This work is licensed under a Creative Commons license

Attribution-NonCommercial-NoDerivatives 4.0 International
(CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

x86/x64 ABIs

An overview of Application Binary Interfaces on x86

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
X & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Outline

Data representation

Here:

- word $\rightarrow$ 32-bit object (which makes sense, OTOH it's confusing)
- a null pointer has the value zero

Type	C	sizeof	Alignment (bytes)	Intel386 Architecture
Integral	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

	`int`
	`signed int`
	`long`
	`signed long`
	`enum`
		4	4	signed word

	`unsigned int` `unsigned long`	4	4	unsigned word
Pointer	*any-type* `*` *any-type* `(*)()`	4	4	unsigned word
Floating-point	`float`	4	4	single-precision (IEEE)
	`double`	8	4	double-precision (IEEE)
	`long double`	12	4	extended-precision (IEEE)

Outline

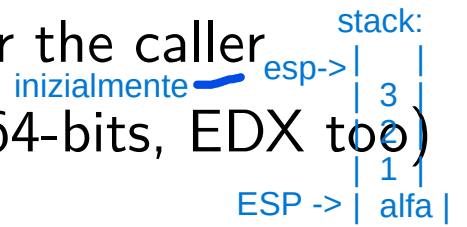
Function calling (32 bits)

ESP= stack pointer
EIP=instruction pointer

- stack kept 16-byte aligned (32/64 in some special cases)
- argument words pushed in reverse order; i.e. C calling convention
  - argument size padded (if necessary) to keep word alignment
- EBP is the optional frame-pointer
- EBP, EBX, EDI, ESI, and ESP must be preserved for the caller
- integral/pointer return values are stored in EAX (if 64-bits, EDX too)
- EBX is the GOT base register (for PIC code only)
- flag *direction* of EFLAGS must be zero on entry and upon exit
- . . .

f(1,2,3) ->

```
push 3
push 2
push 1
call  f
alfa: ADD ESP, 12
```



in caso di local variabile? le metto sotto alfa? cosa che funziona

alfa è l'indirizzo dell'EIP prima di chiamare f

It's useful seeing how C gets compiled

Compiler explorer: `https://gcc.godbolt.org/`

Also (x64 only), "Just Enough Assembly for Compiler Explorer" by A. S. Knatten @ CppCon 2021: `https://youtu.be/_sSFtJwgVYQ`

Standard stack frame

dato che in una funzione possiamo creare variabili locali e chiamare altre funzione tutte cose che possono modificare lo stack (e di conseguenza ESP) per cui si inserisce un riferimento a dove sono i parametri EBP e nella casella di memoria in a cui punta EBP (subito dopo alfa) si inserisce il valore di EBP prima di cambiarlo che quindi sara la funzione chamante

in ogni funzione:

Prologue:
```
PUSH EBP
MOV EBP,ESP
SUB ESP, 8
```

epilogue:
```
MOV ESP,EBP
POP EBP
RET
```

Position	Contents	Frame	
4*n*+8 (%ebp)	argument word *n*	Previous	*High addresses*
	. . .		
8 (%ebp)	argument word 0		
4 (%ebp)	return address	Current	
0 (%ebp)	previous %ebp (optional)		
-4 (%ebp)	unspecified		
	. . .		
0 (%esp)	variable size		*Low addresses*

Let's check sum.c (with/without frame-pointer)

