Stack frame layout on x86-64 (https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64)

September 06, 2011 at 20:13 **Tags** Articles (https://eli.thegreenplace.net/tag/articles) , Assembly (https://eli.thegreenplace.net/tag/assembly) , Linux (https://eli.thegreenplace.net/tag/linux)

A few months ago I've written an article named Where the top of the stack is on x86 (http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/), which aimed to clear some misunderstandings regarding stack usage on the x86 architecture. The article concluded with a useful diagram presenting the stack frame layout of a typical function call.

In this article I will examine the stack frame layout of the newer 64-bit version of the x86 architecture, x64 [1]. The focus will be on Linux and other OSes following the official System V AMD64 ABI. Windows uses a somewhat different ABI, and I will mention it briefly in the end.

I have no intention of detailing the complete x64 calling convention here. For that, you will literally have to read the whole AMD64 ABI.

Registers galore

x86 has just 8 general-purpose registers available `(eax, ebx, ecx, edx, ebp, esp, esi, edi)`. x64 extended them to 64 bits (prefix "r" instead of "e") and added another 8 `(r8, r9, r10, r11, r12, r13, r14, r15)`. Since some of x86's registers have special implicit meanings and aren't *really* used as general-purpose (most notably `ebp` and `esp`), the effective increase is even larger than it seems.

There's a reason I'm mentioning this in an article focused on stack frames. The relatively large amount of available registers influenced some important design decisions for the ABI, such as passing many arguments in registers, thus rendering the stack less useful than before [2].

Argument passing

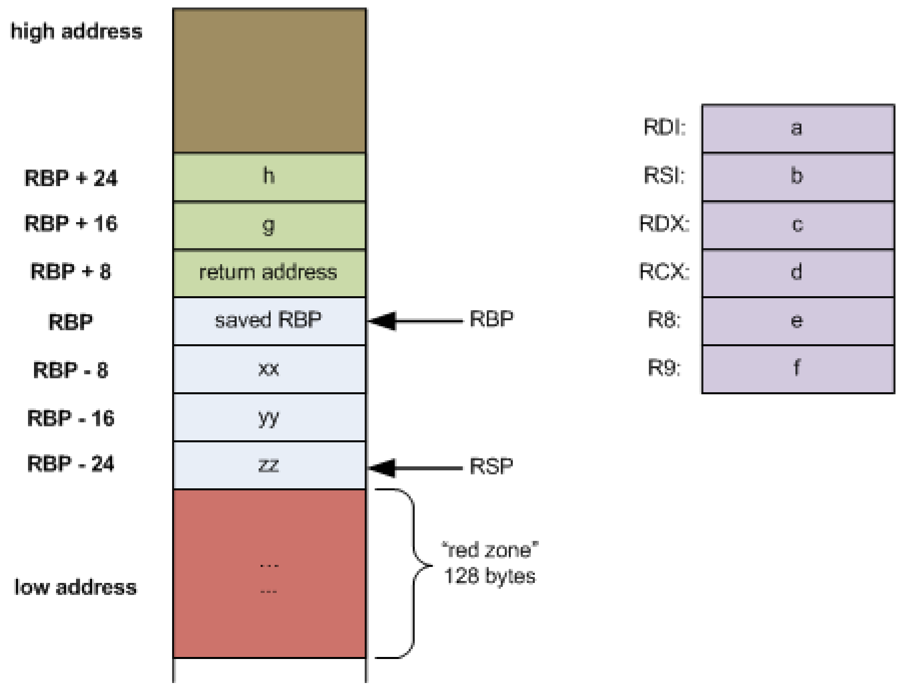
I'm going to simplify the discussion here on purpose and focus on integer/pointer arguments [3]. According to the ABI, the first 6 integer or pointer arguments to a function are passed in registers. The first is placed in `rdi`, the second in `rsi`, the third in `rdx`, and then `rcx`, `r8` and `r9`. Only the 7th argument and onwards are passed on the stack.

The stack frame

With the above in mind, let's see how the stack frame for this C function looks:

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```

This is the stack frame:



So the first 6 arguments are passed via registers. But other than that, this doesn't look very different from what happens on x86 [4], except this strange "red zone". What is that all about?

