4. System Calls

4.1 System call numbers

System calls are identified by their numbers. The number of the call `foo` is `__NR_foo`. For example, the number of `_llseek` used above is `__NR__llseek`, defined as 140 in `/usr/include/asm-i386/unistd.h`. Different architectures have different numbers.

Often, the kernel routine that handles the call `foo` is called `sys_foo`. One finds the association between numbers and names in the `sys_call_table`, for example in `arch/i386/kernel/entry.S`.

Change

The world changes and system calls change. Since one must not break old binaries, the semantics associated to any given system call number must remain fully backwards compatible.

What happens in practice is one of two things: either one gets a new and improved system call with a new name and number, and the libc routine that used to invoke the old call is changed to use the new one, or the new call (with new number) gets the old name, and the old call gets "old" prefixed to its name.

For example, long ago user IDs had 16 bits, today they have 32. `__NR_getuid` is 24, and `__NR_getuid32` is 199, and the former belongs to the 16-bit version of the call, the latter to the 32-bit version. Looking at the associated kernel routines, we find that these are `sys_getuid16` and `sys_getuid`, respectively. (Thus, `sys_getuid` does not have number `__NR_getuid`.) Looking at glibc, we find code somewhat like

```
int getuid32_available = UNKNOWN;

uid_t getuid(void) {
        if (getuid32_available == TRUE)
                return INLINE_SYSCALL(getuid32, 0);
        if (getuid32_available == UNKNOWN) {
                uid_t res = INLINE_SYSCALL(getuid32, 0);

                if (res == 0 || errno != ENOSYS) {
                        getuid32_available = TRUE;
                        return res;
                }
                getuid32_available = FALSE;
        }
        return INLINE_SYSCALL(getuid, 0);
}
```

For an example where the name was moved and the old call got a name prefixed by "old", see `__NR_oldolduname`, `__NR_olduname`, `__NR_uname`, belonging to `sys_olduname`, `sys_uname`, `sys_newuname`, respectively. One also has `__NR_oldstat`, `__NR_stat`, `__NR_stat64` belonging to `sys_stat`, `sys_newstat`, `sys_stat64`, respectively. And `__NR_umount`, `__NR_umount2` belonging to `sys_oldumount`, `sys_umount`, respectively. And

`__NR_select`, `__NR__newselect` belonging to `old_select`, `sys_select`, respectively.

These moving names are confusing - now you have been warned: the system call with number `__NR_foo` does not always belong to the kernel routine `sys_foo()`.

4.2 The call

What happens? The assembler for a call with 0 parameters (on i386) is

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \
__syscall_return(type,__res); \
}
```

Thus, the basic ingredient is the assembler instruction INT 0x80. This causes a programmed exception and calls the kernel `system_call` routine. Some relevant code fragments:

```
/* include/asm-i386/hw_irq.h */
#define SYSCALL_VECTOR              0x80

/* arch/i386/kernel/traps.c */
        set_system_gate(SYSCALL_VECTOR,&system_call);

/* arch/i386/kernel/entry.S */
#define GET_CURRENT(reg) \
        movl $-8192, reg; \
        andl %esp, reg

#define SAVE_ALL \
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $(__KERNEL_DS),%edx; \
        movl %edx,%ds; \
        movl %edx,%es;

#define RESTORE_ALL     \
        popl %ebx;      \
        popl %ecx;      \
        popl %edx;      \
        popl %esi;      \
        popl %edi;      \
        popl %ebp;      \
        popl %eax;      \
1:      popl %ds;       \
2:      popl %es;       \
        addl $4,%esp;   \
3:      iret;

ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value

ENTRY(ret_from_sys_call)
        cli                             # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
        RESTORE_ALL
```

We transfer execution to `system_call`, save the original value of the EAX register (it is the number of the system call), save all other registers, verify that we are not being traced (otherwise the tracer must be informed and entirely different things happen), make sure that the system call number is within range, and call the appropriate kernel routine from the table `sys_call_table`. Upon return we check a few things and when all is well restore the registers and call IRET to return from this INT.

(This was for the i386 architecture. All details differ on other architectures, but the basic idea is the same: store the syscall number and the syscall parameters somewhere the kernel can find them, in registers, on the stack, or in a known place of memory, do something that causes a transfer to kernel code, etc.)

4.3 System call parameters

On i386, the parameters of a system call are transported via registers. The system call number goes into `%eax`, the first parameter in `%ebx`, the second in `%ecx`, the third in `%edx`, the fourth in `%esi`, the fifth in `%edi`, the sixth in `%ebp`.

Ancient history

Earlier versions of Linux could handle only four or five system call parameters, and therefore the system calls `select()` (5 parameters) and `mmap()` (6 parameters) used to have a single parameter that was a pointer to a parameter block in memory. Since Linux 1.3.0 five parameters are supported (and the earlier `select` with memory block was renamed `old_select`), and since Linux 2.3.31 six parameters are supported (and the earlier `mmap` with memory block was succeeded by the new `mmap2`).

4.4 Error return

Above we said: typically, the kernel returns a negative value to indicate an error. But this would mean that any system call only can return positive values. Since the negative error returns are of the form **`-ESOMETHING`**, and the error numbers have small positive values, there is only a small negative error range. Thus

