


What are the calling conventions for UNIX & Linux system calls (and user-space functions) on i386 and x86-64

Asked 12 years, 6 months ago Modified 1 year, 10 months ago Viewed 153k times  Part of Intel Collective

Following links explain x86-32 system call conventions for both UNIX (BSD flavor) & Linux:

182

- <http://www.int80h.org/bsdasm/#system-calls>
- <http://www.freebsd.org/doc/en/books/developers-handbook/x86-system-calls.html>

But what are the x86-64 system call conventions on both UNIX & Linux?

linux assembly x86-64 calling-convention abi

Share Improve this question Follow

edited Aug 18, 2020 at 17:40

asked Mar 29, 2010 at 5:48



Peter Cordes

300k 43 545 771



claws


50.5k 57 144 193

There is no "standard" for Unix calling conventions. For linux sure, but I'm sure that Solaris, OpenBSD, Linux and Minix probably have different at least slightly different calling conventions and they are all unix.
– Earlz Mar 29, 2010 at 5:51

2 That's not entirely true - there is a set of UNIX ABIs available for most machine types, which allows C compilers to achieve interoperability. C++ compilers have a bigger problem. – Jonathan Leffler Mar 29, 2010 at 6:54

1 Both of you are correct. I'm looking for FreeBSD & Linux. – claws Mar 29, 2010 at 7:19

I would appreciate if the answer contains information about what registers are preserved accross system calls. Of course the stack pointer is, (unless changed in a controlled way in the __NR_clone call), but are their others? – Albert van der Horst Jan 22, 2016 at 12:28

@AlbertvanderHorst: yes, I just updated the wiki answer with the details for 32bit. 64bit was already accurate: rcx and r11 are destroyed because of the way sysret works, along with rax being replaced with the return value. All other registers are preserved on amd64. – Peter Cordes  Feb 1, 2016 at 23:22

4 Answers

Sorted by:

Highest score (default)

Não encontrou uma resposta? [Pergunte em Stack Overflow em Português.](#)



Further reading for any of the topics here: [The Definitive Guide to Linux System Calls](#)

277 I verified these using GNU Assembler (gas) on Linux.



Kernel Interface



x86-32 aka i386 Linux System Call convention:

In x86-32 parameters for Linux system call are passed using registers. `%eax` for `syscall_number`. `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp` are used for passing 6 parameters to system calls.

The return value is in `%eax`. All other registers (including EFLAGS) are preserved across the `int $0x80`.

I took following snippet from the [Linux Assembly Tutorial](#) but I'm doubtful about this. If any one can show an example, it would be great.

If there are more than six arguments, `%ebx` must contain the memory location where the list of arguments is stored - but don't worry about this because it's unlikely that you'll use a `syscall` with more than six arguments.

For an example and a little more reading, refer to <http://www.int80h.org/bsdasm/#alternate-calling-convention>. Another example of a Hello World for i386 Linux using `int 0x80`: [Hello, world in assembly language with Linux system calls?](#)

There is a faster way to make 32-bit system calls: using `sysenter`. The kernel maps a page of memory into every process (the vDSO), with the user-space side of the `sysenter` dance, which has to cooperate with the kernel for it to be able to find the return address. Arg to register mapping is the same as for `int $0x80`. You should normally call into the vDSO instead of using `sysenter` directly. (See [The Definitive Guide to Linux System Calls](#) for info on linking and calling into the vDSO, and for more info on `sysenter`, and everything else to do with system calls.)

x86-32 [Free|Open|Net|DragonFly]BSD UNIX System Call convention:

Parameters are passed on the stack. Push the parameters (last parameter pushed first) on to the stack. Then push an additional 32-bit of dummy data (It's not actually dummy data. refer to following link for more info) and then give a system call instruction `int $0x80`

<http://www.int80h.org/bsdasm/#default-calling-convention>

x86-64 Linux System Call convention:

(Note: [x86-64 Mac OS X is similar but different](#) from Linux. TODO: check what *BSD does)

Refer to section: "A.2 AMD64 **Linux** Kernel Conventions" of [System V Application Binary Interface AMD64 Architecture Processor Supplement](#). The latest versions of the i386 and x86-64 System V psABIs can be found [linked from this page in the ABI maintainer's repo](#). (See also the [x86](#) tag wiki for up-to-date ABI links and lots of other good stuff about x86 asm.)