The red zone

First I'll quote the formal definition from the AMD64 ABI:

> The 128-byte area beyond the location pointed to by %rsp is considered to be reserved and shall not be modified by signal or interrupt handlers. Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue. This area is known as the red zone.

Put simply, the red zone is an optimization. Code can assume that the 128 bytes below `rsp` will not be asynchronously clobbered by signals or interrupt handlers, and thus can use it for scratch data, *without explicitly moving the stack pointer*. The last sentence is where the optimization lays - decrementing `rsp` and restoring it are two instructions that can be saved when using the red zone for data.

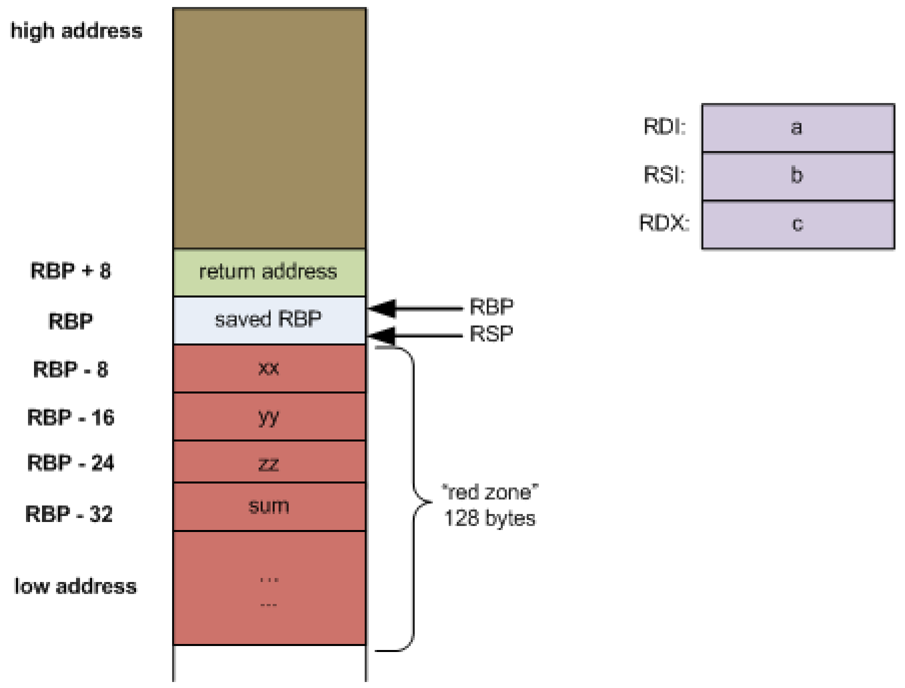
However, keep in mind that the red zone *will* be clobbered by function calls, so it's usually most useful in leaf functions (functions that call no other functions).

Recall how `myfunc` in the code sample above calls another function named `utilfunc`. This was done on purpose, to make `myfunc` non-leaf and thus prevent the compiler from applying the red zone optimization. Looking at the code of `utilfunc`:

```
long utilfunc(long a, long b, long c)
{
    long xx = a + 2;
    long yy = b + 3;
    long zz = c + 4;
    long sum = xx + yy + zz;

    return xx * yy * zz + sum;
}
```

This is indeed a leaf function. Let's see how its stack frame looks when compiled with `gcc`:



Since `utilfunc` only has 3 arguments, calling it requires no stack usage since all the arguments fit into registers. In addition, since it's a leaf function, `gcc` chooses to use the red zone for all its local variables. Thus, `rsp` needs not be decremented (and later restored) to allocate space for this data.

Preserving the base pointer

The base pointer `rbp` (and its predecessor `ebp` on x86), being a stable "anchor" to the beginning of the stack frame throughout the execution of a function, is very convenient for manual assembly coding and for debugging [5]. However, some time ago it was noticed that compiler-generated code doesn't really need it (the compiler can easily keep track of offsets from `rsp`), and the DWARF debugging format provides means (CFI) to access stack frames without the base pointer.

This is why some compilers started omitting the base pointer for aggressive optimizations, thus shortening the function prologue and epilogue, and providing an additional register for general-purpose use (which, recall, is quite useful on x86 with its limited set of GPRs).

