[**LINUX平台可以用GDB进行反汇编和调试。**](http://www.cppblog.com/liu1061/articles/53762.html)

Posted on 2008-06-17 21:16 [T.S Liu](http://www.cppblog.com/liu1061/) 阅读(13832) [评论(2)](http://www.cppblog.com/liu1061/articles/53762.html#Post)  [编辑](http://www.cppblog.com/liu1061/admin/EditArticles.aspx?postid=53762) [收藏](http://www.cppblog.com/liu1061/AddToFavorite.aspx?id=53762) [引用](http://www.cppblog.com/liu1061/services/trackbacks/53762.aspx) 所属分类: [ubuntu](http://www.cppblog.com/liu1061/category/7273.html) http://www.cppblog.com/liu1061/aggbug/53762.html?webview=1

如果在Linux平台可以用gdb进行反汇编和调试。(转)  
  
2. 最简C代码分析  
  
    为简化问题，来分析一下最简的c代码生成的汇编代码：  
    # vi test1.c  
        
    int main()  
    {  
        return 0;  
    }     
      
    编译该程序，产生二进制文件：  
    # gcc test1.c -o test1  
    # file test1    
    test1: ELF 32-bit LSB executable 80386 Version 1, dynamically linked, not stripped   
  
    test1是一个ELF格式32位小端(Little Endian)的可执行文件，动态链接并且符号表没有去除。  
    这正是Unix/Linux平台典型的可执行文件格式。  
    用mdb反汇编可以观察生成的汇编代码：  
  
    # mdb test1  
    Loading modules: [ libc.so.1 ]  
    > main::dis                       ; 反汇编main函数，mdb的命令一般格式为  <地址>::dis  
    main:          pushl   %ebp       ; ebp寄存器内容压栈，即保存main函数的上级调用函数的栈基地址  
    main+1:        movl    %esp,%ebp  ; esp值赋给ebp，设置main函数的栈基址  
    main+3:          subl    $8,%esp  
    main+6:          andl    $0xf0,%esp  
    main+9:          movl    $0,%eax  
    main+0xe:        subl    %eax,%esp  
    main+0x10:     movl    $0,%eax    ; 设置函数返回值0  
    main+0x15:     leave              ; 将ebp值赋给esp，pop先前栈内的上级函数栈的基地址给ebp，恢复原栈基址  
    main+0x16:     ret                ; main函数返回，回到上级调用  
    >   
  
    注：这里得到的汇编语言语法格式与Intel的手册有很大不同，Unix/Linux采用AT&T汇编格式作为汇编语言的语法格式  
         如果想了解AT&T汇编可以参考文章：[Linux AT&T 汇编语言开发指南](http://blog.csdn.net/yayong/archive/2004/10/17/139983.aspx)   
  
    问题：谁调用了 main函数？  
       
     在C语言的层面来看，main函数是一个程序的起始入口点，而实际上，ELF可执行文件的入口点并不是main而是\_start。  
     mdb也可以反汇编\_start：  
         
    > \_start::dis                       ;从\_start 的地址开始反汇编  
    \_start:              pushl   $0  
    \_start+2:            pushl   $0  
    \_start+4:            movl    %esp,%ebp  
    \_start+6:            pushl   %edx  
    \_start+7:            movl    $0x80504b0,%eax  
    \_start+0xc:          testl   %eax,%eax  
    \_start+0xe:          je      +0xf            <\_start+0x1d>  
    \_start+0x10:         pushl   $0x80504b0  
    \_start+0x15:         call    -0x75           <atexit>  
    \_start+0x1a:         addl    $4,%esp  
    \_start+0x1d:         movl    $0x8060710,%eax  
    \_start+0x22:         testl   %eax,%eax  
    \_start+0x24:         je      +7              <\_start+0x2b>  
    \_start+0x26:         call    -0x86           <atexit>  
    \_start+0x2b:         pushl   $0x80506cd  
    \_start+0x30:         call    -0x90           <atexit>  
    \_start+0x35:         movl    +8(%ebp),%eax  
    \_start+0x38:         leal    +0x10(%ebp,%eax,4),%edx  
    \_start+0x3c:         movl    %edx,0x8060804  
    \_start+0x42:         andl    $0xf0,%esp  
    \_start+0x45:         subl    $4,%esp  
    \_start+0x48:         pushl   %edx  
    \_start+0x49:         leal    +0xc(%ebp),%edx  
    \_start+0x4c:         pushl   %edx  
    \_start+0x4d:         pushl   %eax  
    \_start+0x4e:         call    +0x152          <\_init>  
    \_start+0x53:         call    -0xa3           <\_\_fpstart>  
    \_start+0x58:        call    +0xfb        <main>              ;在这里调用了main函数  
    \_start+0x5d:         addl    $0xc,%esp  
    \_start+0x60:         pushl   %eax  
    \_start+0x61:         call    -0xa1           <exit>  
    \_start+0x66:         pushl   $0  
    \_start+0x68:         movl    $1,%eax  
    \_start+0x6d:         lcall   $7,$0  
    \_start+0x74:         hlt  
    >   
  