Here is the snippet from this section:

1. User-level applications use as integer registers for passing the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. **The kernel interface uses `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`.**
2. A system-call is done via the `syscall` instruction. This [clobbers `%rcx` and `%r11`](#) as well as the `%rax` return value, but other registers are preserved.
3. The number of the syscall has to be passed in register `%rax`.
4. System-calls are limited to six arguments, no argument is passed directly on the stack.
5. Returning from the syscall, register `%rax` contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is `-errno`.
6. Only values of class INTEGER or class MEMORY are passed to the kernel.

Remember this is from the Linux-specific appendix to the ABI, and even for Linux it's informative not normative. (But it is in fact accurate.)

This 32-bit `int $0x80` ABI is usable in 64-bit code (but highly not recommended). [What happens if you use the 32-bit int 0x80 Linux ABI in 64-bit code?](#) It still truncates its inputs to 32-bit, so it's unsuitable for pointers, and it zeros `r8-r11`.

User Interface: function calling

x86-32 Function Calling convention:

In x86-32 parameters were passed on stack. Last parameter was pushed first on to the stack until all parameters are done and then `call` instruction was executed. This is used for calling C library (libc) functions on Linux from assembly.

Modern versions of the i386 System V ABI (used on Linux) require 16-byte alignment of `%esp` before a `call`, like the x86-64 System V ABI has always required. Callees are allowed to assume that and use SSE 16-byte loads/stores that fault on unaligned. But historically, Linux only required 4-byte stack alignment, so it took extra work to reserve naturally-aligned space even for an 8-byte `double` or something.

Some other modern 32-bit systems still don't require more than 4 byte stack alignment.

x86-64 System V user-space Function Calling convention:

x86-64 System V passes args in registers, which is more efficient than i386 System V's stack args convention. It avoids the latency and extra instructions of storing args to memory (cache) and then loading them back again in the callee. This works well because there are more registers available, and is better for modern high-performance CPUs where latency and out-of-order execution matter. (The i386 ABI is very old).

In this *new* mechanism: First the parameters are divided into classes. The class of each parameter determines the manner in which it is passed to the called function.

For complete information refer to : "3.2 Function Calling Sequence" of [System V Application Binary Interface AMD64 Architecture Processor Supplement](#) which reads, in part:

Once arguments are classified, the registers get assigned (in left-to-right order) for passing as follows:

1. If the class is MEMORY, pass the argument on the stack.
2. If the class is INTEGER, the next available register of the sequence %rdi, %rsi, %rdx, %rcx, %r8 and %r9 is used

So %rdi, %rsi, %rdx, %rcx, %r8 and %r9 are the registers *in order* used to pass integer/pointer (i.e. INTEGER class) parameters to any libc function from assembly. %rdi is used for the first INTEGER parameter. %rsi for 2nd, %rdx for 3rd and so on. Then `call` instruction should be given. The stack (%rsp) must be 16B-aligned when `call` executes.

If there are more than 6 INTEGER parameters, the 7th INTEGER parameter and later are passed on the stack. (Caller pops, same as x86-32.)

The first 8 floating point args are passed in %xmm0-7, later on the stack. There are no call-preserved vector registers. (A function with a mix of FP and integer arguments can have more than 8 total register arguments.)

Variadic functions ([like printf](#)) always need `%a1` = the number of FP register args.

There are rules for when to pack structs into registers (`rdx:rax` on return) vs. in memory. See the ABI for details, and check compiler output to make sure your code agrees with compilers about how something should be passed/returned.

Note that [the Windows x64 function calling convention](#) has multiple significant differences from x86-64 System V, like shadow space that *must* be reserved by the caller (instead of a red-zone), and call-preserved xmm6-xmm15. And very different rules for which arg goes in which register.



-
- 1 In linux 32 "all registers except ax bx cd dx si di bp are preserved". I can't think of any ...
– [Albert van der Horst](#) Mar 16, 2016 at 2:41
-
- 1 @Nicolás: caller cleans the stack. I updated the answer with more details about the function-calling convention. – [Peter Cordes](#) 🌟 Sep 3, 2017 at 3:05
-
- 1 If you use Linux's `int 0x80` ABI in 64-bit code, this is exactly what happens:
stackoverflow.com/questions/46087730/... It zeros r8-r11, and works exactly like when run in a 32-bit process. In that Q&A I have an example showing it working, or failing with truncating a pointer. And I also dug into the kernel source to show why it behaves that way. – [Peter Cordes](#) 🌟 Sep 7, 2017 at 4:38
-
- 1 @EvanCarroll : The snippet (quoted text) is at the link given [Linux Assembly Tutorial](#) specifically in section 4.3 *Linux System Calls* – [Michael Petch](#) Oct 26, 2018 at 0:50
-
- 1 @r0ei It's the same as with 64-bit registers. It's ax instead of rax, it's bx instead of rbx and so on. Except if you have a 16-bit calling convention, there are other ways of passing the arguments. – [JCWasmx86](#) Sep 15, 2020 at 11:05
-

