UNDERSTANDING STACK ALIGNMENT IN 64-BIT CALLING CONVENTIONS

For x86-64 Assembly Programmers

This article is part of CPU2.0 presentation
Soffian Abdul Rasad
Date : Oct 2nd, 2017
Updated : June 19th 2020

Note: This article is released in conjunction with CPU2.0 to enable users to use CPU2.0 correctly in regards to stack alignment. It does not present full stack programming materials.

MS 64-bit Application Binary Interface (ABI) defines an x86_64 software calling convention known as the *fastcall*. System V also introduces similar software calling convention known as *AMD64 calling convention*. These calling conventions specify the standards and the requirements of the 64-bit APIs to enable codes to access and use them in a standardized and uniform fashion. This is the standard convention being employed by Win64 APIs and Linux64 respectively. While the two are radically different, they share similar stack alignment policy – the stack must be well-aligned to 16-byte boundaries prior to calling a 64-bit API. **A memory address is said to be aligned to 16 if it is evenly divisible by 16, or if the last digit is 0 in hexadecimal notation.**

Please note that to simplify discussions, we ignore the requirement for *shadow space* or *red zone* when calling the 64-bit APIs for now. But still with or without the shadow space, the discussion below applies. Also note that the CPU honors no calling convention.

Align 16 Ecosystem

This requirement specifies that for the 64-bit APIs to function correctly it must work in an ecosystem or runtime environment in which the stack memory is always aligned to 16-byte boundaries. While applications can use the stack memory as they please, they must make sure that this alignment requirement is observed prior to calling any of the 64-bit API. In simpler words, *a caller has to make sure that the stack is well-aligned in its own function body prior to calling an 64-bit API or other ABI-compliant modules.*

Initiating an Aligned Eco-system

In most cases, an aligned ecosystem is initialized at the entry points where the codes start executing and / or where the stack memory is first allocated, either by the hosting OS, compilers or by the linkers. The stack alignment status is shown via the RSP register at that entry point. This initial RSP will give us the crucial information about the stack alignment at that point. It determines how an aligned environment should be created and eventually be observed by the entire application from that point on.

In a typical and controlled environment, the stack is either in align-8 or align-16 alignment at this point due to the CPUs that work on its own natural boundaries (multiples of 8).

Executives role

To support the 64-bit API alignment ecosystem, the OS, linkers or loaders often set the stack memory with addresses aligned to 8. That means, process or programs often start off with misaligned stack (that is aligned to 8 instead of aligned to 16).

```
main:
        [stack starts with align 8]
```

The rationale behind this align-8 (misaligned) allocation policy is ensure consistency in implementing stack alignment strategies by the subsequent modules. This is because any inconsistent initial alignment of the stack may require separate alignment strategies that affect all subsequent functions in the same environment or library. We will soon see why.

Alignment Chaining

Any modules wishing to use the 64-bit APIs must chain or hook their stack to the aligned ecosystem. That starts with the entry point or modules that contains such entry point. Therefore since the stack is misaligned from the beginning, functions will have to align it manually by subtracting the RSP by 8

```
main:
        [stack is misaligned or aligned to 8]
        sub rsp,8     ;align the stack to 16
        [stack now is aligned to 16]
        ...
        call A_FUNCTION
        add rsp,8
        ret
```

The code above demonstrates how a program initiates an aligned eco-system from the entry point (`main`) from an unaligned initial stack. From that point on, the stack is said to be aligned to 16 in the `main`'s body and the aligned ecosystem is enforced upon the stack. You can now theoretically call a 64-bit API that observes similar stack alignment requirement.

Calling Others

Recall that a `call` instruction is an aggregate or complex instruction that contains other instruction sequences. A `call` instruction is implementing this semantics at the lower level;

```
push   $+n           ;Save the return address (the next_instruction)
jmp    A_FUNCTION
next_instruction
```

The `push` instructions is actually a stack-based instruction that modifies the RSP implicitly by subtracting the stack by 8 bytes (further down) to keep the return address of the `next_instruction`. This action creates a *call stack*.

This unfortunately will harm the alignment ecosystem – from an aligned state back to misaligned state due to `push` instruction. By the time the execution path reaches the callee `A_FUNCTION`, the stack is in misaligned state.

