The history of calling conventions, part 5: amd64

devblogs.microsoft.com/oldnewthing/20040114-00

January 14, 2004

Raymond Chen

The last architecture I'm going to cover in this series is the AMD64 architecture (also known as x86-64).

The AMD64 takes the traditional x86 and expands the registers to 64 bits, naming them rax, rbx, etc. It also adds eight more general purpose registers, named simply R8 through R15.

- The first four parameters to a function are passed in rcx, rdx, r8 and r9. Any further parameters are pushed on the stack. Furthermore, space for the register parameters is reserved on the stack, in case the called function wants to spill them; this is important if the function is variadic.
- Parameters that are smaller than 64 bits are **not** zero-extended; the upper bits are garbage, so remember to zero them explicitly if you need to. Parameters that are larger than 64 bits are passed by address.
- The return value is placed in rax. If the return value is larger than 64 bits, then a secret first parameter is passed which contains the address where the return value should be stored.
- All registers must be preserved across the call, except for rax, rcx, rdx, r8, r9, r10, and r11, which are scratch.
- The callee does **not** clean the stack. It is the caller's job to clean the stack.
- The stack must be kept 16-byte aligned. Since the "call" instruction pushes an 8-byte return address, this means that every non-leaf function is going to adjust the stack by a value of the form 16n+8 in order to restore 16-byte alignment.

Here's a sample:

```
void SomeFunction(int a, int b, int c, int d, int e);
void CallThatFunction()
{
    SomeFunction(1, 2, 3, 4, 5);
    SomeFunction(6, 7, 8, 9, 10);
}
```

On entry to CallThatFunction, the stack looks like this:

xxxxxxx0	.. rest of stack ..	
xxxxxxx8	return address	<- RSP

Due to the presence of the return address, the stack is misaligned. CallThatFunction sets up its stack frame, which might go like this:

```
    sub    rsp, 0x28
```

Notice that the local stack frame size is 16n+8, so that the result is a realigned stack.

xxxxxxx0	.. rest of stack ..	
xxxxxxx8	return address	
xxxxxxx0		(arg5)
xxxxxxx8		(arg4 spill)
xxxxxxx0		(arg3 spill)
xxxxxxx8		(arg2 spill)
xxxxxxx0		(arg1 spill) <- RSP

Now we can set up for the first call:

```
        mov     dword ptr [rsp+0x20], 5     ; output parameter 5
        mov     r9d, 4                      ; output parameter 4
        mov     r8d, 3                      ; output parameter 3
        mov     edx, 2                      ; output parameter 2
        mov     ecx, 1                      ; output parameter 1
        call    SomeFunction                ; Go Speed Racer!
```

When SomeFunction returns, the stack is **not** cleaned, so it still looks like it did above. To issue the second call, then, we just shove the new values into the space we already reserved:

```
mov     dword ptr [rsp+0x20], 10    ; output parameter 5
mov     r9d, 9                      ; output parameter 4
mov     r8d, 8                      ; output parameter 3
mov     edx, 7                      ; output parameter 2
mov     ecx, 6                      ; output parameter 1
call    SomeFunction                ; Go Speed Racer!
```

CallThatFunction is now finished and can clean its stack and return.

```
add     rsp, 0x28
ret
```

Notice that you see very few "push" instructions in amd64 code, since the paradigm is for the caller to reserve parameter space and keep re-using it.

[**Updated 11:00am**: Fixed some places where I said "ecx" and "edx" instead of "rcx" and "rdx"; thanks to Mike Dimmick for catching it.]

Raymond Chen

Follow