The mysterious second parameter to the x86 ENTER instruction

devblogs.microsoft.com/oldnewthing/20231211-00

December 11, 2023

Raymond Chen

The x86 instruction set has an `ENTER` instruction which builds a stack frame. It is almost always used with a zero as the second parameter.

```
        enter   n, 0
```

This is functionally equivalent to

```
        push    ebp
        mov     ebp, esp
        sub     esp, n
```

But what happens if you increase that second parameter beyond zero?

Values greater than zero for the second parameter are intended for languages like Pascal which support nested functions that can access the local variables of their lexical parents. We learned about these functions a short time ago. But the designers of the x86 instruction set had a different design in mind for how a function can access the variables of its lexical parent: Instead of receiving a pointer to the start of a linked list of lexical parent frames, they receive an *array* of pointers to lexical parent frames.

In its full generality, the

```
    enter n, k + 1
```

instruction goes like this:

```
    push    ebp
    mov     internal_register, esp
    sub     ebp, 4   ⎱ k times
    push    [ebp]    ⎰
    push    internal_register
    mov     ebp, internal_register
    sub     esp, n
```

If you ignore the order of operations and worry just about the final state, then you can reinterpret it like this, which I think captures the essence of the instruction better:

```
    push    ebp
    push    [ebp-4]
    push    [ebp-8]    k pushes
    :
    push    [ebp-4*k]
    lea     ebp, [esp + 4*k]    ; where we pushed the previous ebp
    push    ebp                 ; add our own frame to the array
    sub     esp, n
```

Let’s look at our example function again.

```
function Outer(n: integer) : integer;
    var i: integer;

    procedure Update(j: integer);
    begin
        i := i + j
    end;

    procedure Inner(m: integer);

        procedure MoreInner;
        begin
            Update(m)
        end;

    (* Inner body begins here *)
    begin
        MoreInner
    end;

(* Outer body begins here *)
begin
    i := 0;
    Inner(n);
    Outer := i
end;
```

On entry to `Outer`, the stack looks like this:

`n` parameter	
return address	← *esp*

The `Outer` function establishes its stack frame by performing an `enter 4, 1`. The extra `1` at the end means that this is the outermost of a chain of nested functions. In our cookbook, *k* is zero, so the functional equivalent is

```
push    ebp
                                        ; no pointers copied from parent
lea     ebp, [esp+0]        ; equivalently, "mov ebp, esp"
push    ebp                 ; pointer to our own frame
sub     esp, 4
```

and we wind up with this stack frame for `Outer`:

`Outer` frame	
`n` parameter	
return address	
▶ previous *ebp*	← *ebp*
`Outer` frame pointer	
`i`	← *esp*

That extra `,1` caused us to push the address of where we saved the previous `ebp`, which I've called the `Outer` frame pointer. That value isn't really useful to us right now, since we already have that value in the *ebp* register. But it comes in handy when we call `Inner`.

On entry to `Inner`, the stack looks like this:

`m` parameter	
return address	← *esp*

The `Inner` function performs an `enter 0, 2`. The 0 means that `Inner` has no local variables, and the `2` means that we are now the second level in a chain of nested functions.

The functional equivalent now has one extra memory push before we push a pointer to our own frame:

```
push    ebp
push    [ebp-4]             ; one pointer copied from parent
lea     ebp, [esp+4]
push    ebp                 ; pointer to our own frame
sub     esp, 4
```

Before pushing the address of its own frame, the `enter` instruction also copies one pointer from the parent's frame, namely the `Outer` frame pointer.

`Inner` frame		
`m` parameter		`Outer` frame
return address		`n` parameter
▶ previous *ebp*	← *ebp*	return address
`Outer` frame pointer		▶ previous *ebp*
`Inner` frame pointer	← *esp*	`Outer` frame pointer
		`i`

Now things are interesting.

The `Inner` function has access to its own frame, via the *ebp* register (and redundantly via the `Inner` frame pointer on its stack). It also has access to the `Outer` frame through its local copy of the `Outer` frame pointer.

The next thing that happens is that `Inner` calls `MoreInner` with no parameters. This time `MoreInner` uses `enter 0, 3` where the 0 means that `MoreInner` has no local variables, and the `3` means that it is a nested function three levels deep, so it should copy *two* frame pointers from its parent.

`MoreInner` frame		
return address	`Inner` frame	
▶ previous *ebp*	`m` parameter	`Outer` frame

`Outer` frame pointer	return address	`n` parameter
`Inner` frame pointer	▶ previous *ebp*	return address
`MoreInner` frame pointer	`Outer` frame pointer	▶ previous *ebp*
	`Inner` frame pointer	`Outer` frame pointer
		`i`

The frame for `MoreInner` contains its own parameters and local variables (nothing), plus pointers to both parent frames, plus a pointer to its own frame (which `MoreInner` doesn't use, but which is ready for any nested function to use).

The code generation for `MoreInner` therefore reads the value of `m` by following the `Inner` frame pointer and then reading the `m` parameter from the `Inner` frame's parameter space.

After `MoreInner` calls `Update`, the `Update` function starts with an `enter 0, 2` because it is a level-2 nested function. This copies only the `Outer` frame pointer to `Update`'s frame, resulting in this:

`Update` frame		
`j` parameter		`Outer` frame
return address		`n` parameter
▶ previous *ebp*	← *ebp*	return address
`Outer` frame pointer		▶ previous *ebp*
`Update` frame pointer	← *esp*	`Outer` frame pointer

	`i`

I didn’t draw it, but the “previous *ebp*” in the `Update` frame points to the `MoreInner` frame.

The `Update` function reads `j` from its own parameter space and uses to update the `i` variable in `Outer`‘s frame by following the `Outer` frame pointer.

The result is the same as the System V Application Binary Interface static chain pointer, but it’s done in a different way. Instead of passing the head of a linked list of frames, the `enter` instruction copies an entire array of pointers to frames. This reduces the number of instructions required in order to access faraway frames, but it increases the cost of a function call due to the extra copying.

I wonder if anybody uses the Intel design for nested functions. I suspect it’s silicon on the CPU that is completely wasted.