Outline

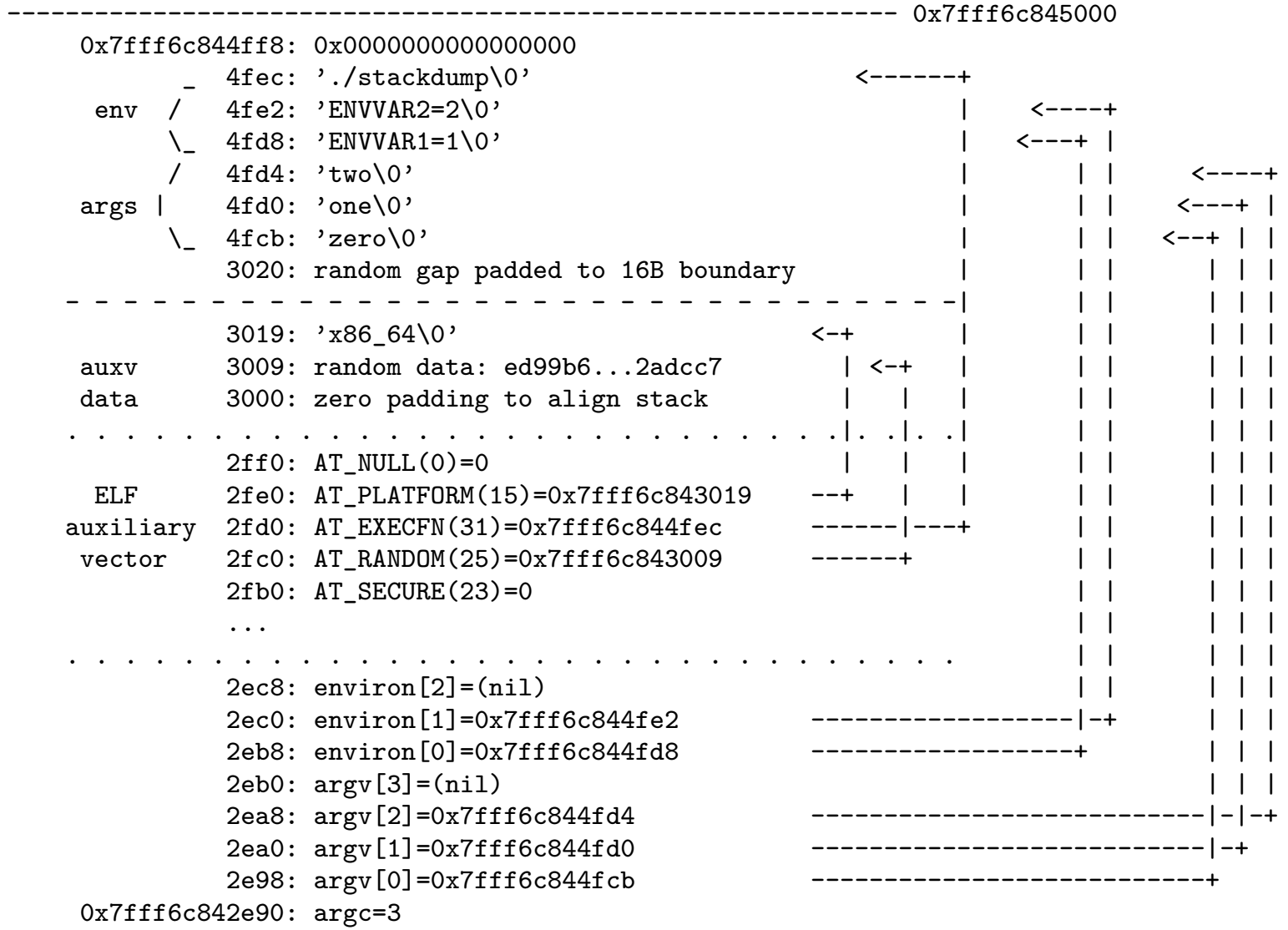
Stack Layout (32 and 64 bits)

entry-point (indirectly) calls `main` passing: argc, argv ($=\alpha$), envp ($=\beta$)

In the stack segment, from higher to lower addresses we have:

- NULL
- program name
- environment strings
- argv strings
- ELF Auxiliary Table
- NULL that ends envp[]
- environment pointers (at address β)
- NULL that ends argv[]
- argv pointers (at address α)
- argc

Stack layout (at the entry-point), an example



Source: `https://lwn.net/Articles/631631/`

Outline

System calls

Normal programs don't need to be concerned: they are just function calls to wrapper functions in C library

> *. . . there are functions in the GNU C Library to do virtually everything that system calls do. These functions work by making system calls themselves.*
> *For example, there is a system call that changes the permissions of a file, but you don't need to know about it because you can just use the GNU C Library's `chmod` function . . .*