Linux kernel 5.0 source comments

- 16 I knew that x86 specifics are under `arch/x86`, and that syscall stuff goes under `arch/x86/entry`. So a quick `git grep rdi` in that directory leads me to [arch/x86/entry/entry_64.S](#):

```
/*
 * 64-bit SYSCALL instruction entry. Up to 6 arguments in registers.
 *
 * This is the only entry point used for 64-bit system calls. The
 * hardware interface is reasonably well designed and the register to
 * argument mapping Linux uses fits well with the registers that are
 * available when SYSCALL is used.
 *
 * SYSCALL instructions can be found inlined in libc implementations as
 * well as some other programs and libraries. There are also a handful
 * of SYSCALL instructions in the vDSO used, for example, as a
 * clock_gettimeofday fallback.
 *
 * 64-bit SYSCALL saves rip to rcx, clears rflags.RF, then saves rflags to r11,
 * then loads new ss, cs, and rip from previously programmed MSRs.
 * rflags gets masked by a value from another MSR (so CLD and CLAC
 * are not needed). SYSCALL does not save anything on the stack
 * and does not change rsp.
 *
 * Registers on entry:
 * rax  system call number
 * rcx  return address
 * r11  saved rflags (note: r11 is callee-clobbered register in C ABI)
 * rdi  arg0
 * rsi  arg1
 * rdx  arg2
 * r10  arg3 (needs to be moved to rcx to conform to C ABI)
 * r8   arg4
 * r9   arg5
```

```

* (note: r12-r15, rbp, rbx are callee-preserved in C ABI)
*
* Only called from user space.
*
* When user can change pt_regs->foo always force IRET. That is because
* it deals with uncanonical addresses better. SYSRET has trouble
* with them due to bugs in both AMD and Intel CPUs.
*/

```

and for 32-bit at [arch/x86/entry/entry_32.S](#):

```

/*
 * 32-bit SYSENTER entry.
 *
 * 32-bit system calls through the vDSO's __kernel_vsyscall enter here
 * if X86_FEATURE_SEP is available. This is the preferred system call
 * entry on 32-bit systems.
 *
 * The SYSENTER instruction, in principle, should *only* occur in the
 * vDSO. In practice, a small number of Android devices were shipped
 * with a copy of Bionic that inlined a SYSENTER instruction. This
 * never happened in any of Google's Bionic versions -- it only happened
 * in a narrow range of Intel-provided versions.
 *
 * SYSENTER loads SS, ESP, CS, and EIP from previously programmed MSRs.
 * IF and VM in RFLAGS are cleared (IOW: interrupts are off).
 * SYSENTER does not save anything on the stack,
 * and does not save old EIP (!!!), ESP, or EFLAGS.
 *
 * To avoid losing track of EFLAGS.VM (and thus potentially corrupting
 * user and/or vm86 state), we explicitly disable the SYSENTER
 * instruction in vm86 mode by reprogramming the MSRs.
 *
 * Arguments:
 * eax  system call number
 * ebx  arg1
 * ecx  arg2
 * edx  arg3
 * esi  arg4
 * edi  arg5
 * ebp  user stack
 * 0(%ebp) arg6
 */

```

glibc 2.29 Linux x86_64 system call implementation

Now let's cheat by looking at a major libc implementations and see what they are doing.

What could be better than looking into glibc that I'm using right now as I write this answer? :-)

glibc 2.29 defines x86_64 syscalls at [sysdeps/unix/sysv/linux/x86_64/sysdep.h](#) and that contains some interesting code, e.g.:

```

/* The Linux/x86-64 kernel expects the system call parameters in
   registers according to the following table:

   syscall number  rax

```

```

arg 1    rdi
arg 2    rsi
arg 3    rdx
arg 4    r10
arg 5    r8
arg 6    r9

```

The Linux kernel uses and destroys internally these registers:

```

return address from
syscall    rcx
eflags from syscall r11

```

Normal function call, including calls to the system call stub functions in the libc, get the first six parameters passed in registers and the seventh parameter and later on the stack. The register use is as follows:

system call number in the DO_CALL macro

```

arg 1    rdi
arg 2    rsi
arg 3    rdx
arg 4    rcx
arg 5    r8
arg 6    r9

```

We have to take care that the stack is aligned to 16 bytes. When called the stack is not aligned since the return address has just been pushed.

