RETURN1:
    ...
    jmp  SHOW_HELLO2
RETURN2:
    ...

SHOW_HELLO1:
    mov  dx,  offset  MY_MSG1
```

```
    mov   ah,   09h
    int   21h
    jmp   RETURN1
SHOW_HELLO2:
    mov   dx,   offset   MY_MSG1
    mov   ah,   09h
    int   21h
    jmp   RETURN2
```

方法2:

```

    ;不通用，有一定的局限性SHOW_HELLO  内部使用了所有寄存器
    ;这种方法就不能使用了
    mov   bx,   offset   RETURN1
    jmp   SHOW_HELLO
RETURN1:
    ...
    mov   bx,   offset   RETURN2
    jmp   SHOW_HELLO
RETURN2:
    ...

    ;退出程序
    mov   ax,   4c00h
    int   21h

SHOW_HELLO:
    ;内部使用了所有寄存器
    mov   bx,   0
    mov   cx,   0
    mov   si,   0
    mov   di,   0
    mov   dx,   offset   MY_MSG1
    mov   ah,   09h
    int   21h
    jmp   bx ;bx寄存器数值被修改，跳不回去了
```

方法3：使用堆栈保存跳转时的地址，返回的时候进行恢复 -- **子程序（一段可以重复执行的代码）**

```
    ;子程序
    mov   ax,   offset   RETURN1
    push   ax
    jmp   SHOW_HELLO
RETURN1:
```

```
    mov  ax,  offset  RETURN2
    push  ax
    jmp  SHOW_HELLO
RETURN2:

    ;退出程序
    mov  ax,  4c00h
    int  21h

SHOW_HELLO:
    push  bx
    mov  bx,  0
    mov  cx,  0
    mov  si,  0
    mov  di,  0
    mov  dx,  offset  MY_MSG1
    mov  ah,  09h
    int  21h

    pop  bx    ;自己保存的，需要自己恢复
    pop  ax
    jmp  ax    ;跳转到进行跳转之前保存的地址

MY_MSG1    db  "Hello  World!",  0dh,  0ah,  '$'
```

> 使用**子程序指令：** call 、ret （两个指令不影响寄存器）优化方法3：

```
  ;子程序
    ;  mov  ax,  offset  RETURN1
    ;  push  ax
    ;  jmp  SHOW_HELLO
;  RETURN1:
    call  SHOW_HELLO ;等价上面三行代码

    ;  mov  ax,  offset  RETURN2
    ;  push  ax
    ;  jmp  SHOW_HELLO
;  RETURN2:
    call  SHOW_HELLO  ;等价上面三行代码

    ;退出程序
    mov  ax,  4c00h
    int  21h

SHOW_HELLO:
    push  bx ;在入口处保存环境
```

```
    mov  bx,  0
    mov  cx,  0
    mov  dx,  offset  MY_MSG1
    mov  ah,  09h
    int  21h

    pop  bx    ;自己保存的，需要自己恢复（在ret指令执行前，恢复环境）

    ;pop  ax
    ;jmp  ax  ;跳转到进行跳转之前保存的地址 --> pop ip
    ret          ;等价上面两行汇编指令，不影响寄存器

MY_MSG1    db  "Hello  World!",  0dh,  0ah,  '$'
```

> 使用子程序指令 `call` 不需要在写标号，CPU会将下一条指令的地址存储到堆栈中（CPU执行这条指令的时候，根据指令长度可以计算处下一条指令的地址）。**自己压入堆栈内的数据需要在执行 `ret` 指令之前弹出，否则栈顶的地址会当作返回地址使用。**

> **注意**：调用子程序之前，有进行寄存器操作，在子程序中有对寄存器复制等，退出子程序后，之前使用的寄存器数值发生变化。
> **解决办法**：保存寄存器环境（在子程序中入口处将子程序中使用的寄存器入栈），推出前恢复寄存器环境（ret指令执行前，将入栈的寄存器弹出），**子程序中使用哪些寄存器就将哪些寄存器入栈即可。**
> 代码示例：

```
    mov  cx,  1
    call  SHOW_HELLO
    mov  cx,  2
    call  SHOW_HELLO
    mov  cx,  3
    call  SHOW_HELLO

    ;退出程序
    mov  ax,  4c00h
    int  21h
SHOW_HELLO:
    push  bx    ;保存寄存器环境
    push  cx
    push  si
    push  di
    mov  bx,  0
    mov  cx,  0
    mov  si,  0
    mov  di,  0
    mov  dx,  offset  MY_MSG1
```