`https://www.gnu.org/software/libc/manual/html_node/System-Calls.html`

Indeed, you *cannot* invoke a system call in C (without `asm`)

System call wrappers

- A wrapper function *w* leverages the "magic" of assembly code to
  - put the arguments into CPU registers
  - put the syscall-# into a register
  - "trap" into the kernel
- Kernel executes the syscall handler, that
  - checks the validity of syscall-# and arguments, then
  - calls the actual routine corresponding to that syscall
  - puts the result in a register
  - switches back to user-mode with a special instruction
- *w* checks the result
  - on error sets `errno` and returns an error code (typically: -1)
  - otherwise, return the result

So, syscalls are one/two orders of magnitude *slower* than a function call

Syscall (32 bits)

Parameters are passed by setting:

- `EAX` = syscall #
  - syscall tables (x86 and x64 use different syscall-#): `https://syscalls.w3challs.com/`
- `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` = parameters 1 – 6

and issuing `INT 0x80`

On return,

- `EAX` contains the return value
- all other registers are preserved

Syscall example (32 bit, AT&T syntax)

```
# taken from: https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux
.data
msg: .ascii "Hello World\n"

.text
.global main

main:
    movl $4, %eax   # use the write syscall
    movl $1, %ebx   # write to stdout
    movl $msg, %ecx # use string "Hello World"
    movl $12, %edx  # write 12 characters
    int $0x80       # make syscall

    movl $1, %eax   # use the exit syscall
    movl $0, %ebx   # status code 0
    int $0x80       # make syscall
```

To assemble and link:

```
as --32 -o hello32-att.o hello32-att.s
ld -m elf_i386 -e main ./hello32-att.o -o hello32-att
```

Syscall example (32 bit, Intel syntax)

```
        section .text
        global _start

                                write è il 4
_start:
        mov eax, 4          ; use the write syscall
        mov ebx, 1          ; write to stdout
        mov ecx, msg        ; use string "Hello World"
        mov edx, msglen     ; write 12 characters
        int 0x80            ; make syscall

        mov eax, 1          ; use the exit syscall
        mov ebx, 0          ; status code 0
        int 0x80            ; make syscall

        section .data

msg:    db "Hello World", 10
msglen equ $-msg
```

To assemble and link:

```
nasm -f elf32 ./hello32-intel.asm
ld -m elf_i386 -o hello32-intel hello32-intel.o
```