Syscalls of more than 6 arguments are not supported. */

and:

```

/* Registers clobbered by syscall. */
#define REGISTERS_CLOBBERED_BY_SYSCALL "cc", "r11", "cx"

#undef internal_syscall6
#define internal_syscall6(number, err, arg1, arg2, arg3, arg4, arg5, arg6) \
({ \
    unsigned long int resultvar; \
    TYPEFY (arg6, __arg6) = ARGIFY (arg6); \
    TYPEFY (arg5, __arg5) = ARGIFY (arg5); \
    TYPEFY (arg4, __arg4) = ARGIFY (arg4); \
    TYPEFY (arg3, __arg3) = ARGIFY (arg3); \
    TYPEFY (arg2, __arg2) = ARGIFY (arg2); \
    TYPEFY (arg1, __arg1) = ARGIFY (arg1); \
    register TYPEFY (arg6, _a6) asm ("r9") = __arg6; \
    register TYPEFY (arg5, _a5) asm ("r8") = __arg5; \
    register TYPEFY (arg4, _a4) asm ("r10") = __arg4; \
    register TYPEFY (arg3, _a3) asm ("rdx") = __arg3; \
    register TYPEFY (arg2, _a2) asm ("rsi") = __arg2; \
    register TYPEFY (arg1, _a1) asm ("rdi") = __arg1; \
    asm volatile ( \
        "syscall\n\t" \
        : "=a" (resultvar) \
        : "0" (number), "r" (_a1), "r" (_a2), "r" (_a3), "r" (_a4), \
          "r" (_a5), "r" (_a6) \
        : "memory", REGISTERS_CLOBBERED_BY_SYSCALL); \
})

```

10/11/22, 3:22 PM assembly - What are the calling conventions for UNIX & Linux system calls (and user-space functions) on i386 a...
(long int) resultvar; \
}))

which I feel are pretty self explanatory. Note how this seems to have been designed to exactly match the calling convention of regular System V AMD64 ABI functions:

https://en.wikipedia.org/wiki/X86_calling_conventions#List_of_x86_calling_conventions

Quick reminder of the clobbers:

- cc means flag registers. But [Peter Cordes comments](#) that this is unnecessary here.
- memory means that a pointer may be passed in assembly and used to access memory

For an explicit minimal runnable example from scratch see this answer: [How to invoke a system call via syscall or sysenter in inline assembly?](#)

Make some syscalls in assembly manually

Not very scientific, but fun:

- x86_64.S

```
.text
.global _start
_start:
asm_main_after_prologue:
    /* write */
    mov $1, %rax    /* syscall number */
    mov $1, %rdi    /* stdout */
    mov $msg, %rsi   /* buffer */
    mov $len, %rdx   /* len */
    syscall

    /* exit */
    mov $60, %rax    /* syscall number */
    mov $0, %rdi     /* exit status */
    syscall
msg:
    .ascii "hello\n"
len = . - msg
```

[GitHub upstream](#).

Make system calls from C

Here's an example with register constraints: [How to invoke a system call via syscall or sysenter in inline assembly?](#)

aarch64

I've shown a minimal runnable userland example at:

<https://reverseengineering.stackexchange.com/questions/16917/arm64-syscalls->

[table/18834#18834](#) TODO grep kernel code here, should be easy.

Share Improve this answer Follow



edited Dec 7, 2020 at 9:09

answered Mar 2, 2019 at 9:34



Ciro Santilli
OurBigBook.com

317k 89 1129 911

- 1 The "cc" clobber is unnecessary: Linux syscalls save/restore RFLAGS (The `syscall / sysret` instructions do that using R11, and the kernel doesn't modify the saved R11 / RFLAGS other than via `ptrace` debugger system calls.) Not that it ever matters, because a "cc" clobber is implicit for x86 / x86-64 in GNU C Extended asm, so you can't gain anything by leaving it out. – Peter Cordes  Mar 2, 2019 at 18:21 

Perhaps you're looking for the x86_64 ABI?

15

- www.x86-64.org/documentation/abi.pdf (404 at 2018-11-24)
- www.x86-64.org/documentation/abi.pdf (via Wayback Machine at 2018-11-24)
- [Where is the x86-64 System V ABI documented? - https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI](https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI) is kept up to date (by HJ Lu, one of the ABI maintainers) with links to PDFs of the official current version.

If that's not precisely what you're after, use 'x86_64 abi' in your preferred search engine to find alternative references.

