Functions

Ray Seyfarth

August 4, 2011

Functions

- We will write C compatible function
- C++ can also call C functions using "`extern "C" {...}`"
- It is generally not sensible to write complete assembly programs
  - About 10% of your program uses 90% of the time
  - The compiler does an excellent job of code generation
  - Writing about 10% of your application in assembly might be worth doing if you can take advantage of instructions like SSE or AVX
- We will write functions which can be called from C
- We will also take advantage of C library functions
  - `malloc` to allocate memory
  - `scanf` to read data
  - `printf` to print data

Outline

1. The stack
2. The call instruction
3. The return instruction
4. Function parameters
5. Stack frames
6. Recursion

The stack

- The run-time stack is a region of memory which is used for a variety of temporary storage needs
- It starts with a high address of `0x7fffa6b79000` for my `bash` process
- It can be used for temporary storage of partially computed expressions
- It is used for some of the parameters to functions
- It is used for local variables in C/C++ functions
- It is used to store the address to return to after completing a function call
- The `push` instruction decrements the `rsp` register and stores the value being pushed at this address
- The `pop` instruction places the value at the top of the stack into its operand and increments `rsp`
- With the x86-64 instructions you should push and pop 8 bytes at a time

Initial stack setup

- The operating system starts a process by creating a stack with possibly randomly selected starting addresses
- Then it places a variety of data items into the stack.
- Finally it transfer to `_start` (not really a call)
- The parameters to `_start` are placed on the stack.
- The first parameter (last pushed on the stack) in the number of command line parameters
- The second parameter is the address of the string (on the stack) which is the first command line parameter (program name)
- These command line parameters continue and end with a 0 value on the stack.
- Above this point on the stack are addresses of the strings which constitute the environment
  - Strings like `"USER=seyfarth"`
  - Or `"PATH=/bin:/usr/bin:/usr/local/bin"` with multiple parts
  - All these variables were contained in the starting process
  - A child process inherits an environment

The call instruction

- After preparing any parameters you call a function this way

```
    call    my_function
```

- my_function should be an appropriate address in the code segment
- The function's return value will be in rax or xmm0
- The effect of a function call is much like

```
    push    next_instruction
    jmp     my_function
next_instruction:
```

The return instruction

- The effect of the return instruction (`ret`) is to pop an address off the stack and branch to it
- We could get much the same effect using

```
pop    rdi
jmp    rdi
```

Function parameters

- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.
- Linux and Mac OS/X pass integer and address parameters 1 through 6 in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`
- The remaining integer and address parameters are pushed onto the stack
- The first 6 floating point parameters are passed in registers `xmm0` - `xmm5`
- The remaining floating point parameters are passed on the stack
- Windows uses registers `rcx`, `rdx`, `r8` and `r9` for the first 4 integer and address parameters and pushes the rest
- Windows uses `xmm0` - `xmm3`
- In all cases pushed parameters are pushed in reverse order

Function parameters (2)

- Functions like `printf` having a variable number of parameters must place the number of floating point parameters in `rax`
- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions
- The reason behind this requirement is to make it possible for local variables (on the stack) to be on 16 byte boundaries, a requirement for some SSE and AVX instructions
- Conforming functions generally start with "`push rbp`" re-establishes the 15 byte bounding temporarily botched by the function call
- Following that conforming functions subtract multiple of 16 from `rsp` to allocate stack space or push pairs of 8 byte values

Hello world, at last

```
        section .data
msg:    db      "Hello World!",0x0a,0

        section .text
        global  main
        extern  printf
main:
        push    rbp
        mov     rbp, rsp
        lea     rdi, [msg]   ; parameter 1 for printf
        xor     eax, eax     ; 0 floating point parameters
        call    printf
        xor     eax, eax     ; return 0
        pop     rbp
        ret
```


Stack frames

- Stack frames are used by the gdb debugger to trace backwards through the stack to inspect calls make in a process
- The set of stack frames is accessible using the rbp register which contains the previous value of rsp
- At the previous rsp location is stored the old value of rbp for the previous function
- Just above the previous rbp is the return address
- The rbp addresses give a linked list of stack frames which works great with the backtrace or bt command in gdb
- Your functions should look like

```
push    rbp
mov     rbp, rsp
sub     rsp, multiple_of_16
...
leave                 ; undoes the first 3 instructions
ret
```

Symbolic names for local variables

- Local variables in a function are at rsp and above
- Use the equ pseudo-op to give names to their offsets relative to rsp

```
a     equ     0
b     equ     8
c     equ     16
d     equ     24
      push    rbp
      mov     rbp, rsp
      sub     rsp, 32
      mov     [rsp+a], rdi ; stores the first parameter in a
      mov     [rsp+b], rsi ; save the second parameter
      mov     rdi, 16
      call    malloc
      mov     [rsp+d], rax ; save address returned by malloc
      leave
      ret
```

Register preservation

- For Linux a function must preserve registers `rbx`, `rbp`, and `r12-r15`
- Try to dodge them, but if you need them place them in local variables on the stack first and restore before you leave
- It can be a relief to use these registers since they will still be available to you after a function call
- Windows functions must preserve registers `rbx`, `rbp`, `rsi`, `rdi` and `r12-r15`

Recursion

- A recursive function calls itself (perhaps indirectly)
- Using proper stack frames can help in debugging, especially with recursion
- Recursive solutions involve breaking a big problem into smaller problems, solving the smaller problems and building a complete solution from the sub-solutions
- If you break a problem up enough it generally becomes obvious how to solve it
- Perhaps you are defining a recursive sum of array elements. When you get down to 0 array elements it is easy to solve.
- These easy cases are called "base cases"
- A recursive function begins by checking if it is being asked to solve a base case
- If so, then it produces an immediate solution
- If not, then it applies recursion on sub-problems

Recursive factorial function

```
fact:                                   ; recursive function
n       equ     8
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16                 ; make room for storing n
        cmp     rdi, 1                  ; compare argument with 1
        jg      greater                 ; if n <= 1, return 1
        mov     eax, 1                  ; set return value to 1
        leave
        ret
greater:
        mov     [rsp+n], rdi            ; save n
        dec     rdi                     ; call fact with n-1
        call    fact
        mov     rdi, [rsp+n]            ; restore original n
        imul    rax, rdi                ; multiply fact(n-1)*n
        leave
        ret
```