

# Advanced Operating Systems and Virtualization

## [4] System Calls



Department of Computer,  
Control and Management  
Engineering "A. Ruberti",  
Sapienza University of Rome

# Outline

1. Introduction
2. Handler / Dispatcher
3. Invoking Process
  1. User Space Invoking process
  2. Kernel Wrapper Routines
  3. X86\_64 Invoking Process
4. vDSO
5. Conclusions

## 4.1

### 4. System Calls

# Introduction

# System Calls

Operating Systems offer processes running in User Mode a set of interfaces to interact with hardware devices. This extra layer between applications and hardware has several advantages:

1. making programming easier by freeing programmers to study low-level programming for hardware devices
2. increasing system security because the kernel can check the accuracy of the request at the interface level before attempting to satisfy it
3. increasing the programs portability because they can be compiled and executed correctly on every kernel that offers the same set of interfaces

Linux implements most interfaces between User Mode and Kernel mode by means of system calls.

# POSIX APIs

There is a difference between an API and a system call. Since the former is a function definition and the latter is an explicit request to the kernel made via a software interrupt.

Most of the system calls API that are provided to programmers are given by the libc and they refer to **wrapper routines** whose purpose is the one of invoking a system call. Usually, each system call has a corresponding wrapper routine but the converse is not true:

- the API could offer services directly in User Mode
- a single API function could make several system calls
- some API could wrap extra functions, for instance `malloc()`, `calloc()` and `free()` all use the `brk()` system call to enlarge or reduce the process heap and they keep track of the allocations

The POSIX standard only refers to API and not to system calls, a system that is POSIX compliant offers the set of POSIX APIs.

## 4.2

### 4. System Calls

# Handler / Dispatcher

# System Calls Handler

When a User Mode process invokes a system call the CPU switches to Kernel Mode and starts the execution of a kernel function. In the 8086 system calls can be invoked in two ways but both end with a jump to an assembly language function that is called the **system call handler**.

Each system call is identified by a **system call number** which must be expressed by the user mode process before starting the invoking process. This must usually be passed in the EAX register. All the system calls return an integer value, in general a positive or 0 indicates success, while negative values indicate error, in particular the negation of the error code -- the kernel does not set **errno**, that is set by wrapper routines.

The system call handler is very similar to other exception handlers (that we will see later in the course).

# System Call Handler

V2.4

The system call handler, when invoked:

1. saves the content of most registers in the Kernel Mode stack
2. handles the system call by invoking a corresponding C function called **system call service routine** (via a `call`)
3. after completing the execution of the system call the registers are loaded with the values saved in the Kernel Mode stack and the CPU is switched back to User Mode