The A_FUNCTION, upon seeing the misaligned stack, should attempt to chain its stack back to the aligned ecosystem by making a sub against the RSP.

```
A_FUNCTION:
        [stack is misaligned or aligned to 8]
        sub rsp,8           ;align the stack to 16
        [stack now is aligned to 16]
        …
        add rsp,8
        ret
```

If you look carefully, both main and A_FUNCTION are actually **doing the same exact thing** in regards to the stack alignment, that is by sub-bing the RSP by 8 upon entry.

```
main:
        [stack is initially misaligned or aligned to 8]
        sub rsp,8           ;align the stack to 16
        [stack now is aligned to 16]
        call A_FUNCTION     ;misalignment due to internal push
        add rsp,8
        ret

A_FUNCTION:
        [stack is misaligned, inherited from main]
        sub rsp,8           ;align the stack to 16
        [stack alignment is now restored to 16]
        add rsp,8
        ret
```

These actions to hook the stack back to the aligned ecosystem create a *chained efforts* where callees keep on chaining their stack to the other functions stack in order to maintain their presence in the aligned ecosystem. This will be more clear if we introduce another function, B_FUNCTION like below;

```
main:
        sub rsp,8           ;initiates align to 16 ecosystem
        call A_FUNCTION     ;misaligned stack due to PUSH
        add rsp,8
        ret

A_FUNCTION:
        sub rsp,8           ;chaining/restore alignment to 16
        call B_FUNCTION
        add rsp,8
        ret

B_FUNCTION:
        sub rsp,8           ;chaining
        ...
        add rsp,8
        ret
```

Verifying with CPU2.0

To help visualize how stack alignment works, we'll use CPU2.0s `dumpreg` routine to keep track of the stack movement in all three functions discussed above. We will use this initial code below where the stack alignment is **not** observed.

```
option casemap:none
externdef dumpreg:proc
externdef exitx:proc

.code
main proc
       call dumpreg
       call A_FUNCTION
       call exitx
main endp

A_FUNCTION proc
       call dumpreg
       call B_FUNCTION
       ret
A_FUNCTION endp

B_FUNCTION proc
       call dumpreg
       ret
B_FUNCTION endp
end
```

The output of the three `dumpreg` follows. Observe the RSP register.

```
--snip--
R11|0000000000000246 R12|0000000000000001 R13|0000000000000008
R14|0000000000000000 R15|0000000000000000 RBP|000000000E21350
RSP|000000000062FE58 RIP|0000000000402CE0 [UHEX]       ;in main

--snip--
R11|0000000000000246 R12|0000000000000001 R13|0000000000000008
R14|0000000000000000 R15|0000000000000000 RBP|000000000E21350
RSP|000000000062FE50 RIP|0000000000402CEF [UHEX]       ;A_FUNCTION

--snip--
R11|0000000000000246 R12|0000000000000001 R13|0000000000000008
R14|0000000000000000 R15|0000000000000000 RBP|000000000E21350
RSP|000000000062FE48 RIP|0000000000402CFA [UHEX]       ;B_FUNCTION
```

The first RSP from `main` confirms our previous discussion that the program starts with misaligned stack (align-8) from the entry point. When it executes the `call` (which has an internal `push`), the RSP is further reduced by 8, hence the output of RSP in `A_FUNCTION`. The RSP shows that the stack is in aligned to 16 condition in `A_FUNCTION`. We then call the `B_FUNCTION`, further reducing the RSP by 8, hence misaligning the stack by the time it reaches `B_FUNCTION`.

There's nothing wrong with the program above. It's a perfectly normal code but that's not a desirable conditions if we want to use the 64-bit API because the stack is not in the aligned ecosystem - the three functions above demonstrates *alternating alignments* (align 8 and align 16) in each of their bodies as evident by the RSP register.

Creating Alignment Chaining

To overcome that problem, each and every functions or modules wishing to communicate with the API must chain their stack together to maintain the aligned ecosystem.