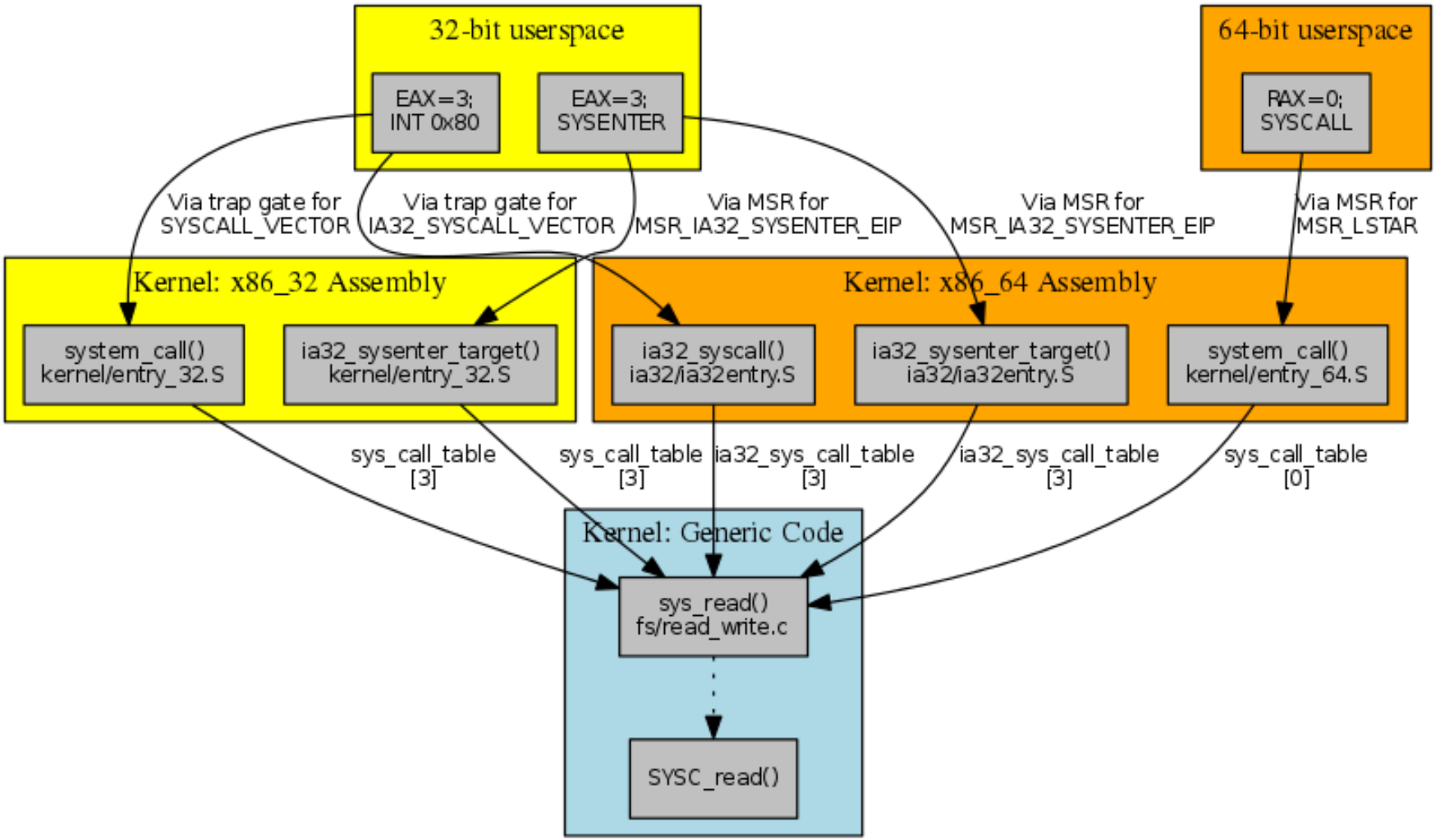
Faster syscalls

- modern CPUs allow faster syscalls via `sysenter` and `sysexit`
  - note: in 64 bit mode there are other instructions: `syscall` and `sysret`
- making sysenter work properly is complicated
  - the kernel takes care of it
  - the bookkeeping to executing a sysenter instruction can change over time (and it has changed)
  - (32 bit) user programs should use a function called `__kernel_vsyscall`, which is implemented in the kernel, but mapped into each user process in vDSO
    - can be found via the ELF auxilliary vector
    - libc puts its value in `gs:[SYSINFO_OFFSET]`
  - vDSO = virtual Dynamic Shared Object, which allow programs to execute kernel code in userland

The Definitive Guide to Linux System Calls

A well-written blog post with many details:
`https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/`

Syscalls



https://lwn.net/Articles/604515/

Data representation ("word" is not used anymore!)

Type	C	sizeof	Alignment (bytes)	AMD64 Architecture
Integral	`_Bool`†	1	1	boolean
	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed twobyte
	`unsigned short`	2	2	unsigned twobyte

	`int`
	`signed int`
	`enum`
		4	4	signed fourbyte

	`unsigned int`	4	4	unsigned fourbyte

	`long`
	`signed long`
	`long long`
	`signed long long`
		8	8	signed eightbyte

	`unsigned long`	8	8	unsigned eightbyte
	`unsigned long long`	8	8	unsigned eightbyte
	`__int128`†† `signed __int128`††	16 16	16 16	signed sixteenbyte signed sixteenbyte
	`unsigned __int128`††	16	16	unsigned sixteenbyte
Pointer	*any-type* `*` *any-type* `(*)()`	8	8	unsigned eightbyte
Floating-point	`float`	4	4	single (IEEE)
	`double`	8	8	double (IEEE)
	`long double`	16	16	80-bit extended (IEEE)
	`__float128`††	16	16	128-bit extended (IEEE)
Packed	`__m64`†† `__m128`††	8 16	8 16	*MMX* and 3DNow! SSE and SSE-2

† This type is called `bool` in C++.