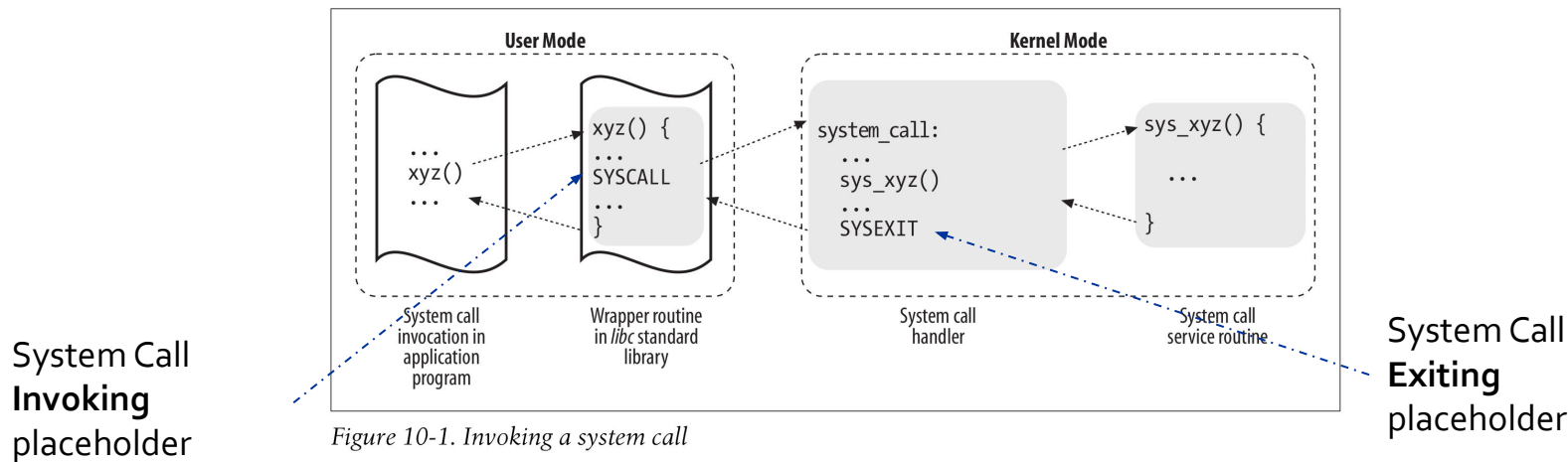
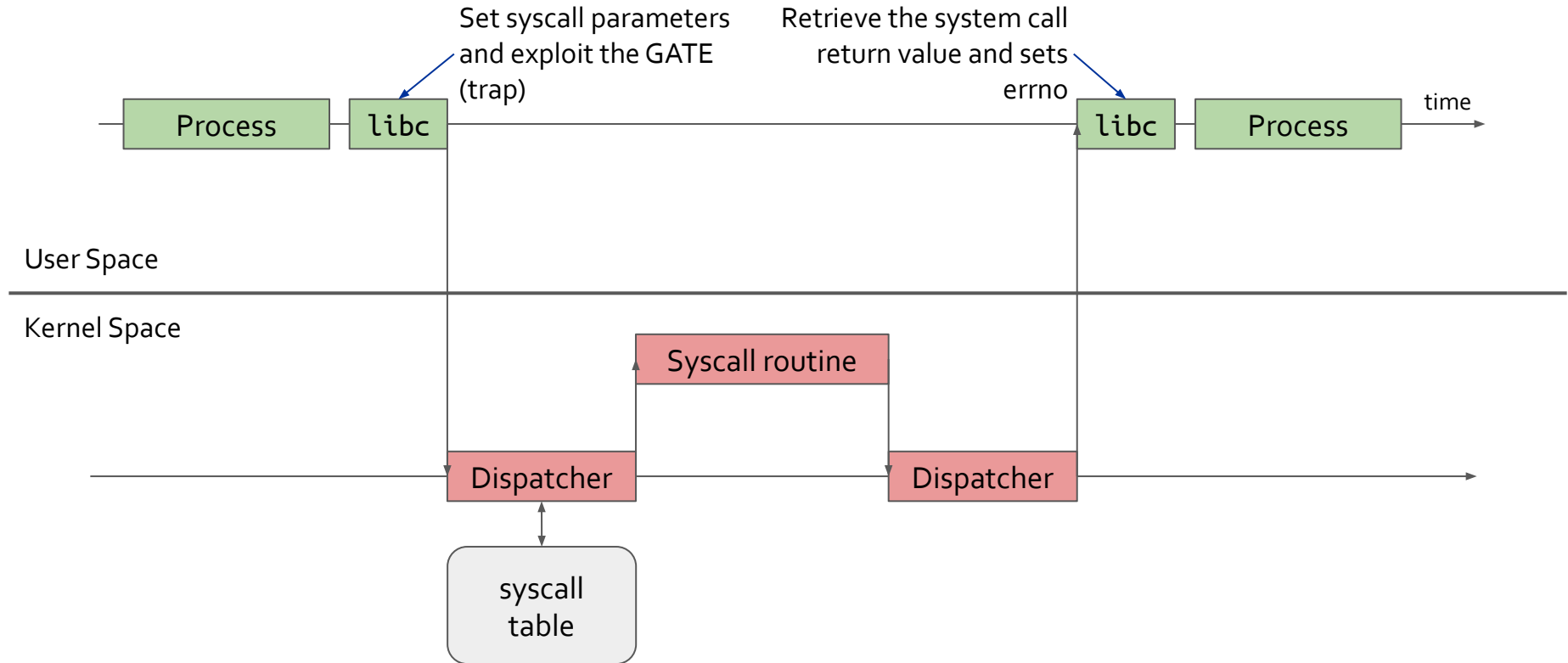


Figure 10-1. Invoking a system call

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. "O'Reilly Media, Inc.", 2005.



# System Call Flow



# System Call Dispatch Table

V2.4

To associate each system call number with its corresponding service routine the kernel uses a **system call dispatch table**, which is stored in a fixed size array called `sys_call_table` array and has `NR_syscall` entries (289 in v2.6), the  $n^{\text{th}}$  entry of the array contains the address to the service routine for the syscall  $n$ .

Remind that `NR_syscall` is not the actual number of implemented system calls, is only the size of the possible maximum number of system calls, therefore there are free slots. In general the not-used entries points to `sys_ni_syscall()` which is the service routine for the “Non-implemented” system calls that always returns `-ENOSYS`.

