Syscalls

Syscalls are the interface between user programs and the Linux kernel. They are used to let the kernel perform various system tasks, such as file access, process management and networking. In the C programming language, you would normally call a wrapper function which executes all required steps or even use high-level features such as the standard IO library.

On Linux, there are several ways to make a syscall. This page will focus on making syscalls by calling a software interrupt using `int $0x80` or `syscall`. This is an easy and intuitive method of making syscalls in assembly-only programs.

Making a syscall

For making a syscall using an interrupt, you have to pass all required information to the kernel by copying them into general purpose registers.

Each syscall has a fixed number (note: the numbers differ between `int $0x80` and `syscall`!). You specify the syscall by writing the number into the `eax` / `rax` register.

Most syscalls take parameters to perform their task. Those parameters are passed by writing them in the appropriate registers before making the actual call. Each parameter index has a specific register. See the tables in the subsections as the mapping differs between `int $0x80` and `syscall`. Parameters are passed in the order they appear in the function signature of the corresponding C wrapper function. You may find syscall functions and their signatures in every Linux API documentation, like the reference manual (type `man 2 open` to see the signature of the `open` syscall).

After everything is set up correctly, you call the interrupt using `int $0x80` or `syscall` and the kernel performs the task.

The return / error value of a syscall is written to `eax` / `rax`.

The kernel uses its own stack to perform the actions. The user stack is not touched in any way.

int 0x80

On both Linux x86 and Linux x86_64 systems you can make a syscall by calling interrupt 0x80 using the `int $0x80` command. Parameters are passed by setting the general purpose registers as following:

Syscall #	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6
eax	ebx	ecx	edx	esi	edi	ebp

Return value
eax

The syscall numbers are described in the Linux generated file `$build/arch/x86/include/generated/uapi/asm/unistd_32.h` or `$build/usr/include/asm/unistd_32.h`. The latter could also be present on your Linux system, just omit the `$build`.

All registers are preserved during the syscall.

syscall

The x86_64 architecture introduced a dedicated instruction to make a syscall. It does not access the interrupt descriptor table and is faster. Parameters are passed by setting the general purpose registers as following:

Syscall #	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6
rax	rdi	rsi	rdx	r10	r8	r9

Return value
rax

The syscall numbers are described in the Linux generated file `$build/usr/include/asm/unistd_64.h`. This file could also be present on your Linux system, just omit the `$build`.

All registers, except `rcx` and `r11` (and the return value, `rax`), are preserved during the syscall.

library call

In call of Linux's library functions parameter 4 is passed on RCX and further parameters, onto the stack.

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

Examples

To summarize and clarify the information, let's have a look at a very simple example: the hello world program. It will write the text "Hello World" to stdout using the `write` syscall and quit the program using the `_exit` syscall.

Syscall signatures:

```
ssize_t write(int fd, const void *buf, size_t count);
void _exit(int status);
```

This is the C program which is implemented in assembly below:

```
#include <unistd.h>

int main(int argc, char *argv[])
{
    write(1, "Hello World\n", 12); /* write "Hello World" to stdout */
    _exit(0);                      /* exit with error code 0 (no error) */
}
```

Both examples start alike: a string stored in the data segment and `_start` as a global symbol.

```
.data
msg: .ascii "Hello World\n"

.text
.global _start
```

int 0x80

As defined in `$build/usr/include/asm/unistd_32.h`, the syscall numbers for `write` and `_exit` are:

```
#define __NR_exit 1
#define __NR_write 4
```

The parameters are passed exactly as one would in a C program, using the correct registers. After everything is set up, the syscall is made using `int $0x80`.

```
_start:
    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   ; error code 0
    int $0x80       ; make syscall
```

syscall

In `$build/usr/include/asm/unistd_64.h`, the syscall numbers are defined as following:

```
#define __NR_write 1
#define __NR_exit 60
```

Parameters are passed just like in the `int $0x80` example, except that the order of the registers is different. The syscall is made using `syscall`.

```
_start:
    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
```

library call

Here is the C Prototype of an example library function.

```
Window XCreateWindow(display, parent, x, y, width, height, border_width, depth,
                     class, visual, valuemask, attributes)
```

Parameters are passed just like in the `int $0x80` example, except that the order of the registers is different.

Library function is declared at the beginning of the source file (and the path to the library, at compilation-linking time).

```
extern XCreateWindow
```

```
        mov rdi, [xserver_pdisplay]
        mov rsi, [xwin_parent]
        mov rdx, [xwin_x]
        mov rcx, [xwin_y]
        mov r8, [xwin_width]
        mov r9, [xwin_height]
        mov rax, attributes
        push rax                ; ARG 12
        sub rax, rax
        mov eax, [xwin_valuemask]
        push rax                ; ARG 11
        mov rax, [xwin_visual]
        push rax                ; ARG 10
        mov rax, [xwin_class]
        push rax                ; ARG 9
        mov rax, [xwin_depth]
        push rax                ; ARG 8
        mov rax, [xwin_border_width]
        push rax                ; ARG 7
        call XCreateWindow
        mov [xwin_window], rax
```

Note the last parameters of function, pushed into the stack, is done in reverse order.

Last edited 8 months ago by an anonymous user