The Stack and Functions

September 9, 2015
The Stack

- Up until now we have mostly ignored the stack.
- Instead we have relied on the data/bss segment.
- This has been sufficient so far, but this will change.
- We are now in a position where our programs will start being complex enough that we cannot rely entirely on fixed size allocations.
Consider the following (intentionally inefficient) recursive C++ function.

```cpp
// assuming fib(1)=1
int fib(int n)
{
    if(n<2)
        return 1;
    int fibM1=fib(n-1);
    int fibM2=fib(n-2);
    return fibM1+fibM2;
}
```

If $N$ is too large our program will experience a stack overflow.
The Stack in a Familiar Context

- But why?

There are two causes (though in a sense they are the same):

- The more obvious one is each execution of the fib will push at least two integers onto the stack. One for each of \( \text{FibM1} \) and \( \text{FibM2} \). Now the space complexity of this naive fibincci algorithm is \( O(n) \). We can try and roughly calculate \( \text{fib}(10000000) \) we need \( 2 \times 4 \text{bytes} \times 10000000 = 76.29 \text{mb} \) just to store the max required number of \( \text{FibM1} \) and \( \text{FibM2} \).

- Most Linux distros have a stack between 8 mb and 16 mb

What if I implement my recurrence function to somehow not push any data onto the stack?

- Even if this was possible there is one stack item we would have to push.
  - The return address.
But why?

There are two causes (though in a sense they are the same)

- The more obvious one is each execution of the \( \text{fib} \) will push at least two integers onto the stack. One for each of FibM1, and FibM2. Now the space complexity of this naive fibonacci algorithm is \( O(n) \). We can try and roughly calculate \( \text{fib}(10000000) \) we need \( 2 \times 4 \text{bytes} \times 10000000 = 76.29 \text{mb} \) just to store the max required number of FibM1, and FibM2.
  - Most Linux distros have a stack between 8mb and 16mb.
The Stack in a Familiar Context

- But why?
- There are two causes (though in a sense they are the same)
  - The more obvious one is each execution of the fib will push at least two integers onto the stack. One for each of FibM1, and FibM2. Now the space complexity of this naive fibonacci algorithm is $O(n)$. We can try and roughly calculate $fib(10000000)$ we need $2 \times 4 \text{bytes} \times 10000000 = 76.29 \text{mb}$ just to store the max required number of FibM1, and FibM2.
    - Most Linux distros have a stack between 8mb and 16mb
- What if I implement my recurrence function to some how not push any data onto the stack?
  - Even if this was possible there is one stack item we would have to push.
  - The return address.
The Stack

- Now that we have seen how it can break, lets us use the stack.
- The first point to recall is that the stack starts at :0x7fffffff. Unless there is some stack randomization in place.
- The second point is that the \textit{rsp} register stores the stack pointer.
- If nothing has been added to the stack \textit{rsp} =0x7ffffffffffff.
- We can interact with the stack in a number of ways, simplest of which are the \texttt{push} and \texttt{pop} instructions.
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.
A Stack Example

Example with start of the stack at 0x7ffffffff

;A
mov rax,74
push rax
;B
inc rax
push rax
;C
inc rax
push rax
;D
pop rax
;E
pop rax
;F
The Stack

- Stack space is often reserved for local variables by subtracting the size needed from the stack pointer (rsp).
- Then an offset is used to refer to the variables.

```
sub rsp, 16 ;subtract 16 bytes
mov [rsp+8], 123 ;set out first qword variable to 123
mov [rsp], 24 ;set out second qword variable to 24
```
The stack

<table>
<thead>
<tr>
<th>Address</th>
<th>0x7fffffffffffffff0</th>
<th>0x7fffffffffffffff8</th>
<th>0x7fffffffffffffff0</th>
<th>.</th>
<th>.</th>
<th>.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>sub rsp, 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td>mov [rsp+8], 123</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mov [rsp], 24</td>
</tr>
<tr>
<td></td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

64 Bit Intel Assembly Language
The Stack

- Remembering offsets for a large number of variables can become a burden. Instead use the `equ` pseudo-op.

```assembly
first  equ 8
second equ 0
mov [rsp+first],123
mov [rsp+second],24
```

How do we delete the variables after use?