```
    mov  ah,  09h
    int  21h

    pop  bx    ;恢复寄存器环境
    pop  cx
    pop  si
    pop  di
    ;pop  ax
    ;jmp  ax    ;跳转到进行跳转之前保存的地址  -->  pop  ip
    ret            ;等价上面两行汇编指令，不影响寄存器
MY_MSG1    db  "Hello  World!",  0dh,  0ah,  '$'
```

函数

> 子程序中保存环境是函数的概念。

提供子程序的通用性，去除子程序中使用固定的字符串 -- **给子程序添加参数。**

给子程序添加参数

> 通过寄存器传递参数（**调用约定**）。

```
;堆栈段
MyStack  segment    stack
    db  256  dup(?)
MyStack  ends

;数据段
MyData  segment
    MY_MSG1    db  "Hello  World1!",  0dh,  0ah,  '$'
    MY_MSG2    db  "Hello  World2!",  0dh,  0ah,  '$'
MyData  ends

;代码段
MyCode  segment
MAIN:
    mov  ax,  MyData
    mov  ds,  ax
    mov  es,  ax
    ;给子程序添加参数，通过寄存器传递参数
    mov  ax,  offset  MY_MSG1
    call  SHOW_HELLO
    mov  ax,  offset  MY_MSG2
    call  SHOW_HELLO
    mov  ax,  offset  MY_MSG1
```

```
    call  SHOW_HELLO

    ;退出程序
    mov  ax,  4c00h
    int  21h

SHOW_HELLO:
    push  ax     ;保存寄存器环境
    push  dx
    push  cx
    mov  cx,  0
    mov  dx,  ax     ;ax  保存要输出的字符串地址
    mov  ah,  09h
    int  21h

    pop  dx     ;恢复寄存器环境
    pop  cx
    pop  ax
    ;pop  ax
    ;jmp  ax     ;跳转到进行跳转之前保存的地址  -->  pop  ip
    ret             ;等价上面两行汇编指令，不影响寄存器

MyCode  ends
end  MAIN
```

通过寄存器传递参数（**调用约定**），参数少的时候可取，当传递参数较多的时候，寄存器不够用。可以使用内存（堆栈）传递参数。使用堆栈传递参数的好处：可传递参数较多，子程序使用玩，堆栈可以进行释放，达到重复使用。**注意：为了参数访问时方便，参数的顺序（参数存放到栈顶访问时比较方便）需要是 "从右往左依次入栈"。**

子程序使用参数时，需要进行计算。在call的时候，栈底保存的是返回值（入栈顺序：参数1 --> 参数2 --> 执行 `call` 指令，保存的是下一条指令的地址（返回地址）），所以在获取参数1：`[sp+2]`，获取参数2：`[sp+4]`。注意：sp不是基址寄存器，不能用来直接访问内存，可使用：mov bp, sp。

> **注意**：跳转到子程序之前**入栈的参数**，在退出子程序后并不会出栈，会导致堆栈不平衡（子程序使用过多的时候会导致栈爆）。
> **解决办法**：
> 方法1：在退出子程序后，手动平衡对应字节数的栈空间（调用者平衡栈：C调用约定）。示例：add sp, 4
> 方法2：在子程序内部返回之前，函数自己平衡对应字节数的栈空间（函数本身进行释放栈空间：stdcall）示例：
> pop ax ;比较低效
> add sp, 4 ;栈顶需要是返回地址
> push ax
> ret   ;等价上面两行汇编指令，不影响寄存器

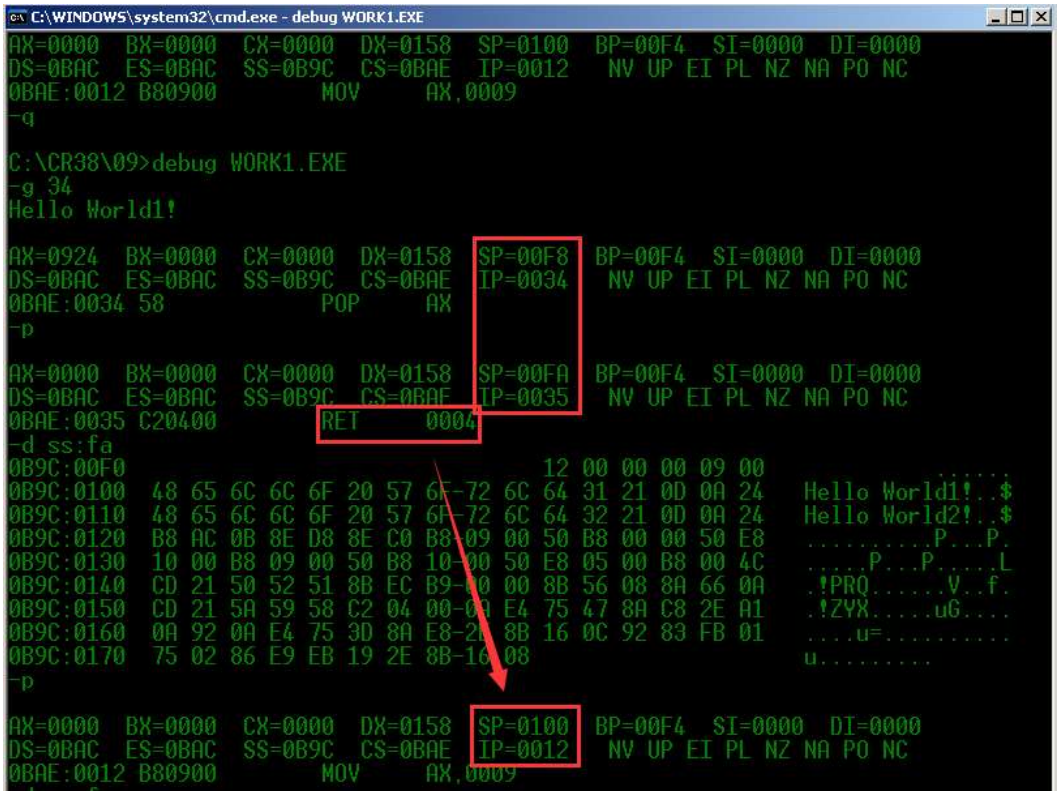
CPU 指令 `retn` 等价上面四行代码，效率比上面四行要高，使用示例：

> retn 4
> 执行mov ip, [sp]（将栈顶数据，给IP）之前，先执行add sp, 4。CPU可以做到，CPU可以将返回地址存放到暂存寄存器中。

代码示例：