    问题：为什么用EAX寄存器保存函数返回值？  
    实际上IA32并没有规定用哪个寄存器来保存返回值。但如果反汇编Solaris/Linux的二进制文件，就会发现，都用EAX保存函数返回值。  
    这不是偶然现象，是操作系统的ABI(Application Binary Interface)来决定的。  
    Solaris/Linux操作系统的ABI就是Sytem V ABI。  
  
  
    概念：SFP (Stack Frame Pointer) 栈框架指针   
  
    正确理解SFP必须了解：  
        IA32 的栈的概念  
        CPU 中32位寄存器ESP/EBP的作用  
        PUSH/POP 指令是如何影响栈的  
        CALL/RET/LEAVE 等指令是如何影响栈的  
  
    如我们所知：  
    1)IA32的栈是用来存放临时数据，而且是LIFO，即后进先出的。栈的增长方向是从高地址向低地址增长，按字节为单位编址。  
    2) EBP是栈基址的指针，永远指向栈底（高地址），ESP是栈指针，永远指向栈顶（低地址）。  
    3) PUSH一个long型数据时，以字节为单位将数据压入栈，从高到低按字节依次将数据存入ESP-1、ESP-2、ESP-3、ESP-4的地址单元。  
    4) POP一个long型数据，过程与PUSH相反，依次将ESP-4、ESP-3、ESP-2、ESP-1从栈内弹出，放入一个32位寄存器。  
    5) CALL指令用来调用一个函数或过程，此时，下一条指令地址会被压入堆栈，以备返回时能恢复执行下条指令。  
    6) RET指令用来从一个函数或过程返回，之前CALL保存的下条指令地址会从栈内弹出到EIP寄存器中，程序转到CALL之前下条指令处执行  
    7) ENTER是建立当前函数的栈框架，即相当于以下两条指令：  
        pushl   %ebp  
        movl    %esp,%ebp  
    8) LEAVE是释放当前函数或者过程的栈框架，即相当于以下两条指令：  
        movl ebp esp  
        popl  ebp  
  
    如果反汇编一个函数，很多时候会在函数进入和返回处，发现有类似如下形式的汇编语句：   
          
        pushl   %ebp            ; ebp寄存器内容压栈，即保存main函数的上级调用函数的栈基地址  
        movl    %esp,%ebp       ; esp值赋给ebp，设置 main函数的栈基址  
        ...........             ; 以上两条指令相当于 enter 0,0  
        ...........  
        leave                   ; 将ebp值赋给esp，pop先前栈内的上级函数栈的基地址给ebp，恢复原栈基址  
        ret                     ; main函数返回，回到上级调用  
  
    这些语句就是用来创建和释放一个函数或者过程的栈框架的。  
    原来编译器会自动在函数入口和出口处插入创建和释放栈框架的语句。  
    函数被调用时：  
    1) EIP/EBP成为新函数栈的边界  
    函数被调用时，返回时的EIP首先被压入堆栈；创建栈框架时，上级函数栈的EBP被压入堆栈，与EIP一道行成新函数栈框架的边界  
    2) EBP成为栈框架指针SFP，用来指示新函数栈的边界  
    栈框架建立后，EBP指向的栈的内容就是上一级函数栈的EBP，可以想象，通过EBP就可以把层层调用函数的栈都回朔遍历一遍，调试器就是利用这个特性实现 backtrace功能的  
    3) ESP总是作为栈指针指向栈顶，用来分配栈空间  
    栈分配空间给函数局部变量时的语句通常就是给ESP减去一个常数值，例如，分配一个整型数据就是 ESP-4  
    4) 函数的参数传递和局部变量访问可以通过SFP即EBP来实现   
    由于栈框架指针永远指向当前函数的栈基地址，参数和局部变量访问通常为如下形式：  
        +8+xx(%ebp)         ; 函数入口参数的的访问  
        -xx(%ebp)           ; 函数局部变量访问  
              