## **kernel/timer.c**

```
asmlinkage long sys_getuid(void)
{
    /* Only we change this so SMP safe */
    return current->uid;
}
```

**Example of simple syscall service routine**

# System Call Dispatcher

aka `system_call()`

```

196  * Return to user mode is not as complex as all this looks,
197  * but we want the default path for a system call return to
198  * go as quickly as possible which is why some of this is
199  * less clear than it otherwise should be.
200  */
201
202 ENTRY(system_call)
203     pushl %eax                # save orig_eax
204     SAVE_ALL
205     GET_CURRENT(%ebx)
206     testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
207     jne tracesys
208     cmpl $(NR_syscalls),%eax
209     jae badsys
210     call *SYMBOL_NAME(sys_call_table)(,%eax,4)
211     movl %eax,EAX(%esp)      # save the return value
212 ENTRY(ret_from_sys_call)
213     cli                      # need_resched and signals atomic test
214     cmpl $0,need_resched(%ebx)
215     jne reschedule
216     cmpl $0,sigpending(%ebx)
217     jne signal_return
218 restore_all:
219     RESTORE_ALL

```

<https://elixir.bootlin.com/linux/2.4.31/source/arch/i386/kernel/entry.S#L202>

## 4.3

### 4. System Calls

# Invoking Process

# Entering and Exiting a System Call

Native applications can invoke a system call in two different ways:

1. by executing the **int \$0x80** assembly instruction, this was the only way in older versions of the kernel
2. by executing the **sysenter** assembly instruction, introduced from Pentium II and supported from kernel 2.6

Similarly, the kernel can exit from a system call in two ways:

1. by executing **iret** assembly instruction
2. by executing the **sysexit** assembly instruction

The handlers for the two methods are:

1. `system_call()`
2. `sysenter_entry()`

However maintaining the compatibility of both strategies `int/iret` and `sysenter/sysexit` is not easy as it might look for different reasons, for example the kernel should allow to execute the system call even if the `sysenter` instruction is not supported.

# int \$0x80

```
49 #define IA32_SYSCALL_VECTOR
50 #ifdef CONFIG_X86_32
51 # define SYSCALL_VECTOR
52 #endif
```

0x80

0x80

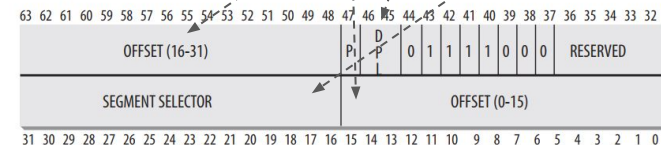
V2.4

The 0x80 is registered during the trap\_init() function as a trap gate.

```
824 void __init trap_init(void)
825 {
836     set_intr_gate(0, &divide_error);
837     set_intr_gate_ist(2, &nmi, NMI_STACK);
838     /* int4 can be called from all */
839     set_system_intr_gate(4, &overflow);
840     set_intr_gate(5, &bounds);
841     set_intr_gate(6, &invalid_op);
842     set_intr_gate(7, &device_not_available);
843 #ifdef CONFIG_X86_32
844     set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
845 #else
846     set_intr_gate_ist(8, &double_fault, DOUBLEFAULT_STACK);
847 #endif
848     set_intr_gate(9, &coprocessor_segment_overrun);
849     set_intr_gate(10, &invalid_TSS);
870 #ifdef CONFIG_X86_32
871     set_system_trap_gate(SYSCALL_VECTOR, &system_call);
872     set_bit(SYSCALL_VECTOR, used_vectors);
873 #endif
881 }
```

```
365 static inline void set_system_trap_gate(unsigned int n, void *addr)
366 {
367     BUG_ON((unsigned)n > 0xFF);
368     _set_gate(n, GATE_TRAP, addr, 0x3, 0, __KERNEL_CS);
369 }
```

Trap Gate Descriptor



System call handler

<https://elixir.bootlin.com/linux/v2.6.39.4/source/arch/x86/kernel/traps.c#L824>

# sysenter/sysexit

## aka Fast System Call

The int assembly instruction is inherently slow, because it performs several consistency and security checks. The sysenter instruction is called Fast System Call by Intel, since it provides a faster way to switch from User to Kernel Mode, the instruction make use of **three MSR registers** (remember they are loaded with wrmsr and read with rdmsr - see Lab#3):

`SYSENTER_CS_MSR`

The Segment Selector of the kernel code segment

`SYSENTER_EIP_MSR`

The linear address of the kernel entry point

`SYSENTER_ESP_MSR`

The kernel stack pointer

Bovet, Daniel P., and Marco Cesati. Understanding the Linux Kernel: from I/O ports to process management. "O'Reilly Media, Inc.", 2005.

<https://wiki.osdev.org/SYSENTER>

# The vsyscall page

v2.6

Obviously a libc wrapper can use the `sysenter` instruction only if both the CPU and the Linux kernel supports it. This compatibility problem has a non-trivial solution.

During the kernel initialization phase the function `sysenter_setup()` builds a page frame called vsyscall page, containing a tiny ELF dynamic library. When a process issues an `execve()` system call to start executing an ELF program, the code in the vsyscall page is dynamically linked to the process address space. The code in that page uses the best available instruction to issue a system call, `int $0x80` or `sysenter`.