```
;堆栈段
MyStack  segment    stack
    db  256  dup(?)
MyStack  ends

;数据段
MyData  segment
    MY_MSG1    db  "Hello  World1!",  0dh,  0ah,  '$'
    MY_MSG2    db  "Hello  World2!",  0dh,  0ah,  '$'
MyData  ends

;代码段
MyCode  segment
MAIN:
    mov  ax,  MyData
    mov  ds,  ax
    mov  es,  ax

    ;从右往左依次入栈
    mov  ax,  09h
    push  ax
    mov  ax,  offset  MY_MSG1    ;调用约定  寄存器传参，注意参数顺序
    push  ax
    call  SHOW_HELLO
    ;add  sp,  4  ;调用者手动平衡对应字节数的栈空间

    mov  ax,  09h
    push  ax
    mov  ax,  offset  MY_MSG2    ;调用约定  寄存器传参，注意参数顺序
    push  ax
    call  SHOW_HELLO
    ;add  sp,  4  ;调用者手动平衡对应字节数的栈空间

    ;退出程序
    mov  ax,  4c00h
    int  21h

SHOW_HELLO:
    push  ax    ;保存寄存器环境
    push  dx
    push  cx
```

```
    ;获取参数1：[sp+2]
    ;获取参数2：[sp+4]
    mov  bp,  sp              ;sp不是基址寄存器，不能用来直接访问内存
    mov  cx,  0
    mov  dx,  word  ptr[bp+8]     ;[sp+8]:子程序入口有三个push
    mov  ah,  byte  ptr[bp+10]
    int  21h

    pop  dx     ;恢复寄存器环境
    pop  cx
    pop  ax
    ;pop  ax
    ;jmp  ax     ;跳转到进行跳转之前保存的地址  -->  pop  ip、

    ;pop  ax
    ;add  sp,  4     ;栈顶需要是返回地址
    ;push  ax
    ;ret             ;等价pop  ax、jmp  ax，不影响寄存器
    retn  4         ;函数内部自动平衡栈
MyCode  ends
end  MAIN
```



局部变量

上面的代码还有需要改进的地方，就是当子程序中，右新的寄存器需要入栈，那么调用参数的部分就需要改动。代码如下：