    假如函数A调用函数B，函数B调用函数C ，则函数栈框架及调用关系如下图所示：

+-------------------------+----> 高地址  
 | EIP (上级函数返回地址) |   
 +-------------------------+   
 +--> | EBP (上级函数的EBP) | --+ <------当前函数A的EBP (即SFP框架指针)   
 | +-------------------------+ +-->偏移量A   
 | | Local Variables | |  
 | | .......... | --+ <------ESP指向函数A新分配的局部变量,局部变量可以通过A的ebp-偏移量A访问   
 | f +-------------------------+  
 | r | Arg n(函数B的第n个参数) |   
 | a +-------------------------+  
 | m | Arg .(函数B的第.个参数) |  
 | e +-------------------------+  
 | | Arg 1(函数B的第1个参数) |  
 | o +-------------------------+  
 | f | Arg 0(函数B的第0个参数) | --+ <------ B函数的参数可以由B的ebp+偏移量B访问  
 | +-------------------------+ +--> 偏移量B  
 | A | EIP (A函数的返回地址) | |   
 | +-------------------------+ --+   
 +--- | EBP (A函数的EBP) |<--+ <------ 当前函数B的EBP (即SFP框架指针)   
 +-------------------------+ |  
 | Local Variables | |  
 | .......... | | <------ ESP指向函数B新分配的局部变量  
 +-------------------------+ |  
 | Arg n(函数C的第n个参数) | |  
 +-------------------------+ |  
 | Arg .(函数C的第.个参数) | |  
 +-------------------------+ +--> frame of B  
 | Arg 1(函数C的第1个参数) | |  
 +-------------------------+ |  
 | Arg 0(函数C的第0个参数) | |  
 +-------------------------+ |  
 | EIP (B函数的返回地址) | |  
 +-------------------------+ |  
 +--> | EBP (B函数的EBP) | --+ <------ 当前函数C的EBP (即SFP框架指针)   
 | +-------------------------+  
 | | Local Variables |  
 | | .......... | <------ ESP指向函数C新分配的局部变量  
 | +-------------------------+----> 低地址  
frame of C  
   
 图 1-1

    再分析test1反汇编结果中剩余部分语句的含义：  
          
    # mdb test1  
    Loading modules: [ libc.so.1 ]  
    > main::dis                        ; 反汇编main函数  
    main:          pushl   %ebp                              
    main+1:        movl    %esp,%ebp        ; 创建Stack Frame(栈框架)  
    main+3:       subl    $8,%esp       ; 通过ESP-8来分配8字节堆栈空间  
    main+6:       andl    $0xf0,%esp    ; 使栈地址16字节对齐  
    main+9:       movl    $0,%eax       ; 无意义  
    main+0xe:     subl    %eax,%esp     ; 无意义  
    main+0x10:     movl    $0,%eax          ; 设置main函数返回值  
    main+0x15:     leave                    ; 撤销Stack Frame(栈框架)  
    main+0x16:     ret                      ; main 函数返回  
    >  
  
    以下两句似乎是没有意义的，果真是这样吗？  
        movl    $0,%eax   
        subl     %eax,%esp  
         
    用gcc的O2级优化来重新编译test1.c:  
    # gcc -O2 test1.c -o test1  
    # mdb test1  
    > main::dis  
    main:         pushl   %ebp  
    main+1:       movl    %esp,%ebp  
    main+3:       subl    $8,%esp  
    main+6:       andl    $0xf0,%esp  
    main+9:       xorl    %eax,%eax      ; 设置main返回值，使用xorl异或指令来使eax为0  
    main+0xb:     leave  
    main+0xc:     ret  
    >   
    新的反汇编结果比最初的结果要简洁一些，果然之前被认为无用的语句被优化掉了，进一步验证了之前的猜测。  
    提示：编译器产生的某些语句可能在程序实际语义上没有用处，可以用优化选项去掉这些语句。  
  