Whenever a wrapper routine in the libc must invoke a system call it calls the function `__kernel_vsyscall()`, in the vsyscall page.

The vsyscall page has been replaced with the vDSO (see end of this pack of slides).



# sysenter/sysexit

## Procedure

### sysenter

1. CS register set to the value of (SYSENTER\_CS\_MSR) (points to \_\_KERNEL\_CS)
2. EIP register set to the value of (SYSENTER\_EIP\_MSR) (points to sysenter\_entry())
3. SS register set to the sum of (8 plus the value in SYSENTER\_CS\_MSR)
4. ESP register set to the value of (SYSENTER\_ESP\_MSR)

### sysexit

1. CS register set to the sum of (16 + SYSENTER\_CS\_MSR)
2. EIP register set to the value contained in the EDI register
3. SS register set to the sum of (24 + SYSENTER\_CS\_MSR)
4. ESP register set to the value contained in the ECX register

<https://wiki.osdev.org/SYSENTER>

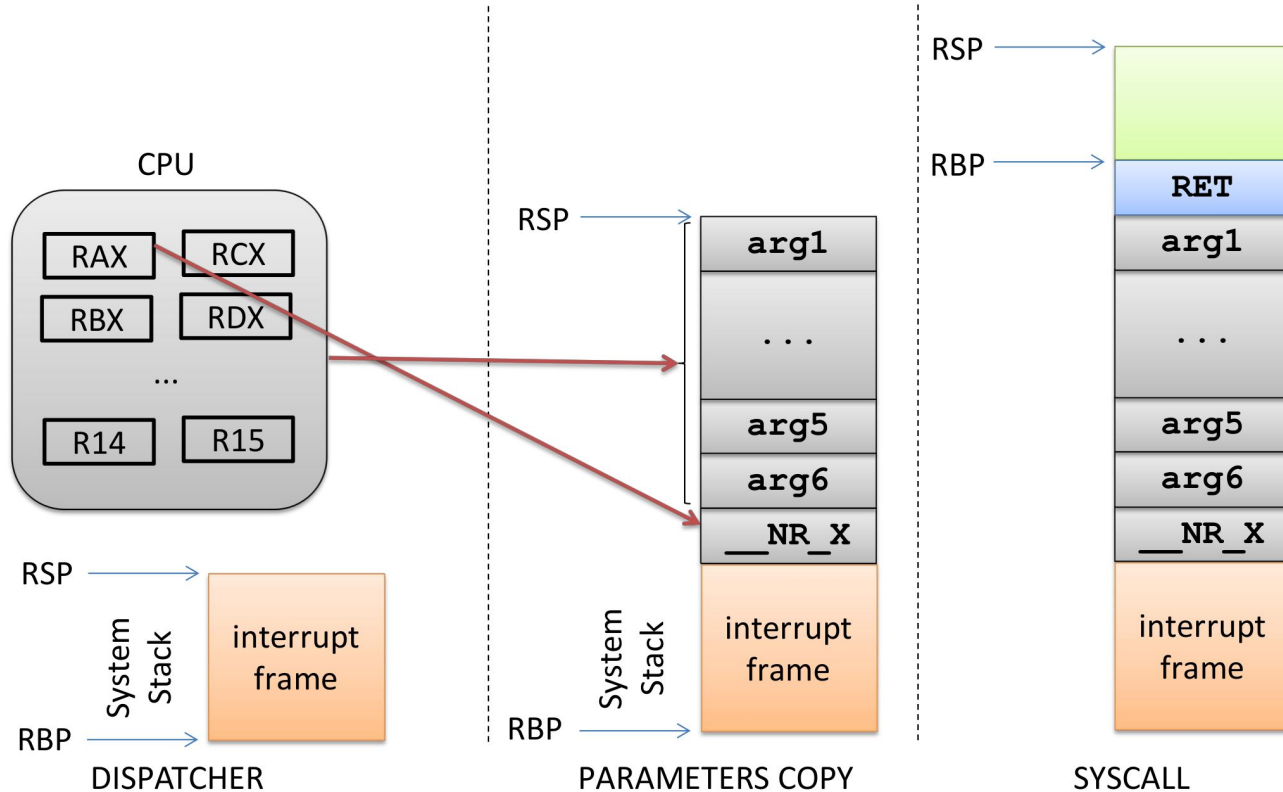
# Parameter Passing

Independently by int/ret or sysenter/sysexit the system call handler has always at least one parameter: the system call number, always passed in the eax register.

The parameters of ordinary C functions are usually passed in the stack (CDECL standard) but since system calls are special functions that cross user and kernel lands, **neither** the user mode **nor** the kernel mode stacks **can be used**. For this reason the parameters are written in CPU registers before issuing the system call. The syscall dispatcher then copies the parameters stored in the CPU registers onto the Kernel Mode stack before invoking the system call service routine because the latter is a standard C function.

