x86 Control Flow, Stacks and Procedure Calls

Assigned Monday, July 21, 2014
Suggested Completion Date *Monday, July 28, 2014*

Introduction

The purpose of written homework assignments is to get you thinking about the topics being covered in lecture and in readings in the textbook which are not represented in the hands-on, programming lab assignments. It is worth noting that the book contains many practice problems similar to the problems we ask here. The solutions for those practice problems are located at the end of each chapter and should give you a feel for the kind of answers we expect you to turn in for these kind of assignments.

Logistics

Written homeworks will not be turned in for credit like the programming lab assignments. Solutions will NOT be provided, but the forums are open to all discussion about the homework. We encourage you to complete the homework to the best of your ability and then discuss your findings/questions with your peers on the forums.

Questions

Answer the following problems:

1. Practice Problem 3.30 (pg 223)
   The following code fragment occurs often in the compiled version of library routines:

   ```
   1     call next
   2   next:
   3     popl %eax
   ```

   A. To what value does register `%eax` get set?
   B. Explain why there is no matching `ret` instruction to this call.
   C. What useful purpose does this code fragment serve?
   D. Is the stack affected after executing these two instructions? If so, how? If not, why not?
2. Practice Problem 3.31 (pg 224)
   The following code sequence occurs right near the beginning of the assembly code generated by GCC for a C procedure.

```
1     subl  $12, %esp
2     movl  %ebx, (%esp)
3     movl  %esi, 4(%esp)
4     movl  %edi, 8(%esp)
5     movl  8(%ebp), %ebx
6     movl  12(%ebp), %edi
7     movl  (%ebx), %esi
8     movl  (%edi), %eax
9     movl  16(%ebp), %edx
10    movl  (%edx), %ecx
```

We see that just three registers (`%ebx` , `%esi` , and `%edi`) are saved on the stack (lines 2-4). THe program modifies these and three other registers (`%eax` , `%ecx` , and `%edx`). At the end of the procedure, the values of registers `%edi` , `%esi` , and `%ebx` are restored (not shown), while the other three are left in their modified states.

Explain this apparent inconsistency in the saving and restoring of register states.

3. Practice Problem 3.33 (pg 228)
   Given the C function

```
1   int proc(void)
2   {
3       int x,y;
4       scanf("%x %x", &y, &x);
5       return x-y;
6   }
```

GCC generates the following assembly code:

```
1   proc:
2     pushl  %ebp
3     movl  %esp, %ebp
4     subl   $40, %esp
5     leal  -4(%ebp), %eax
6     movl  %eax, 8(%esp)
7     leal  -8(%ebp), %eax
8     movl  %eax, 4(%esp)
9     movl  $.LC0, (%esp)
10    call  scanf
 Diagram stack frame at this point
11    movl  -4(%ebp), %eax
12    subl  -8(%ebp), %eax
13    leave
14    ret
```

Assume that procedure `proc` starts executing with the following register values:

Register	Value
`%esp`	`0x800040`
`%ebp`	`0x800060`

Suppose `proc` calls `scanf` (line 10), and that `scanf` reads values `0x46` and `0x53` from the standard input. Assume that the string `"%x %x"` is stored at memory location `0x300070`.

A. What vlaue does `%ebp` get set to on line 3?
B. What value does `%esp` get set to on line 4?
C. At what addresses are local variables `x` and `y` stored?
D. Draw a diagram of the stack frame for `proc` right after `scanf` returns. Include as much information as you can about the addresses and the contents of the stack frame elements.
E. Indicate the regions of the stack frame that are not used by `proc`.

GCC allocates unused space on the stack in order to assure alignment of data - therefore all allocations on the stack frame are multiples of 16 bytes. Here 40 bytes are allocated in addition to the 4 for the return address and the 4 for the saved %ebp - for a total of 48. If we wanted to eliminate the unused space in `proc`'s stack frame ignoring alignment concerns, how would you change the assembly code to do this?