†† These types are optional.

Function calling (64 bits)

in 32 bits si usa solo la stack, perche questo perche la memoria era piu vicina.
Anche se piu vicina è piu lenta dei registri

- arguments (of simple scalar types) are passed
  - first six: using registers: RDI, RSI, RDX, RCX, R8 and R9
  - the rest: using the stack
- oltre i 6 argomenti si passa alla stack, meno efficente
- return value: RAX (and, depending on type, RDX too)
- RBP, RBX, R12-R15 and RSP must be preserved for the caller
  - Note: in 32-bit also EDI and ESI must be preserved
- the end of argument area shall be aligned on a 16 byte boundary
- stack pointer
- the *red-zone*, a 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
  - compilers can optimize leaf-function frames

Check `sum64` out

And add other four arguments to `sum`

Syscall (64 bits)

sempre register quindi massimo 6 parametri

Parameters are passed by setting:

- `RAX` = syscall #
  - syscall tables (x86 and x64 use *different* syscall-#): `https://syscalls.w3challs.com/`
- `RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9` = parameters 1 – 6

and issuing `syscall`

On return,

- `RAX` contains the return value
- all other registers, except `RCX` and `R11` are preserved
  - `RCX` and `R11` are implicitly used by the `syscall` instruction for saving `RIP` and `RFLAGS`

Syscall example (64 bit, AT&T syntax)

```
# taken from https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux
.data
msg: .ascii "Hello World\n"

.text
.global main

main:
    movq $1, %rax   # use the write syscall
    movq $1, %rdi   # write to stdout
    movq $msg, %rsi # use string "Hello World"
    movq $12, %rdx  # write 12 characters
    syscall         # make syscall

    movq $60, %rax  # use the exit syscall
    movq $0, %rdi   # error code 0
    syscall         # make syscall
```

To assemble and link:

```
as -o hello64-att.o hello64-att.s
ld -e main ./hello64-att.o -o hello64-att
```

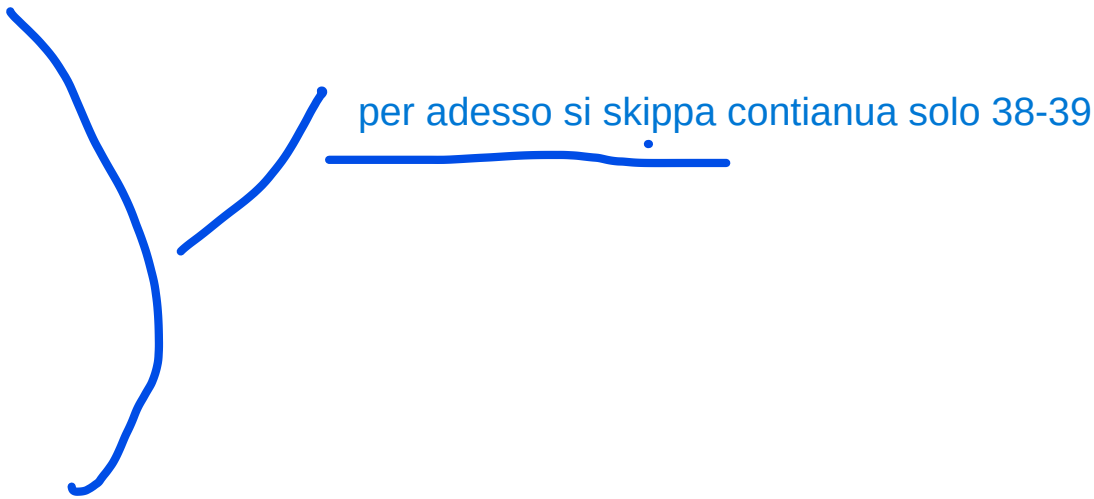
Syscall example (64 bit, Intel syntax)

```
        section .text
        global main

main:
    mov rax, 1      ; use the write syscall
    mov rdi, 1      ; write to stdout
    mov rsi, msg    ; use string "Hello World"
    mov rdx, msglen ; write 12 characters
    syscall         ; make syscall

    mov rax, 60     ; use the exit syscall
    mov rdi, 0      ; error code 0
    syscall         ; make syscall

        section .data
msg:    db "Hello World", 10
msglen equ $-msg
```