Why the kernel does not copy the parameters from the User Mode stack directly into the Kernel Mode one?

# Parameter Passing



# Parameter Passing

However, passing parameters in registers requires two conditions:

- the length is maximum the length of a register (32bit)
- the number of parameters cannot exceed six

In any case we can use pointers to memory areas. The registers used are in order `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi` and `ebp`. The register copy in the stack is done by the `SAVE_ALL` macro and the return code of the `syscall` is always put in `eax`.

In some cases, even if the system call does not use parameters, we need to know the content of CPU registers (e.g. `do_fork()`), in these cases a single parameter of type `pt_regs` allows the service routine to access the values saved in the kernel mode stack by `SAVE_ALL`.

# pt\_regs

V2.4

```
/*
 *    0 (%esp) - %ebx
 *    4 (%esp) - %ecx
 *    8 (%esp) - %edx
 *   C (%esp) - %esi
 *  10 (%esp) - %edi
 *  14 (%esp) - %ebp
 *  18 (%esp) - %eax ← Syscall number
 *  1C (%esp) - %ds
 *  20 (%esp) - %es
 *  24 (%esp) - orig_eax
 *  28 (%esp) - %eip
 *  2C (%esp) - %cs
 *  30 (%esp) - %eflags
 *  34 (%esp) - %oldesp
 *  38 (%esp) - %oldss
 */
```

pt\_regs

arguments

Interrupt frame

4.3.1

4. System Calls  
3. Invoking Process

# User Space Invoking process

# Compile Time Syscall interface

V2.4

The mapping to system call numbers for using in a user space program are defined in the header `include/asm-xxx/unistd.h`.

In that header we will find:

- **system call numerical codes**, that are numbers used to invoke a syscall for userspace and also a displacement in the syscall table for kernel space
- the **Kernel Wrapper Routines**, namely standard **macros** to let userspace access the **gate** to the Kernel, there is a macro for each range of parameters, from 0 to 6

# System Call Codes

V2.4

```
4  /*
5   * This file contains the system call numbers.
6   */
7
8  #define __NR_exit          1
9  #define __NR_fork         2
10 #define __NR_read         3
11 #define __NR_write        4
12 #define __NR_open         5
13 #define __NR_close        6
14 #define __NR_waitpid      7
15 #define __NR_creat        8
16 #define __NR_link         9
17 #define __NR_unlink       10
18 #define __NR_execve       11
19 #define __NR_chdir        12
20 #define __NR_time         13
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255 #define __NR_io_submit    248
256 #define __NR_io_cancel    249
257 #define __NR_alloc_hugepages 250
258 #define __NR_free_hugepages 251
259 #define __NR_exit_group   252
```

<https://elixir.bootlin.com/linux/2.4.31/source/include/asm-i386/unistd.h#L8>



# syscall()

syscall() is a construct that has been added in kernel 2.6 for the Pentium 3 chip, it is implemented through glibc (stdlib.h) and its role is to trigger a trap to execute a generic system call.

SYSCALL(2)

Linux Programmer's Manual

SYSCALL(2)

NAME

syscall - indirect system call

SYNOPSIS

```
#include <unistd.h>
#include <sys/syscall.h> /* For SYS_xxx definitions */

long syscall(long number, ...);
```

The first argument is the system call number, the other parameters are the input for the system call code. The function is based on new x86 instructions: sysenter / sysexit or syscall/sysret (initially for AMD chips). See `man syscall` ([L](#)).

# Complete path


If the kernel supports the vsyscall this is the complete path for calling a system call, suppose that you called syscall() from User Space, the function calls \_\_kernel\_vsyscall(), then

1. If the CPU **does not** support sysenter the function is

```
__kernel_vsyscall:  
    int $0x80  
    ret
```

2. If the CPU **supports** sysenter the function is:

```
__kernel_vsyscall:  
    pushl %ecx  
    pushl %edx  
    pushl %ebp  
    movl %esp, %ebp  
    sysenter
```



These registers are going to be used by the system call handler so they are saved

# Complete path

## (1) `int $0x80`

1. `int` raises the interrupt at `0x80` index, that is a Trap Gate associated to the handler `system_call()` routine, namely the System Call Dispatcher
2. The dispatcher saves the CPU registers in the stack with `SAVE_ALL` macro
3. The validity of the system call number is checked against the `NR_syscalls` number
  - a. If not valid the function stores `-ENOSYS` in the `eax` position in the stack and then jumps to `resume_userspace()`
  - b. Otherwise the the system call service routine is called with the number passed in `eax`
4. When the system call service routines terminates `system_calls` gets its return code from `eax` and stores it the `eax` position in the stack
5. The kernel checks if there is some other work to do before returning in user mode (e.g. other interrupts, this will be clearer in next lectures)
6. `RESTORE_ALL` restores the contents of registers