`gcc` keeps the base pointer by default on x86, but allows the optimization with the `-fomit-frame-pointer` compilation flag. How recommended it is to use this flag is a debated issue - you may do some googling if this interests you.

Anyhow, one other "novelty" the AMD64 ABI introduced is making the base pointer explicitly optional, stating:

> The conventional use of %rbp as a frame pointer for the stack frame may be avoided by using %rsp (the stack pointer) to index into the stack frame. This technique saves two instructions in the prologue and epilogue and makes one additional general-purpose register (%rbp) available.

`gcc` adheres to this recommendation and by default omits the frame pointer on x64, when compiling with optimizations. It gives an option to preserve it by providing the `-fno-omit-frame-pointer` flag. For clarity's sake, the stack frames showed above were produced without omitting the frame pointer.

The Windows x64 ABI

Windows on x64 implements an ABI of its own, which is somewhat different from the AMD64 ABI. I will only discuss the Windows x64 ABI briefly, mentioning how its stack frame layout differs from AMD64. These are the main differences:

1. Only 4 integer/pointer arguments are passed in registers (`rcx, rdx, r8, r9`).
2. There is no concept of "red zone" whatsoever. In fact, the ABI explicitly states that the area beyond `rsp` is considered volatile and unsafe to use. The OS, debuggers or interrupt handlers may overwrite this area.
3. Instead, a "register parameter area" [6] is provided by the caller in each stack frame. When a function is called, the last thing allocated on the stack before the return address is space for at least 4 registers (8 bytes each). This area is available for the callee's use without explicitly allocating it. It's useful for variable argument functions as well as for debugging (providing known locations for parameters, while registers may be reused for other purposes). Although the area was originally conceived for spilling the 4 arguments passed in registers, these days the compiler uses it for other optimization purposes as well (for example, if the function needs less than 32 bytes of stack space for its local variables, this area may be used without touching `rsp`).

Another important change that was made in the Windows x64 ABI is the cleanup of calling conventions. No more `cdecl/stdcall/fastcall/thiscall/register/safecall` madness - just a single "x64 calling convention". Cheers to that!

For more information on this and other aspects of the Windows x64 ABI, here are some good links:

- Official MSDN page on x64 software conventions (http://msdn.microsoft.com/en-us/library/7kcdt6fy.aspx) - well organized information, IMHO easier to follow and understand than the AMD64 ABI document.
- Everything You Need To Know To Start Programming 64-Bit Windows Systems (http://msdn.microsoft.com/en-us/magazine/cc300794.aspx) - MSDN article providing a nice overview.
- The history of calling conventions, part 5: amd64 (http://blogs.msdn.com/b/oldnewthing/archive/2004/01/14/58579.aspx) - an article by the prolific Windows programming evangelist Raymond Chen.
- Why does Windows64 use a different calling convention from all other OSes on x86-64? (http://stackoverflow.com/questions/4429398/why-does-windows64-use-a-different-calling-convention-from-all-other-oses-on-x86) - an interesting discussion of the question that just begs to be asked.
- Challenges of Debugging Optimized x64 code (http://blogs.msdn.com/b/ntdebugging/archive/2009/01/09/challenges-of-debugging-optimized-x64-code.aspx) - focuses on the "debuggability" (and lack thereof) of compiler-generated x64 code.

[1] This architecture goes by many names. Originated by AMD and dubbed AMD64, it was later implemented by Intel, which called it IA-32e, then EM64T and finally Intel 64. It's also being called x86-64. But I like the name x64 - it's nice and short.

[2] There are calling conventions for x86 that also dictate passing some of the arguments in registers. The best known is probably `fastcall`. Unfortunately, it's not consistent across platforms.

[3] The ABI also defines passing floating-point arguments via the `xmm` registers. The idea is pretty much the same as for integers, however, and IMHO including floating-point arguments in the article will needlessly complicate it.

[4] I'm cheating a bit here. Any compiler worth its salt (and certainly `gcc`) will use registers for local variables as well, especially on x64 where registers are plentiful. But if there are a lot of local variables (or they're large, like arrays or structs), they will go on the stack anyway.

[5] Since inside a function `rbp` always points at the previous stack frame, it forms a kind of linked list of stack frames which the debugger can use to access the execution stack trace at any given time (in core dumps as well).

[6] Also called "home space" sometimes.