First thing first, **the entry point must initiate such aligned ecosystem** (to 16) because everybody else will depend on it and will follow suit. This is done simply by using `sub rsp,8` instruction to re-align the stack to 16-bytes boundaries. This works on the assumption that the executives (OS, linkers, compilers etc) will always allocate the stack aligned to 8 (misaligned) at the entry points.

> **Note** Certain linkers provide an aligned stack from the entry point, by default. In this case, your `main` or entry point is already in aligned condition. Always verify RSP at entry points. If RSP is already in aligned state, there's no apparent need to chaining it via `sub rsp,8` instruction at the start.

```
main proc
       sub    rsp,8         ;align to 16
       call   dumpreg       ;verify RSP
       call   A_FUNCTION
       add    rsp,8         ;balancing
       ret
main endp
```

This effectively creates a stack that is aligned to 16 and initiates such aligned environment for everybody else to chain to and to maintain. Now what `main` has to do is to maintain such aligned state in its own body.

As discussed earlier, a call (to `A_FUNCTION`) will break the alignment for the callee every time. This creates exactly the same situation in `main` (entry point) where it starts off with unaligned stack from the executives. So what a callee does to restore the alignment is actually the same as the `main` function; that is by using `sub rsp,8` instruction.

```
A_FUNCTION proc
       sub    rsp,8  ;chain the stack back to aligned ecosystem
       call   dumpreg
       add    rsp,8  ;balancing before returns
       ret
A_FUNCTION endp
```

The same goes to other functions designed to be 64-bit API compliant. So if you're making such compliant functions, you will naturally repeat the stack alignment chaining demonstrated by both functions above.

```
NEW_FUNCTION proc
       sub    rsp,8
       ...
       add    rsp,8
       ret
NEW_FUNCTION endp
```

Or in low-level syntax;

```
NEW_FUNCTION:
       sub    rsp,8
       …
       add    rsp,8
       ret
```

If you look carefully, all functions are working exactly the same way and demonstrating the same stack chaining technique. This is exactly how functions and modules, both user's own and 64-bit APIs work to create, maintain and enforce the stack alignment. If for some reason that a call to the NEW_FUNCTION failed, then there's a chance that the caller does not work in the same alignment eco-system or simply being *non-compliant callers*.

The program below demonstrates the idea of such implementation. All stack (RSP) are well-aligned to 16 in each of the functions' body. In MASM syntax;

```
	option casemap:none
	externdef dumpreg:proc
	externdef exitx:proc

	.code
	main proc
		sub rsp,8    ;initiates align 16 ecosystem
		call dumpreg ;verify RSP
		call A_FUNCTION
		add rsp,8
		call exitx
	main endp

	A_FUNCTION proc
		sub rsp,8    ;alignment chaining
		call dumpreg ;verify RSP
		call B_FUNCTION
		add rsp,8
		ret
	A_FUNCTION endp

	B_FUNCTION proc
		sub rsp,8    ;alignment chaining
		call dumpreg ;verify RSP
		add rsp,8
		ret
	B_FUNCTION endp
	end
```

In NASM / FASM syntax

```
	section .text
	;section '.text' code readable executable ;FASM section
main:
	sub    rsp,8        ;initiates align 16 ecosystem
	call   dumpreg      ;verify RSP
	call   A_FUNCTION
	add    rsp,8
	call   exitx

A_FUNCTION:
	sub    rsp,8        ;alignment chaining
	call   dumpreg      ;verify RSP
	call   B_FUNCTION
	add    rsp,8
	ret

B_FUNCTION:
	sub    rsp,8        ;alignment chaining
	call   dumpreg      ;verify RSP
	add    rsp,8
	ret
```

The output of the three `dumpreg`;

```
--snip--
R14|0000000000000000 R15|0000000000000000 RBP|00000000001A1350
RSP|000000000062FE50 RIP|0000000000402CE4 [UHEX]     ;in main

--snip--
R14|0000000000000000 R15|0000000000000000 RBP|00000000001A1350
RSP|000000000062FE40 RIP|0000000000402CFB [UHEX]     ;in A_FUNCTION

--snip--
R14|0000000000000000 R15|0000000000000000 RBP|00000000001A1350
RSP|000000000062FE30 RIP|0000000000402D0E [UHEX]     ;in B_FUNCTION
```

Maintaining The Alignment

A function uses the stack for something else as well, particularly in maintaining local variables or other one-off short live storage. Now that the stack is well-aligned, a function needs to make sure that it will stay that way at all times before calling for example, a WinAPI. Any usage of the stack inside a function body must consider how the value of RSP would end up after such usage. In other words, the WinAPI must see the stack is aligned by the time the call to an API-compliant function is executed.