# Complete path

## (2) `sysenter`

1. `ebp`, `edx` and `ecx` content are saved in the stack and `esp` is copied in `ebp`
2. the `sysenter` assembly instruction switches the CPU in kernel mode directly at the function `sysenter_entry()`, the System Call Handler
3. Sets up the kernel stack pointer
4. Enable local interrupts with `sti` command
5. Performs some operations that emulates the `int` assembly instruction
6. Invokes the System Call Service Routine is invoked exactly like `int $0x80` at the start of `system_call()`
7. The `sysexit` assembly instruction is used for returning in User Mode

4.3.2

4. System Calls

3. Invoking Process

# Kernel Wrapper Routines

# Kernel Wrapper Routines

Although system calls are used mainly by User Mode processes, they can also be invoked by kernel threads, which cannot use library functions. To simplify the declarations of the corresponding wrapper routines, Linux defines a set of **seven** macros called `_syscall0` through `_syscall6`, where the number in the name is the number of the pass-able parameters (excluding the system call number).

```
_syscallX(type, name, type1, arg1, ....)
```

## Examples

The wrapper routine to the `fork()` system call could be

```
_syscall0(int, fork)
```


The wrapper routine to `write()` could be:

```
_syscall3(int, write, int, fd, const char*, buf, unsigned int, count)
```

# o-parameters call

V2.4

```
273 #define __syscall0(type,name) \
274 type name(void) \
275 { \
276     long __res; \
277     __asm__ volatile ("int $0x80" \
278                     : "=a" (__res) \
279                     : "0" (__NR_##name)); \
280     __syscall_return(type, __res); \
281 }
```



<https://elixir.bootlin.com/linux/2.4.31/source/include/asm-i386/unistd.h#L8>

# \_\_syscall\_return

V2.4

```
261  /* user-visible error numbers are in the range -1 - -124: see <asm-i386/errno.h> */
262
263  #define __syscall_return(type, res) \
264  do { \
265      if ((unsigned long)(res) >= (unsigned long)(-125)) { \
266          errno = -(res); \
267          res = -1; \
268      } \
269      return (type) (res); \
270  } while (0)
```

<https://elixir.bootlin.com/linux/2.4.31/source/include/asm-i386/unistd.h#L263>

?



# 1-parameter call

V2.4

```
283 #define __syscall1(type,name,type1,arg1) \
284 type name(type1 arg1) \
285 { \
286     long __res; \
287     __asm__ volatile ("int $0x80" \
288         : "=a" (__res) \
289         : "0" (__NR_##name),"b" ((long)(arg1))); \
290     __syscall_return(type,__res); \
291 }
```

<https://elixir.bootlin.com/linux/2.4.31/source/include/asm-i386/unistd.h#L283>

# 6-parameters call

V2.4

```
337 #define __syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \  
338         type5,arg5,type6,arg6) \  
339 type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6) \  
340 { \  
341     long __res; \  
342     __asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int $0x80 ; pop %%ebp" \  
343         : "=a" (__res) \  
344         : "i" (__NR_##name), "b" ((long)(arg1)), "c" ((long)(arg2)), \  
345           "d" ((long)(arg3)), "S" ((long)(arg4)), "D" ((long)(arg5)), \  
346           "0" ((long)(arg6))); \  
347     __syscall_return(type, __res); \  
348 }
```

<https://elixir.bootlin.com/linux/2.4.31/source/include/asm-i386/unistd.h#L337>

# Newer kernel versions

V5.11

On latest version of the kernel, the Kernel Wrapper Routines are defined in `tools/include/nolibc/nolibc.h`, again they are specifically available for minimal programs which does not use the libc wrappers. They consists of three levels:

1. the macro assembly routines from `my_syscall0` to `my_syscall6`, architecture dependent (as the previous ones)
2. functions called `sys_<name_of_the_syscall>` which maps to the macros of the first level
3. call definition as libc does, also sets the `errno`

Further information are in the file [`include/nolibc/nolibc.h`](#)

# Do not call system calls from kernel.

V5.11

*System calls are, as stated above, interaction points between userspace and the kernel. Therefore, **system call functions such as `sys_xyzzy()` or `compat_sys_xyzzy()` should only be called from userspace via the syscall table**, but not from elsewhere in the kernel. If the syscall functionality is useful to be used within the kernel, needs to be shared between an old and a new syscall, or needs to be shared between a syscall and its compatibility variant, it should be implemented by means of a **“helper” function (such as `kysys_xyzzy()`)**. This kernel function may then be called within the syscall stub (`sys_xyzzy()`), the compatibility syscall stub (`compat_sys_xyzzy()`), and/or other kernel code.*

-- <https://www.kernel.org/doc/html/latest/process/adding-syscalls.html>

# Syscall Table

V5.11

The kernel level system call table is defined in specific files:

- for Kernel 2.4.20 on i386 it is defined in `arch/i386/kernel/entry.S`
- for Kernel 2.6 is in `arch/x86/kernel/syscall_table32.S`
- more recent versions: `arch/x86/entry/syscalls/syscall_32.tbl`

The entries in the table keep a reference to the kernel-level system call implementation and typically, the kernel-level name of the system call service routine resembles the one used at application level but starts with the “`sys_`” prefix.