```
#define __syscall_return(type, res) \
do { \
        if ((unsigned long)(res) >= (unsigned long)(-125)) { \
                errno = -(res); \
                res = -1; \
        } \
        return (type) (res); \
} while (0)
```

Here the range [-125,-1] is reserved for errors (the constant 125 is version and architecture dependent) and other values are OK.

What if a system call wants to return a small negative number and it is not an error? The scheduling priority of a process is set by **`setpriority()`** and read by **`getpriority()`**, and this value ranges from -20 (top priority) to 19 (lowest priority background job). The library routines with these names use these numbers, but the system call **`getpriority()`** returns 20 - P instead of P, moving the output interval to positive numbers only.

Or, similarly, the subfunctions PEEK* of **`ptrace`** return the contents of a memory word in the traced process, and any value is possible. However, the system call returns this value in the **`data`** argument, and glibc does something like

```
        res = sys_ptrace(request, pid, addr, &data);
        if (res >= 0) {
                errno = 0;
                res = data;
        }
        return res;
```

so that a user program has to do

```
        errno = 0;
        res = ptrace(PTRACE_PEEKDATA, pid, addr, NULL);
        if (res == -1 && errno != 0)
                /* error */
```

4.5 Interrupted system calls

Above we saw in **`ret_from_sys_call`** the test on **`sigpending`**: if a signal arrived while we were executing kernel code, then just before returning from the system call we first call the user program's signal handler, and when this finishes return from the system call.

When a system call is slow and a signal arrives while it was blocked, waiting for something, the call is aborted and returns **`-EINTR`**, so that the library function will return -1 and set **`errno`** to **`EINTR`**. Just before the system call returns, the user program's signal handler is called.

(So, what is "slow"? Mostly those calls that can block forever waiting for external events; read and write to terminal devices, but not read and write to disk devices, **`wait`**, **`pause`**.)

This means that a system call can return an error while nothing was wrong. Usually one will want to redo the system call. That can be automated by installing the signal handler using a call to **`sigaction`** with the **`SA_RESTART`** flag set. The effect is that upon an interrupt the system call is aborted, the user program's signal handler is called, and afterwards the system call is restarted from the beginning.

Why is this not the default? It was, for a while, but often it is necessary to react to a signal while the reacting is not done by the signal handler itself. It is difficult to do nontrivial things in a signal handler since the rest of the program is in an unknown state, and most signal handlers just set a flag that is tested elsewhere.

A demo:

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int got_interrupt;

void intrup(int dummy) {
        got_interrupt = 1;
}

void die(char *s) {
        printf("%s\n", s);
        exit(1);
}

int main() {
        struct sigaction sa;
        int n;
        char c;

        sa.sa_handler = intrup;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        if (sigaction(SIGINT, &sa, NULL))
                die("sigaction-SIGINT");
        sa.sa_flags = SA_RESTART;
        if (sigaction(SIGQUIT, &sa, NULL))
                die("sigaction-SIGQUIT");

        got_interrupt = 0;
        n = read(0, &c, 1);
        if (n == -1 && errno == EINTR)
                printf("read call was interrupted\n");
        else if (got_interrupt)
                printf("read call was restarted\n");

        return 0;
}
```

Here Ctrl-C will interrupt the read call, while after Ctrl-\ the read call is restarted.

Partial success

There are other cases where a syscall has to be done in several steps. Instead of just calling the system call **`write()`** it may be necessary to do

```
ssize_t my_write(int fd, const void *buf, size_t count) {
        ssize_t res;

        while (count) {
                res = write(fd, buf, count);
                if (res < 0)
                        return res;
                buf += res;
                count -= res;
        }
        return 0;
}
```

even when writing to an ordinary disk file. Indeed, since 2.6.16 there is a limit MAX_RW_COUNT in **`read_write.c`** that causes a maximal write size of **`INT_MAX & PAGE_CACHE_MASK`** which may be 2^31-1-4095 = 2147479552. This might violate POSIX. Usually a write only returns a short count when interrupted by a signal, or when the disk is full, or the max file size is reached.

4.6 Sysenter and the vsyscall page

It has been observed that a 2 GHz Pentium 4 was much slower than an 850 MHz Pentium III on certain tasks, and that this slowness is caused by the very large overhead of the traditional **`int 0x80`** interrupt on a Pentium 4.