```
SHOW_HELLO:
    push  ax     ;保存寄存器环境
    push  dx
    push  cx

    ;获取参数1：[sp+2]
    ;获取参数2：[sp+4]
    mov  bp,  sp              ;sp不是基址寄存器，不能用来直接访问内存
    mov  cx,  0
    mov  dx,  word  ptr[bp+8]     ;[sp+8]:子程序入口有三个push
    mov  ah,  byte  ptr[bp+10]  ; 上下两句都需要改动
    int  21h

    pop  cx     ;恢复寄存器环境
    pop  dx
    pop  ax
    retn  4          ;函数内部自动平衡栈
```

对上面的代码进行修改：入口处**保存栈底（可使访问参数时偏移固定）**

```
SHOW_HELLO:
    mov  bp,  sp              ;sp不是基址寄存器，不能用来直接访问内存
  ;防止在调用子程序之前 bp寄存器被使用，应将其入栈
  push bp
    push  ax     ;保存寄存器环境
    push  dx
    push  cx

    ;获取参数1：[sp+2]
    ;获取参数2：[sp+4]
    mov  cx,  0
    mov  dx,  word  ptr[bp+2]     ;[sp+8]:子程序入口有三个push
    mov  ah,  byte  ptr[bp+4]  ;上下两句都需要改动
    int  21h

    pop  cx     ;恢复寄存器环境
    pop  dx
    pop  ax
  pop bp

    retn  4          ;函数内部自动平衡栈
```

> 访问栈时，通常使用 bp（其保存的的时当前段的栈低，保存sp的值，不是最初原始的栈底），为了使栈底固定，方便访问。
> 子程序入口，出口尽量使用固定的格式，方便访问参数，退出时保证堆栈平衡。

> 同时还需要保存寄存器环境（防止子程序内部使用了寄存器影响外面对应的寄存器）。由于能用的通用寄存器不多，在没有空闲寄存器使用的，可将值保存到栈中。

局部变量的访问地址固定问题，需要依次满足下面步骤：

1. 保存栈底
2. 保存局部变量（手动申请局部变量空间）
3. 保存寄存器环境
4. 恢复寄存器环境
5. 释放局部变量空间（不需要计算局部变量空间大小：mov sp, bp）
6. 恢复栈底

一个函数完整的写法，代码示例：

```
;---------------------------------
;       两数相加
;---------------------------------
MY_ADD:
    ;1.  保存栈底
    push  bp
    mov  bp,  sp

    ;在没有空闲寄存器使用的，可将值保存到栈中
    ;避免大量使用push  pop，可手动抬栈
    ;在保存寄存器环境之前抬栈，可使局部变量访问地址固定
    ;2.  固定局部变量位置
    sub  sp,  4          ;手动抬栈，类似申请局部变量

    ;3.  保存寄存器环境
    push  bx              ;ax作为保存返回值的寄存器，不在需要保存环境

    mov  ax,  [bp+4]
    mov  [bp-2],  ax     ;保存临时数据(第一个局部变量)
    add  ax,  [bp+6]
    mov  [bp-4],  ax     ;保存临时数据(第二个局部变量)
    ;mov  [sp],  ax

    ;4.  释放寄存器环境
    pop  bx

    ;5.  释放局部变量
    ;add  sp,  4          ;释放栈(类似释放局部变量)
    mov  sp,  bp     ;释放局部变量空间(不需要计算局部变量空间大小)