# Syscall Table

V5.11

## For x86 architecture

```
1  #
2  # 32-bit system call numbers and entry vectors
3  #
4  # The format is:
5  # <number> <abi> <name> <entry point> <compat entry point>
6  #
7  # The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
8  # sys_*( ) system calls and compat_sys_*( ) compat system calls if
9  # IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
10 # parameter.
11 #
12 # The abi is always "i386" for this file.
13 #
14 0      i386      restart_syscall      sys_restart_syscall
15 1      i386      exit                  sys_exit
16 2      i386      fork                  sys_fork
17 3      i386      read                  sys_read
18 4      i386      write                  sys_write
19 5      i386      open                  sys_open                  compat_sys_op
20 6      i386      close                  sys_close
21 7      i386      waitpid                 sys_waitpid
443 436   i386      close_range            sys_close_range
444 437   i386      openat2                 sys_openat2
445 438   i386      pidfd_getfd             sys_pidfd_getfd
446 439   i386      faccessat2              sys_faccessat2
447 440   i386      process_madvise          sys_process_madvise
448 441   i386      epoll_pwait2             sys_epoll_pwait2              compat_sys_
```

[https://elixir.bootlin.com/linux/v5.11/source/arch/x86/entry/syscalls/syscall\\_32.tbl](https://elixir.bootlin.com/linux/v5.11/source/arch/x86/entry/syscalls/syscall_32.tbl)

# Defining a syscall service routine

V5.11

```
642 SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
643 {
644     return ksys_read(fd, buf, count);
645 }
```

System call actual implementation

[https://elixir.bootlin.com/linux/v5.11/source/fs/read\\_write.c#L642](https://elixir.bootlin.com/linux/v5.11/source/fs/read_write.c#L642)

```
213 #define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, ##name, __VA_ARGS__)
214 #define SYSCALL_DEFINE2(name, ...) SYSCALL_DEFINEx(2, ##name, __VA_ARGS__)
215 #define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, ##name, __VA_ARGS__)
216 #define SYSCALL_DEFINE4(name, ...) SYSCALL_DEFINEx(4, ##name, __VA_ARGS__)
217 #define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, ##name, __VA_ARGS__)
218 #define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, ##name, __VA_ARGS__)
219
220 #define SYSCALL_DEFINE_MAXARGS 6
221
222 #define SYSCALL_DEFINEx(x, sname, ...) \
223     SYSCALL_METADATA(sname, x, __VA_ARGS__) \
224     __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

<https://elixir.bootlin.com/linux/v5.11/source/include/linux/syscalls.h#L215>

The `__se_sys_*` stub is created for further protection and in the end calls `__do_sys_*` which calls the original `ksys_read`

```
228 /*
229  * The asmlinkage stub is aliased to a function named __se_sys_*() which
230  * sign-extends 32-bit ints to longs whenever needed. The actual work is
231  * done within __do_sys_*().
232  */
233 #ifndef __SYSCALL_DEFINEx
234 #define __SYSCALL_DEFINEx(x, name, ...) \
235     __diag_push(); \
236     __diag_ignore(GCC, 8, "-Wattribute-alias", \
237         "Type aliasing is used to sanitize syscall arguments"); \
238     asmlinkage long sys##name(__MAP(x,__SC_DECL,__VA_ARGS__)) \
239         __attribute__((alias(__stringify(__se_sys##name))))); \
240     ALLOW_ERROR_INJECTION(sys##name, ERRNO); \
241     static inline long __do_sys##name(__MAP(x,__SC_DECL,__VA_ARGS__)); \
242     asmlinkage long __se_sys##name(__MAP(x,__SC_LONG,__VA_ARGS__)); \
243     asmlinkage long __se_sys##name(__MAP(x,__SC_LONG,__VA_ARGS__)) \
244     { \
245         long ret = __do_sys##name(__MAP(x,__SC_CAST,__VA_ARGS__)); \
246         __MAP(x,__SC_TEST,__VA_ARGS__); \
247         __PROTECT(x, ret, __MAP(x,__SC_ARGS,__VA_ARGS__)); \
248         return ret; \
249     } \
250     __diag_pop(); \
251     static inline long __do_sys##name(__MAP(x,__SC_DECL,__VA_ARGS__)) \
252     #endif /* __SYSCALL_DEFINEx */
```

4.3.3

4. System Calls

3. Invoking Process

# x86\_64 Invoking process



# syscall/sysret

On x86\_64 by AMD, there is a similar Fast System Call strategy that is based on the syscall and sysret assembly instructions. Again:

- it is based on MSR registers
- it is involved in the vsyscall page, now improved and called vDSO

During the initialization phase of the kernel the function `syscall_init()` initializes the registers