For that reason, most programmers often use the stack in pairs that translate to multiples of 16 even if they just need a small space. For example;

```
A_FUNCTION proc
       sub    rsp,8         ;chain the stack to align16 ecosystem
       push   rcx           ;oops...
       ;call dumpreg        ;for verification of RSP
       call   aWINAPI       ;Mis-aligned due to push rcx
       pop    rcx
       add    rsp,8
       ret
A_FUNCTION endp

aWinAPI proc
       sub    rsp,8
       call   dumpreg
       add    rsp,8
       ret
aWinAPI endp
```

What happens here is that the programmer wants to save the RCX register on the stack by using `push`. There is nothing wrong with it but by doing so he breaks the alignment prior to making a call to `aWinAPI` that observes aligned stack policy. Uncomment the call to `dumpreg` to see what happens to RSP prior to making the call to `aWinAPI`.

To overcome that problem, a programmer has three practical choices;

i) Complete the `push-pop` pair prior to calling the API

```
push   rcx
...
pop    rcx
call   aWinAPI
```

ii) Make two pushes or multiples of two to preserve the alignment:

```
push   0            ;this doesn't matter
push   rcx          ;this does matter
call   aWinAPI      ;stack alignment is still well-preserved
pop    rcx          ;restore RCX
add    rsp,8        ;restore the stack
```

iii) The step above can also be done by allocating even stack spaces via `sub`, such as;

```
sub    rsp,16       ;Allocate 16 bytes stack space
mov    [rsp],rcx    ;save rcx
call   aWinAPI      ;stack is still well-aligned by this point
mov    rcx,[rsp]    ;restore RCX
add    rsp,16       ;restore stack
```

In a typical program one could see a combination of stack usage as demonstrated above but that should not put you in confusion. Just make sure that the stack is well-aligned to 16-bytes boundary prior to making a call to a 64-bit API. You can use CPU2.0's `dumpreg` to verify the RSP at any point of doubt.

Even-Odd-Even Counting A Win64 program is likely to get complicated particularly when dealing with arguments, parameters and the local stack usage. For example, your code may appear something like this;

```
Change_Font:
       sub    rsp,8        ;align the stack. Or just use a PUSH
       push   rcx          ;but you want to save all these registers
       push   rdx
       push   r11
       sub    rsp,32       ;allocate stack space or other local use
       …
       mov    rcx,something
       mov    rdx,something_else
       call   aWinAPI
       …
       call   bWinAPI
       …
       add    rsp,32       ;restore all stack allocation
       pop    r11
       pop    rdx
       pop    rcx
       add    rsp,8
       ret
```

This is just a typical Win64 code. But how does the multiple `pushes` guarantee stack alignment prior to calling both `aWinAPI` and `bWinAPI`? They do not.

One quick way to confirm the state of the stack alignment at any point is to simply use an **even-odd-even** manual counting to quickly verify the status of the stack at any particular point. This will save you time and easy to implement visually.

An “Even” represents aligned stack, “Odd” represents misaligned stack. Example, taking the above code, you can silently determine the alignment;

```
Change_Font:
       sub    rsp,8         ;even
       push   rcx           ;odd
       push   rdx           ;even
       push   r11           ;odd
       sub    rsp,32        ;for local vars. even, odd, even, ODD (8 x 4)
       …
       mov    rcx,something
       mov    rdx,something_else
       call   aWinAPI
       …
       call   bWinAPI
       …
       add    rsp,32
       pop    r11
       pop    rdx
       pop    rcx
       add    rsp,8
       ret
```

The manual calculation demonstrates that your stack is misaligned due to it ends with an “odd”. So it confirms that your `Change_Font` routine will never be able to call both `aWinAPI` and `bWinAPI` due to it lives outside the aligned ecosystem, regardless of your initial effort to align it via `sub rsp,8`.