Share Improve this answer Follow

edited Nov 25, 2018 at 18:48

answered Mar 29, 2010 at 5:56



Peter Cordes 

300k 43 545 771




Jonathan Leffler

709k 135 878 1239

- 5 actually, I'm only want the System Call convention. esp for UNIX (FreeBSD) – claws Mar 29, 2010 at 8:25

- 3 @claws: the system call convention is one part of the ABI. – Jonathan Leffler Mar 29, 2010 at 13:23

- 1 yeah. I've gone to each individual OS's kernel development irc and asked them about it. They've told me to look into the source and figure out. I don't understand without documenting stuff how can they just start developing? So, I've added an answer from info I collected, hoping for others to fill in the rest of the details. – claws Mar 29, 2010 at 13:43 

@JonathanLeffler the link seems to be not working right now. If you are also getting an issue visiting the link, can you please update it? – Ajay Brahmakshatriya Nov 25, 2018 at 6:46

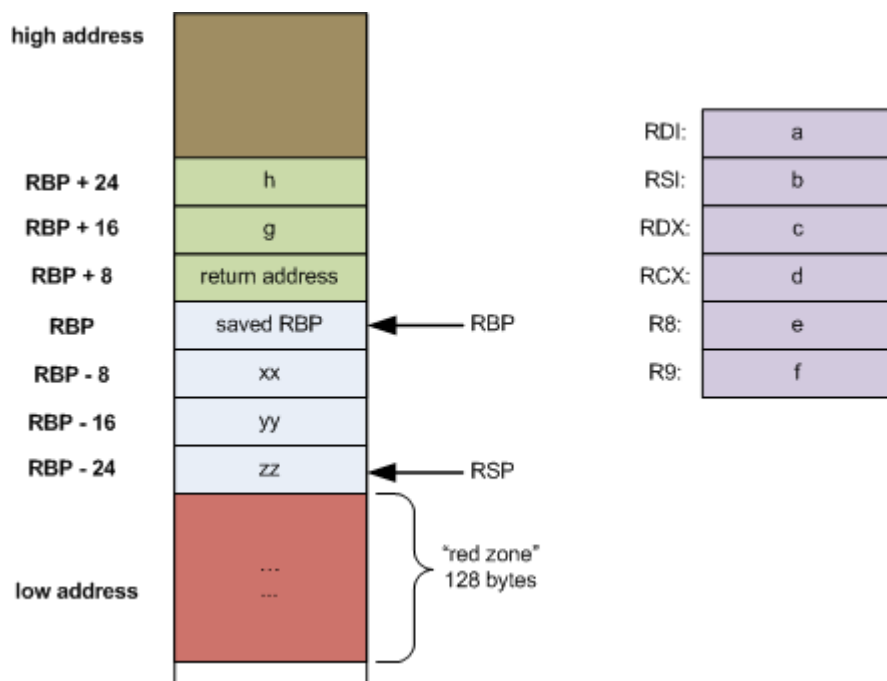
@AjayBrahmakshatriya: Thanks for the heads up; I've added a link to the Wayback Machine record. The whole x86-64.org web site didn't respond with any data. – Jonathan Leffler Nov 25, 2018 at 7:56

Calling conventions defines how parameters are passed in the registers when calling or being called by other program. And the best source of these convention is in the form of ABI standards

12

defined for each these hardware. For ease of compilation, the same ABI is also used by userspace and kernel program. Linux/Freebsd follow the same ABI for x86-64 and another set for 32-bit. But x86-64 ABI for Windows is different from Linux/FreeBSD. And generally ABI does not differentiate system call vs normal "functions calls". Ie, here is a particular example of x86_64 calling conventions and it is the same for both Linux userspace and kernel:

<http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/> (note the sequence a,b,c,d,e,f of parameters):



Performance is one of the reasons for these ABI (eg, passing parameters via registers instead of saving into memory stacks)

For ARM there is various ABI:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html>

<https://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf>

ARM64 convention:

http://infocenter.arm.com/help/topic/com.arm.doc.ih0055b/IHI0055B_aapcs64.pdf

For Linux on PowerPC:

http://refspecs.freestdards.org/elf/elfspec_ppc.pdf

<http://www.0x04.net/doc/elf/psABI-ppc64.pdf>

And for embedded there is the PPC EABI:

http://www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf

This document is good overview of all the different conventions:

http://www.agner.org/optimize/calling_conventions.pdf

Share Improve this answer Follow

edited Jan 23, 2016 at 12:26

answered Jun 12, 2011 at 2:03



Peter Teoh

6,058 4 41 58

Totally besides the point. The poster of the question would not ask for the 64 bit syscall calling convention in linux if it were the same than the general ABI conversions. – [Albert van der Horst](#) Jan 22, 2016 at 13:04