4. Practice Problem 3.42 (pg 251)
For the structure declaration

```
struct {
  char       *a;
  short       b;
  double      c;
  char        d;
  float       e;
  char        f;
  long long   g;
  void       *h;
} foo;
```

suppose it was compiled on a Windows machine, where each primitive data type of *K* bytes must have an offset that is a multiple of *K*.

A. What are the byte offsets of all the fields in the structure?
B. What is the total size of the structure?
C. Rearrange the fields of the structure to minimize wasted space, and then show the byte offsets and total size fo the rearranged structure.
D. What was your strategy (if any) to reduce wasted space? Would this always work? If not, can you describe a general strategy such that the space the struct uses is minimized?

5. Practice Problem 3.51 (pg 289)
For the C program

```
long int local_array(int i)
```

```
{
    long int a[4] = {2L, 3L, 5L, 7L};
    int idx = i & 3;
    return a[idx];
}
```

GCC generates the following code:

```
  x86-64 implementation of local_array
  Argument: i in %edi
1   local_array:
2     movq  $2, -40(%rsp)
3     movq  $3, -32(%rsp)
4     movq  $5, -24(%rsp)
5     movq  $7, -16(%rsp)
6     andl  $3, %edi
7     movq  -40(%rsp,%rdi,8), %rax
8     ret
```

A. Draw a diagram indicating the stack locations used by this function and their offsets relative to the stack pointer.
B. Annotate the assembly code to describe the offect of each instruction.
C. What interesting feature does this example illustrate about the x86-64 stack discipline?
D. What value is returned if the function is called with `i=6`?

6. Homework Problem 3.64 (pg 302)
For this exercise, we will examine the code generated by GCC for functions that have structures as arguments and return values, and from this see how these language features are typically implemented.

The following C code has a function `word_sum` having structures as argument and return values, and a function `prod` that calls `word_sum`:

```
typedef struct {
    int a;
    int *p;
} str1;

typedef struct {
    int sum;
    int diff;
} str2;

str2 word_sum(str1 s1) {
    str2 result;
    result.sum = s1.a + *s1.p;
    result.diff = s1.a - *s1.p;
```

```
    return result;
}

int prod {int x, int y)
{
    str1 s1;
    str2 s2;
    s1.a = x;
    s1.p = &y;
    s2 = word_sum(s1);
    return s2.sum * s2.diff;
}
```

GCC generates the following code for these two functions:

```
1   word_sum:
2     pushl %ebp
3     movl  %esp, %ebp
4     pushl %ebx
5     movl  8(%ebp), %eax
6     movl  12(%ebp), %ebx
7     movl  16(%ebp), %edx
8     movl  (%edx), %edx
9     movl  %ebx, %ecx
10    subl  %edx, %ecx
11    movl  %ecx, 4(%eax)
12    addl  %ebx, %edx
13    movl  %edx, (%eax)
14    popl  %ebx
15    popl  %ebp
16    ret   $4
```

```
1   prod:
2     pushl %ebp
3     movl  %esp, %ebp
4     subl  $20, %esp
5     leal  12(%ebp), %edx
6     leal  -8(%ebp), %ecx
7     movl  8(%ebp), %eax
8     movl  %eax, 4(%esp)
9     movl  %edx, 8(%esp)
10    movl  %ecx, (%esp)
11    call  word_sum
12    subl  $4, %esp
13    movl  -4(%ebp), %eax
14    imull  -8(%ebp), %eax
15    leave
16    ret
```

The instruction `ret $4` is like a normal return instruction, but it increments the stack pointer by 8 (4 for the return address plus 4 additional), rather than 4.

A. We can see in lines 5-7 of the code for `word_sum` that it appears as if three values are being retrieved from the stack, even though the function has only a single argument. Describe what these three values are.
B. We can see in line 4 of the code for `prod` that 20 bytes are allocated in the stack frame. These get used as five fields of 4 bytes each. Describe how each of these fields gets used.
C. How would you describe the general strategy for passing structures as arguments to a function?
D. How would you describe the general strategy for handling a structure as a return value from a function?

Created Wed 1 May 2013 12:23 PM PDT
Last Modified Wed 16 Jul 2014 9:58 AM PDT