As you might have already guessed by now, the practical remedy is to simply increase the size of the local stack allocation so that it ends with an “even”.

```
Change_Font:
       push   rbp           ;even
       push   rcx           ;odd
       push   rdx           ;even
       push   r11           ;odd
       sub    rsp,40        ;even, odd, even, odd, EVEN (8 x 5)
       …
       mov    rcx,something
       mov    rdx,something_else
       call   aWinAPI
       …
       call   bWinAPI
       …
       add    rsp,40        ;Don't forget this
       pop    r11
       pop    rdx
       pop    rcx
       pop    rbp
       ret
```

This explains why many 64-bit codes out there employ various values to be deducted from RSP in order to come up with an aligned stack. It is actually not the values themselves that matter but rather how the RSP register would finally end up in the function body – either **odd** or **even**.

For example, when 32 byte shadow space is allocated (part of Win64 API requirement), you should by now be able to explain why, for example, the value 40 is used to align the stack instead of just 32 for a call to `MessageBoxA,` and some other time it uses different values.

```
myFunction:
       push   rdi           ;even
       push   rsi           ;odd
       sub    rsp,40        ;shadow space (even odd even odd even)

       mov    r9,arg4
       mov    r8,arg3
       mov    rdx,arg2
       mov    rcx,arg1
       call   MessageBoxA

       add    rsp,40
       pop    rsi
       pop    rdi
       ret
```

This simple even-odd calculation works on the assumption that the stack works around its default 8-byte sizes (as in 64-bit architecture) or multiplies of it. If you handle the stack by the size of odd bytes, you're on your own.

Misconceptions Stack alignment is not a CPU requirement. It is a requirement imposed by certain libraries such as WinAPI that define their own calling conventions. Many kernel services do not honor stack alignment but may require shadow space or red zones to be observed. But if you are maintaining SSE / AVX data on the stack, your RSP may need to be aligned anyway to accommodate SSE/AVX **aligned** data transfers and instructions.

Point Alignments Still, chained alignment is not the only way. Some products, for various reasons, prefer a scheme where the alignment is initiated only by the time such requirement exists, regardless of the stack ecosystem. C still uses point alignment in order to maintain command-line arguments via RBP addressing, and to cater to both aligned and non-comformant applications. For example, applying a legacy standard-call stack frame setup;

```
       push   rbp
       mov    rbp,rsp
       sub    rsp,93        ;some weird stuff
       and    rsp,-16       ;create an aligned storage here, on stack
       …
       mov    rsp,rbp
       pop    rbp
       ret
```

One advantage is that this kind of alignment is safe from misalignment issues or threats coming from the others. The disadvantage however is the high overhead in maintaining stack frame setup and the amount of stack programming involved, thus beating the entire purpose of the *fastcall*, which is to minimize the stack movement and programming.

> **Note** Recall that the stack grows downwards. The operation `and,-16` will even down current RSP to the next lower 16-byte boundary, further down the stack memory, achieving the same effect as a `sub` instruction, plus alignment at the same time. Use `stackview` routine from CPU2.0

Leaf Functions As stated previously, stack alignment is required for calling API-compliant routines and modules. If your functions is a leaf, that is it calls no other functions and just simply returns to the caller, then there's no point in aligning the stack. However as stated earlier, if you have SSE/AVX aligned data maintained as local variables, you need to align it anyway.

Suggestions / Download

You can reach me and / or download CPU2.0 via;

1) Twitter @SoffianAbdRasad
2) Gmail: soffianabdulrasad @ gmail . com
3) https://sourceforge.net/projects/baselibs/files/

Test you understanding

Assuming an aligned stack eco-system, find A so that the stack is aligned to 16, and enough room for shadow space (32 bytes), using even-odd-even counting.

FunctionA:
sub rsp,A
….
	FunctionB:
	push rbx
	sub rsp,A
	….
		FunctionC:
		push rbp
		mov rbp,rsp
		sub rsp,512
		and rsp,-16
		fxsave [rsp]
		push rbx
		sub rsp,A
		….

FunctionD:
push rax
...
push 34h
sub rsp,A
call aWinAPI
add rsp,A+
pop rax
ret		

End of document