    问题：为什么用xorl来设置eax的值？  
    注意到优化后的代码中，eax返回值的设置由 movl $0,%eax 变为 xorl %eax,%eax ，这是因为IA32指令中，xorl比movl有更高的运行速度。  
  
    概念：Stack aligned 栈对齐  
    那么，以下语句到底是和作用呢？  
        subl    $8,%esp  
       andl    $0xf0,%esp     ; 通过andl使低4位为0，保证栈地址16字节对齐  
         
    表面来看，这条语句最直接的后果是使ESP的地址后4位为0，即16字节对齐，那么为什么这么做呢？  
    原来，IA32 系列CPU的一些指令分别在4、8、16字节对齐时会有更快的运行速度，因此gcc编译器为提高生成代码在IA32上的运行速度，默认对产生的代码进行16字节对齐  
  
        andl $0xf0,%esp 的意义很明显，那么 subl $8,%esp 呢，是必须的吗？  
    这里假设在进入main函数之前，栈是16字节对齐的话，那么，进入main函数后，EIP和EBP被压入堆栈后，栈地址最末4位二进制位必定是1000，esp -8则恰好使后4位地址二进制位为0000。看来，这也是为保证栈16字节对齐的。  
  
    如果查一下gcc的手册，就会发现关于栈对齐的参数设置：  
    -mpreferred-stack-boundary=n    ; 希望栈按照2的n次的字节边界对齐, n的取值范围是2-12  
  
    默认情况下，n是等于4的，也就是说，默认情况下，gcc是16字节对齐，以适应IA32大多数指令的要求。  
  
    让我们利用-mpreferred-stack-boundary=2来去除栈对齐指令：  
        
    # gcc -mpreferred-stack-boundary=2 test1.c -o test1  
         
    > main::dis  
    main:       pushl   %ebp  
    main+1:     movl    %esp,%ebp  
    main+3:     movl    $0,%eax  
    main+8:     leave  
    main+9:     ret  
    >   
  
    可以看到，栈对齐指令没有了，因为，IA32的栈本身就是4字节对齐的，不需要用额外指令进行对齐。  
    那么，栈框架指针SFP是不是必须的呢？  
    # gcc -mpreferred-stack-boundary=2 -fomit-frame-pointer test1.c -o test  
    > main::dis  
    main:       movl    $0,%eax  
    main+5:     ret  
    >   
  
    由此可知，-fomit-frame-pointer 可以去除SFP。  
         
    问题：去除SFP后有什么缺点呢？  
         
    1)增加调式难度  
        由于SFP在调试器backtrace的指令中被使用到，因此没有SFP该调试指令就无法使用。  
    2)降低汇编代码可读性  
        函数参数和局部变量的访问，在没有ebp的情况下，都只能通过+xx(esp)的方式访问，而很难区分两种方式，降低了程序的可读性。  
         
    问题：去除SFP有什么优点呢？  
         
    1)节省栈空间  
    2)减少建立和撤销栈框架的指令后，简化了代码  
    3)使ebp空闲出来，使之作为通用寄存器使用，增加通用寄存器的数量  
    4)以上3点使得程序运行速度更快  
  
    概念：Calling Convention  调用约定和 ABI (Application Binary Interface) 应用程序二进制接口  
           
        函数如何找到它的参数？  
        函数如何返回结果？  
        函数在哪里存放局部变量？  
        那一个硬件寄存器是起始空间？  
        那一个硬件寄存器必须预先保留？  
  
    Calling Convention  调用约定对以上问题作出了规定。Calling Convention也是ABI的一部分。  
    因此，遵守相同ABI规范的操作系统，使其相互间实现二进制代码的互操作成为了可能。  
    例如：由于Solaris、Linux都遵守System V的ABI，Solaris 10就提供了直接运行Linux二进制程序的功能。  
    详见文章：[关注： Solaris 10的10大新变化](http://blog.csdn.net/yayong/archive/2004/09/14/103625.aspx)  
               
3. 小结  
    本文通过最简的C程序，引入以下概念：  
        SFP 栈框架指针  
        Stack aligned 栈对齐  
        Calling Convention  调用约定 和 ABI (Application Binary Interface) 应用程序二进制接口  
    今后，将通过进一步的实验，来深入了解这些概念。通过掌握这些概念，使在汇编级调试程序产生的core dump、掌握C语言高级调试技巧成为了可能。