▶ We just move the stack pointer back.

```assembly
add rsp,16
```
Remembering offsets for a large number of variables can become a burden. Instead use the equ pseudo-op.

```assembly
define first equ 8
define second equ 0
mov [rsp+first],123
mov [rsp+second],24
```

How do we delete the variables after use?
The Stack

- Remembering offsets for a large number of variables can become a burden. Instead use the equ pseudo-op.
  ```
  first equ 8
  second equ 0
  mov [rsp+first],123
  mov [rsp+second],24
  ```
- How do we delete the variables after use?
  - We just move the stack pointer back.
    ```
    add rsp,16
    ```
Assuming have loaded our parameters (will be explained shortly)

You call a function using

```assembly
call    my_function
```

`my_function` should be an appropriate address/label in the code segment

The function’s return value will be in `rax` or `xmm0`

The effect of a function call is much like

```assembly
push    next_instruction
jmp     my_function
next_instruction:
```
The Return Instruction

- You can return to the location a function was called from using `ret`.
- 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

- How Do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack.
Function Parameters

- How Do we get our parameters to our function?
- 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.

Windows uses registers rcx, rdx, r8, and r9 for the first 4 integer and address parameters.

The remaining integer and address parameters are pushed onto the stack.

Windows uses xmm0 - xmm3.

In all cases pushed parameters are pushed in reverse order.
How Do we get our parameters to our function?

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
Function Parameters

- How Do we get our parameters to our function?
- 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
- Windows uses registers rcx, rdx, r8 and r9 for the first 4 integer and address parameters
  - The remaining integer and address parameters are pushed onto the stack. (there is also a set amount of stack padding)
  - Windows uses xmm0 - xmm3
Function Parameters

- How Do we get our parameters to our function?
- 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
- Windows uses registers rcx, rdx, r8 and r9 for the first 4 integer and address parameters
  - The remaining integer and address parameters are pushed onto the stack. (there is also a set amount of stack padding)
  - 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`.

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Preserved across function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rax</code></td>
<td>temporary register; with variable arguments passes information about the number of vector registers used; 1st return register</td>
<td>No</td>
</tr>
<tr>
<td><code>rbx</code></td>
<td>callee-saved register; optionally used as base pointer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>rcx</code></td>
<td>used to pass 4th integer argument to functions</td>
<td>No</td>
</tr>
<tr>
<td><code>rdx</code></td>
<td>used to pass 3rd argument to functions; 2nd return register</td>
<td>No</td>
</tr>
<tr>
<td><code>rsp</code></td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>rbp</code></td>
<td>callee-saved register; optionally used as frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>rsi</code></td>
<td>used to pass 2nd argument to functions</td>
<td>No</td>
</tr>
<tr>
<td><code>rdi</code></td>
<td>used to pass 1st argument to functions</td>
<td>No</td>
</tr>
<tr>
<td><code>r8</code></td>
<td>used to pass 5th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td><code>r9</code></td>
<td>used to pass 6th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td><code>r10</code></td>
<td>temporary register, used for passing a function’s static chain pointer</td>
<td>No</td>
</tr>
<tr>
<td><code>r11</code></td>
<td>temporary register</td>
<td>No</td>
</tr>
<tr>
<td><code>r12-r15</code></td>
<td>callee-saved registers</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Simple function that returns the larger of two longs.

; long max(long a, long b)
max:
    mov rax, rdi ; move parm1 to rax
    cmp rax, rsi ; compare rax to parm2
    cmovl rax, rsi ; if parm2 > rax then move parm 2 to rax
    ret
Simple Function

Calling the simple function

```
mov rdi, 123 ; load parm1
mov rsi, 742 ; load parm2
call max
```
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 (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) instructions.

Conforming functions generally start with "push rbp" which re-establishes the 16 byte bounding temporarily botched by the function call. Following that, conforming functions subtract multiples of 16 from rsp to allocate stack space or push pairs of 8 byte values.
Real World Functions

- 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,
Real World Functions

- 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 (Streaming SIMD Extensions) and AVX (Advanced Vector Extension) instructions.
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(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.

Conforming functions generally start with “push rbp” which re-establishes the 16 byte bounding temporarily botched by the function call.
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(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.

Conforming functions generally start with “push rbp” which re-establishes the 16 byte bounding temporarily botched by the function call

- Remember, the call operation pushes a 8 byte value onto the stack (the return address)
Real World Functions

- 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(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.