    ;6.  恢复栈底
    pop  bp
    retn  4
```

返回值（保存运算结果）

方法1（比较麻烦）：

- 可以在子程序外申请一个局部变量
- 将局部变量作为参数传到子程序中
- 将运算结果保存到该局部变量中并返回

方法2（返回值使用一个寄存器保存）：

- 需要提前约定好，一般使用寄存器 ax
- 使用 ax 作为保存返回值的寄存器，寄存器ax 在子程序中就不需要保存环境

段间子程序调用

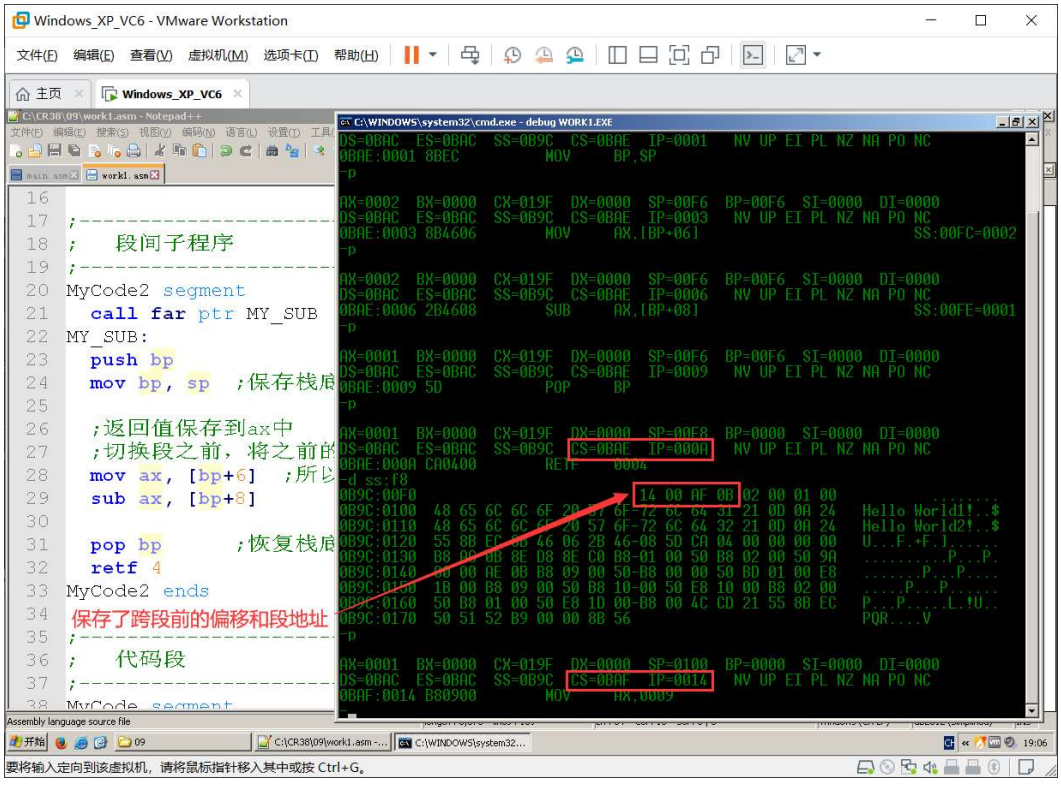
自动切换段（cs）：call far ptr MY_SUB
自动恢复段（cs）：retf

> 切换段之前，将之前的CS和IP保存到堆栈中（便于恢复段）。所以访问参数时，需要[bp+6]；跨段访问的函数，在其段中，有想访问其子程序时在使用 `call` 就不通用，通用需要使用 `call far ptr` 。需要提前确定是：**段间子程序** 还是 **段内子程序。**

代码示例：

```
;---------------------------------
;       堆栈段
;---------------------------------
MyStack  segment    stack
    db  256  dup(?)
MyStack  ends

;---------------------------------
;       数据段
;---------------------------------
MyData  segment
    MY_MSG1    db  "Hello  World1!",  0dh,  0ah,  '$'
    MY_MSG2    db  "Hello  World2!",  0dh,  0ah,  '$'
MyData  ends

;---------------------------------
;       段间子程序
;---------------------------------
MyCode2  segment
    call  far  ptr  MY_SUB  ;不使用far  ptr，段内访问子程序，获取的参数地址不对
MY_SUB:
```

```
    push  bp
    mov   bp,   sp      ;保存栈底

    ;返回值保存到ax中
    ;切换段之前，将之前的CS和IP保存到堆栈中（便于恢复段）
    mov   ax,   [bp+6]      ;所以访问参数1时需要  +6
    sub   ax,   [bp+8]

    pop   bp              ;恢复栈底
    retf  4
MyCode2  ends

;--------------------------------
;       代码段
;--------------------------------
MyCode  segment
MAIN:
    mov   ax,   MyData
    mov   ds,   ax
    mov   es,   ax
    ;段间(跨段)子程序访问
    mov   ax,   1
    push  ax
    mov   ax,   2
    push  ax
    call  far  ptr  MY_SUB
MyCode  ends
end  MAIN
```



中断指令

> 中断（Interrupt）是又一种改变程序执行顺序的方法。

中断请求可以来自处理器外部的中断源，也可以由处理器执行指令引起。

INT

> int 21H：Dos功能调用（打断正在执行的代码），21H号中断是 DOS 提供给用户的用于调用系统功能的中断，它有近百个功能供用户选择使用，主要包括设备管理、目录管理和文件管理三个方面的功能。

中断表（由CPU规定）在内存中，地址是固定的，int 21h（操作系统从中断表中取出地址并调用）

中断函数的返回指令：iret