Some models of the i386 family do have faster ways to enter the kernel. On Pentium II there is the **`sysenter`** instruction. Also AMD has a **`syscall`** instruction. It would be good if these could be used.

Something else is that in some applications **`gettimeofday()`** is a done very often, for example for timestamping all transactions. It would be nice if it could be implemented with very low overhead.

One way of obtaining a fast **`gettimeofday()`** is by writing the current time in a fixed place, on a page mapped into the memory of all applications, and updating this location on each clock interrupt. These applications could then read this fixed location with a single instruction - no system call required.

There might be other data that the kernel could make available in a read-only way to the process, like perhaps the current process ID. A *vsyscall* is a "system" call that avoids crossing the userspace-kernel boundary.

Linux is in the process of implementing such ideas. Since Linux 2.5.53 there is a fixed page, called the vsyscall page, filled by the kernel. At kernel initialization time the routine **`sysenter_setup()`** is called. It sets up a non-writable page and writes code for the **`sysenter`** instruction if the CPU supports that, and for the classical **`int 0x80`** otherwise. Thus, the C library can use the fastest type of system call by jumping to a fixed address in the vsyscall page.

This page was changed to have the structure of an ELF binary (called **`linux-vsyscall.so.1`**) in Linux 2.5.69. In Linux 2.5.74 the name was changed to **`linux-gate.so.1`**.

Concerning **`gettimeofday()`**, a vsyscall version for the x86-64 is already part of the vanilla kernel. Patches for i386 exist. (An example of the kind of timing differences: John Stultz reports on an experiment where he measures **`gettimeofday()`** and finds 1.67 us for the **`int 0x80`** way, 1.24 us for the **`sysenter`** way, and 0.88 us for the vsyscall.)

Some details

The kernel maps a page (**`0xffffe000-0xffffefff`**) in the memory of every process. (This is the next-to-last addressable page. The last is not mapped - maybe to avoid bugs related to wraparound.) We can read it:

```
/* get vsyscall page */
#include <unistd.h>
#include <string.h>

int main() {
        char *p = (char *) 0xffffe000;
        char buf[4096];
#if 0
        write(1, p, 4096);
        /* this gives EFAULT */
#else
        memcpy(buf, p, 4096);
        write(1, buf, 4096);
#endif
        return 0;
}
```

and if we do, find an ELF binary.

```
% ./get_vsyscall_page > syspage
% file syspage
syspage: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), stripped
% objdump -h syspage

syspage:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .hash         00000050  ffffe094  ffffe094  00000094  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .dynsym       000000f0  ffffe0e4  ffffe0e4  000000e4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .dynstr       00000056  ffffe1d4  ffffe1d4  000001d4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .gnu.version  0000001e  ffffe22a  ffffe22a  0000022a  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.version_d 00000038  ffffe248  ffffe248  00000248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .text         00000047  ffffe400  ffffe400  00000400  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .eh_frame_hdr 00000024  ffffe448  ffffe448  00000448  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .eh_frame     0000010c  ffffe46c  ffffe46c  0000046c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .dynamic      00000078  ffffe578  ffffe578  00000578  2**2
```

```
                  CONTENTS, ALLOC, LOAD, DATA
  9 .useless      0000000c  ffffe5f0  ffffe5f0  000005f0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
% objdump -d syspage

syspage:     file format elf32-i386

Disassembly of section .text:

ffffe400 <.text>:
ffffe400:       51                      push   %ecx
ffffe401:       52                      push   %edx
ffffe402:       55                      push   %ebp
ffffe403:       89 e5                   mov    %esp,%ebp
ffffe405:       0f 34                   sysenter
ffffe407:       90                      nop
ffffe408:       90                      nop
        ... more nops ...
ffffe40d:       90                      nop
ffffe40e:       eb f3                   jmp    0xffffe403
ffffe410:       5d                      pop    %ebp
ffffe411:       5a                      pop    %edx
ffffe412:       59                      pop    %ecx
ffffe413:       c3                      ret
        ... zero bytes ...
ffffe420:       58                      pop    %eax
ffffe421:       b8 77 00 00 00          mov    $0x77,%eax
ffffe426:       cd 80                   int    $0x80
ffffe428:       90                      nop
ffffe429:       90                      nop
        ... more nops ...
ffffe43f:       90                      nop
ffffe440:       b8 ad 00 00 00          mov    $0xad,%eax
ffffe445:       cd 80                   int    $0x80
```

The interesting addresses here are found via

```
% grep ffffe System.map
ffffe000 A VSYSCALL_BASE
ffffe400 A __kernel_vsyscall
ffffe410 A SYSENTER_RETURN
ffffe420 A __kernel_sigreturn
ffffe440 A __kernel_rt_sigreturn
%
```

So `__kernel_vsyscall` pushes a few registers and does a `sysenter` instruction. And `SYSENTER_RETURN` pops the registers again and returns. And `__kernel_sigreturn` and `__kernel_rt_sigreturn` do system calls 119 and 173, that is, sigreturn and rt_sigreturn, respectively.