- Conforming functions generally start with “push rbp” which re-establishes the 16 byte bounding temporarily botched by the function call
  - Remember, the call operation pushes a 8 byte value onto the stack (the return address)

- 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 ; will explain shortly
mov rdi, msg ; parameter 1 for printf
xor eax, eax ; 0 floating point parameters
call printf
xor eax, eax ; return 0
mov rsp, rbp ; will explain shortly (NIB)
pop rbp
ret
Stack frames

- Stack frames are used by the gdb debugger to trace backwards through the stack to inspect calls made in a process.
- If we start and end each function like:

  ```
push rbp
mov rbp, rsp

... 
mov rsp, rbp
pop rbp
ret
  ```

- We are in effect constructing a link list of all of the stack frames.
### Stack frames

<table>
<thead>
<tr>
<th></th>
<th>In base function:</th>
<th>In Function L1:</th>
<th>In Function L2:</th>
<th>In Function L3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp</td>
<td>0x0</td>
<td>StackPointer(0)</td>
<td>StackPointer(1)</td>
<td>StackPointer(2)</td>
</tr>
<tr>
<td>rsp</td>
<td>StackPointer(0)</td>
<td>StackPointer(1)</td>
<td>StackPointer(2)</td>
<td>StackPointer(3)</td>
</tr>
<tr>
<td>0x7fffffff0</td>
<td>x</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffffef8</td>
<td>x</td>
<td>x</td>
<td>StackPointer(0)</td>
<td>StackPointer(0)</td>
</tr>
<tr>
<td>0x7fffffffef0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>StackPointer(1)</td>
</tr>
<tr>
<td>0x7fffffffdf8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>0x7fffffffdf0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Stack frames

- If you require space for local variables you simply need to subtract an amount from the stack pointer `rsp`.

```assembly
push rbp
mov rbp, rsp
sub rsp, 16
```

If we allocate local variables we must use
```
mov rsp, rbp
pop rbp
ret
```

Just popping will not work.
If you require space for local variables you simply need to subtract an amount from the stack pointer \texttt{rsp}.

- The subtraction should always maintain the 16 byte boundary.

- \texttt{push rbp}
- \texttt{mov rbp, rsp}
- \texttt{sub rsp, 16}

If we allocate local variables we must use

- \texttt{mov rsp, rbp}
- \texttt{pop rbp}
- \texttt{ret}

Just popping will not work.
Stack frames

- If you require space for local variables you simply need to subtract an amount from the stack pointer `rsp`.
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable.

```assembly
push rbp
mov rbp, rsp
sub rsp, 16
```

If we allocate local variables we must use:

```assembly
mov rsp, rbp
pop rbp
ret
```

Just popping will not work.
Stack frames

- If you require space for local variables you simply need to subtract an amount from the stack pointer \( rbp \).
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable. We only **have** to subtract 8 bytes.
If you require space for local variables you simply need to subtract an amount from the stack pointer `rsp`.

- The subtraction should always maintain the 16 byte boundary.
- For example, say we wish to have a single quadword as a local variable.
  - We only have to subtract 8 bytes.
  - But we should maintain the 16 byte boundary.

```assembly
push rbp
mov rbp, rsp
sub rsp, 16
```
Stack frames

- If you require space for local variables you simply need to subtract an amount from the stack pointer rsp.
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable.
    - We only **have** to subtract 8 bytes.
    - But we **should** maintain the 16 byte boundary.

  ```
  push    rbp
  mov     rbp, rsp
  sub     rsp, 16
  ```

- If we allocate local variables we **must** use

  ```
  mov     rsp, rbp
  pop     rbp
  ret
  ```

  Just popping will not work.
If you prefer you can utilize

```
leave
ret
```

instead of

```
mov rsp, rbp
pop rbp
ret
```
main:
    push rbp
    mov rbp, rsp

; print_max ( 100, 200 );
    mov rdi, 100 ; first parameter
    mov rsi, 200 ; second parameter
    call print_max
    xor eax, eax ; to return 0
leave
ret
; void print_max ( long a, long b )
{
    a equ 0
    b equ 8
    print_max:
        push rbp
        mov rbp, rsp
        sub rsp, 32 ; leave space for a, b and max
        max equ 16 ; long max;
        mov [rsp+a], rdi ; save a
        mov [rsp+b], rsi ; save b
        mov [rsp+max], rdi ; max = a;
        cmp rsi, rdi ; if ( b > max ) max = b
        jng skip
        mov [rsp+max], rsi

print max example

skip:
segment .data
   fmt db "max(%ld,%ld) = %ld",0xa,0
segment .text
   mov rdi, fmt
   mov rsi, [rsp+a]
   mov rdx, [rsp+b]
   mov rcx, [rsp+max]
   xor eax, eax
   call printf
   leave
   ret
Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?

Solution?

- Simply use \( rbp \) as the base address. (But you will also have to cater for stack subtractions.)
Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?
  - If anything is pushed onto the stack our offsets become incorrect.
Can you see the potential issue with the way we are accessing our local variables?

- If anything is pushed onto the stack our offsets become incorrect.

Solution?
Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?
  - If anything is pushed onto the stack our offsets become incorrect.

- Solution?
  - Simply use rbp as the base address. (But you will also have to cater for stack subtractions)