To assemble and link:

```
nasm -f elf64 hello64-intel.asm
ld -e main -o hello64-intel hello64-intel.o
```

Outline



FreeBSD x86

By default, the FreeBSD kernel

- uses the C calling convention for arguments
- is accessed using `int 80h`
  - but, it is assumed the program will call a function that issues int 80h, rather than issuing int 80h directly
  - this allows programs written in any language to call the kernel

`https://docs.freebsd.org/en/books/developers-handbook/x86/`

`https://alfonsosiciliano.gitlab.io/posts/2021-01-02-freebsd-system-calls-table.html`

Example: calling open

```
kernel:
        int     80h     ; Call kernel
        ret

open:
        push    dword mode
        push    dword flags
        push    dword path
        mov     eax, 5
        call    kernel
        add     esp, byte 12
        ret
```

that can be optimized to...

Example: calling open (optimized)

```
open:
        push    dword mode
        push    dword flags
        push    dword path
        mov     eax, 5
        push    eax         ; anything, really
        int     80h         ; Call kernel
        add     esp, byte 16
        ret
```

Function calls

There are several calling conventions, the most common are:

- stdcall, the "standard" calling convention, used by most Windows API
- cdecl, the C calling convention

As in Linux x86, both pass the arguments right to left on the stack, the difference lies in how the stack is cleaned up:

`stdcall` the callee is responsible to cleanup the stack
- less code (callers do not need to do anything), but fixed number of arguments

`cdecl` the caller is responsible
- the advantage is the fact that functions can handle a variable number of arguments

Registers `EAX`, `ECX` and `EDX` are caller-saved; (non-floating) return values are stored in `EAX` (and `EDX`, for 64-bit values)

System calls

Wrappers functions are (almost) always used: too unreliable otherwise;
e.g. `NtCreateFile` is

- x86 `0x163` on 8.0, `0x168` on 8.1, `0x016e` on 10.1507, `0x170` on 10.1511, `0x172` on 10.1607, ...
- x64 `0x53` on 8.0, `0x54` on 8.1, `0x55` on 10.1507 to 10.20H2

However, see also:

- `https://github.com/jthuraisamy/SysWhispers2`
- `https://github.com/crummie5/FreshyCalls`
- `https://github.com/am0nsec/HellsGate`
- `https://www.mdsec.co.uk/2022/04/resolving-system-service-numbers-using-the-exception-directory/`

For more details, a nice video is *Intro to Syscalls for Windows Malware*:
`https://www.youtube.com/watch?v=elA_eiqWefw`

Function calls

In x64 there is only one convention:

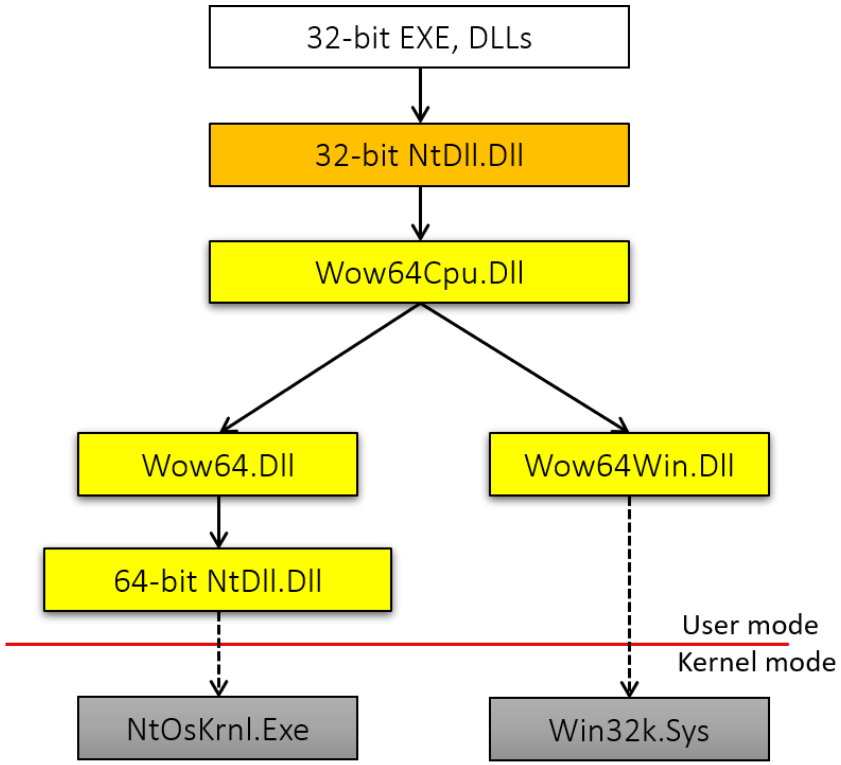
- (non-floating) arguments in the leftmost four positions are passed in left-to-right order in `RCX`, `RDX`, `R8` and `R9`, respectively
- The fifth and higher arguments are passed on the stack

A scalar return value (that fits into 64 bits), is returned through `RAX`

`RAX`, `RCX`, `RDX`, `R8`, `R9`, `R10`, `R11` are caller-saved

WoW64 — DLLs

The 32-bit `ntdll.dll` does NOT invoke system calls but calls into the translation layer that leverages the 64-bit kernel



From `https://leanpub.com/windows10systemprogramming/`

WoW64 — Redirections

- A 32-bit process cannot load a 64-bit DLL and vice versa
  - In theory. . . however, see:
    `https://github.com/JustasMasiulis/wow64pp` and
    `https://github.com/rwfpl/rewolf-wow64ext`
- On x64, 64-bit files are confusingly stored in `C:\Windows\System32`, while 32-bit files are stored in `C:\Windows\SysWow64`
  (in 32-bit processes, `C:\SysNative` gets you to real 64-bit PEs)
- File/registry accesses are automatically redirected; e.g. if a 32-bit process tries to open the former path, it is redirected to the latter

Original Path	Redirected Path for 32-bit x86 Processes
%windir%\System32	%windir%\SysWOW64
%windir%\lastgood\system32	%windir%\lastgood\SysWOW64
%windir%\regedit.exe	%windir%\SysWOW64\regedit.exe

`https://docs.microsoft.com/en-us/windows/win32/winprog64/file-system-redirector`

Outline

Mixing 32/64 bit code

- Each segment can be setup to run code in 32 or 64 bit mode
  - See, e.g., `https://www.malwaretech.com/2014/02/the-0x33-segment-selector-heavens-gate.html`
- Both Linux and Windows set up segment descriptors for
  - 32 bit code → `0x23`
  - 64 bit code → `0x33`
- Inside *any* process you can jump between 32 and 64 modes
- Technique used, for instance,
  - by malware; see, e.g., `https://www.malwaretech.com/2013/06/rise-of-dual-architecture-usermode.html`
  - as a way to escape (badly setup *seccomp*) sandboxes

→ `asm-examples/x86_64_polyglot`

Resources

- Linux system call list:
  `https://syscalls.w3challs.com/`
- Linux User/Kernel ABI: the realities of how C and C++ programs really talk to the OS
  by Greg Law @ ACCU 2018
  `https://www.youtube.com/watch?v=4CdmGxc5BpU`
- "The C++ ABI From the Ground Up"
  by Louis Dionne @ CppCon2019
  `https://youtu.be/DZ93lP1I7wU`