What about the jump just before `SYSENTER_RETURN`? It is a trick to handle restarting of system calls with 6 parameters. As Linus said: I'm a disgusting pig, and proud of it to boot.

The code involved is most easily seen from a slightly earlier patch.

A tiny demo program.

```
#include <stdio.h>

int pid;

int main() {
        __asm__(
                "movl $20, %eax    \n"
                "call 0xffffe400   \n"
                "movl %eax, pid    \n"
        );
        printf("pid is %d\n", pid);
        return 0;
}
```

This does the `getpid()` system call (`__NR_getpid` is 20) using `call 0xffffe400` instead of `int 0x80`.

Address space randomization

The layout of the vsyscall page changes, and the entry point varies. It can be found by inspection of the ELF headers of the page.

Since Linux 2.6.18 the page itself is mapped at a random address. The right entry point can now be found by searching the ELF auxiliary vector.

```
/* get vsyscall address and test - compile with -m32 on x86_64 */
#include <stdio.h>
#include <stdlib.h>
#include <elf.h>

static unsigned int getsys(char **envp) {
        Elf32_auxv_t *auxv;

        /* walk past all env pointers */
        while (*envp++ != NULL)
                ;
        /* and find ELF auxiliary vectors (if this was an ELF binary) */
        auxv = (Elf32_auxv_t *) envp;

        for ( ; auxv->a_type != AT_NULL; auxv++)
                if (auxv->a_type == AT_SYSINFO)
                        return auxv->a_un.a_val;

        fprintf(stderr, "no AT_SYSINFO auxv entry found\n");
        exit(1);
}

unsigned int sys, pid;

int main(int argc, char **argv, char **envp) {
        sys = getsys(envp);
        __asm__(
"               movl $20, %eax  \n"     /* getpid system call */
"               call *sys       \n"     /* vsyscall */
"               movl %eax, pid  \n"     /* get result */
```

```
        );
        printf("pid is %d\n", pid);
        return 0;
}
```

In the auxv vector one may find AT_SYSINFO data, which points at the vsyscall entry address, and AT_SYSINFO_EHDR data, which points at the start of the vsyscall page.

Maybe in the very beginning `call *%gs:0x18` worked as replacement for the old `int $0x80`. I have never seen a library version that actually used `0x18`. The `0x18` here is the offset of the `sysinfo` field in the `struct tcb_head` at the start of the glibc TLS (thread-local storage) segment. It is `0x10` on i386 and x86_64 (in 32-bit mode) in all sources I have examined.

Let us test, with `getsys()` as above.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/unistd.h>
#include <asm/ldt.h>
#include <elf.h>

...
unsigned int sys, gs, *base;

static void getgs() {
        __asm__("movl %gs, gs\n");
        if ((gs & 7) != 3) {
                fprintf(stderr, "unexpected gs = 0x%x\n", gs);
                exit(1);
        }
}

static void getta(){
        struct user_desc u;
        int i;

        u.entry_number = (gs >> 3);
        if (syscall(__NR_get_thread_area, &u)) {
                perror("get_thread_area");
                exit(1);
        }
        base = (unsigned int *) u.base_addr;

        for (i=0; i<100; i++)
                if (base[i] == sys)
                        goto gotit;
        fprintf(stderr, "didn't find the sysinfo entry\n");
        exit(1);

 gotit:
        printf("Enter the kernel via  call *%%gs:0x%x .\n", 4*i);
}
```

```
int main(int argc, char **argv, char **envp) {
        sys = getsys(envp); printf("sys = 0x%x\n", sys);
        getgs();            printf("gs = 0x%x\n", gs);
        getta();
        return 0;
}
```

And now, on x86_64:

```
% cc -m32 -Wall demo.c -o demo
% ./demo
sys = 0x55573420
gs = 0x63
Enter the kernel via  call *%gs:0x10 .
```

and on i386:

```
% ./demo
sys = 0xffffe414
gs = 0x33
Enter the kernel via  call *%gs:0x10 .
```

And indeed this works:

```
% cat exit42.c
int main() {
__asm__(
"               movl $1, %eax  \n"
"               movl $42, %ebx \n"
"               call *%gs:0x10 \n"
);
}
% cc -m32 exit42.c -o x
% ./x; echo $?
42
```

Next Previous Contents