```
1749 void syscall_init(void)
1750 {
1751     wrmsr(MSR_STAR, 0, ((__USER32_CS << 16) | __KERNEL_CS));
1752     wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);
1753
1754     wrmsrl(MSR_CSTAR, (unsigned long)ignore_sysret);
1755     wrmsrl_safe(MSR_IA32_SYSENTER_CS, (u64)GDT_ENTRY_INVALID_SEG);
1756     wrmsrl_safe(MSR_IA32_SYSENTER_ESP, 0ULL);
1757     wrmsrl_safe(MSR_IA32_SYSENTER_EIP, 0ULL);
1758 }
```

<https://elixir.bootlin.com/linux/v2.6.39.4/source/arch/x86/vdso/vdso32-setup.c#L283>

<https://wiki.osdev.org/SYSENTER>

# Syscall Calling Conventions

```
/*
 * Register setup:
 * rax system call number
 * rdi arg0
 * rcx ret.address for syscall/sysret, userspace arg3
 * rsi arg1
 * rdx arg2
 * r10 arg3 (--> to rcx for userspace)
 * r8 arg4
 * r9 arg5
 * r11 eflags for syscall/sysret, temporary for C
 * r12-r15,rbp,rbx saved by C code, not touched.
 *
 * Interrupts are off on entry.
 * Only called from user space.
 */
```

[https://elixir.bootlin.com/linux/v5.11/source/arch/x86/entry/entry\\_64.S](https://elixir.bootlin.com/linux/v5.11/source/arch/x86/entry/entry_64.S)

# Compatibility

Intel x86\_64 ISA and AMD are similar but they are not the same. In particular

- *In 64-bit Long Mode - only SYSCALL works on both ISAs. (SYSENTER doesn't work on AMD.)*
- *In Legacy Mode - only SYSENTER works on both ISAs. (SYSCALL doesn't work on Intel.)*
- *There's no single instruction that works on both Intel and AMD in Compatibility Mode (SYSENTER doesn't work on AMD and SYSCALL doesn't work on Intel), but there's no need for one. A 32-bit kernel will stay in Legacy Mode after boot.*

<https://reverseengineering.stackexchange.com/questions/16454/struggling-between-syscall-or-sysenter-windows>

## 4.4

### 4. System Calls

# vDSO

# From vsyscall to vDSO

The vsyscall page had several limitations:

- it was fixed in size
- it was allocated always at the same address in processes

The vDSO that stands for Virtual Dynamic Shared Object has been introduced for solving the security issues of the vsyscall architecture. *The vDSO is dynamically allocated which solves security concerns. The vDSO links are provided via the glibc library. The linker will link in the glibc vDSO functionality, provided that such a routine has an accompanying vDSO version, such as gettimeofday system call. When your program executes, if your kernel does not have vDSO support, a traditional syscall will be made.*

<https://lwn.net/Articles/446528/>

# Exposing vDSO

```
#include <sys/auxv.h>
void *vdso = (uintptr_t) getauxval(AT_SYSINFO_EHDR);
```

The vDSO is a small shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the vDSO is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take care of using any functionality that is available via the vDSO.

<https://linux.die.net/man/3/getauxval>

# vDSO Entry Point

```
__kernel_vsyscall:  
    push %ecx  
    push %edx  
    push %ebp  
    movl %esp,%ebp  
    sysenter  
    nop  
    /* 14: System call restart point is here! */  
    int $0x80  
    /* 16: System call normal return point is here! */  
    pop %ebp  
    pop %edx  
    pop %ecx  
    ret
```

# vDSO Content

```
16  /* The ELF entry point can be used to set the AT_SYSINFO value.  */
17  ENTRY(__kernel_vsyscall);
18
19  /*
20   * This controls what userland symbols we export from the vDSO.
21   */
22  VERSION
23  {
24      LINUX_2.6 {
25          global:
26              __vdso_clock_gettime;
27              __vdso_gettimeofday;
28              __vdso_time;
29              __vdso_clock_getres;
30              __vdso_clock_gettime64;
31      };
32
33      LINUX_2.5 {
34          global:
35              __kernel_vsyscall;
36              __kernel_sigreturn;
37              __kernel_rt_sigreturn;
38          local: *;
39      };
40  }
```

<https://elixir.bootlin.com/linux/v5.11/source/arch/x86/entry/vdso/vdso32/vdso32.lds.S#L17>



# Remarks

The vDSO Kernel entry point exploits flat addressing to bypass segmentation and the related operations, it therefore reduces the number of accesses to memory in order to support the change to kernel mode.

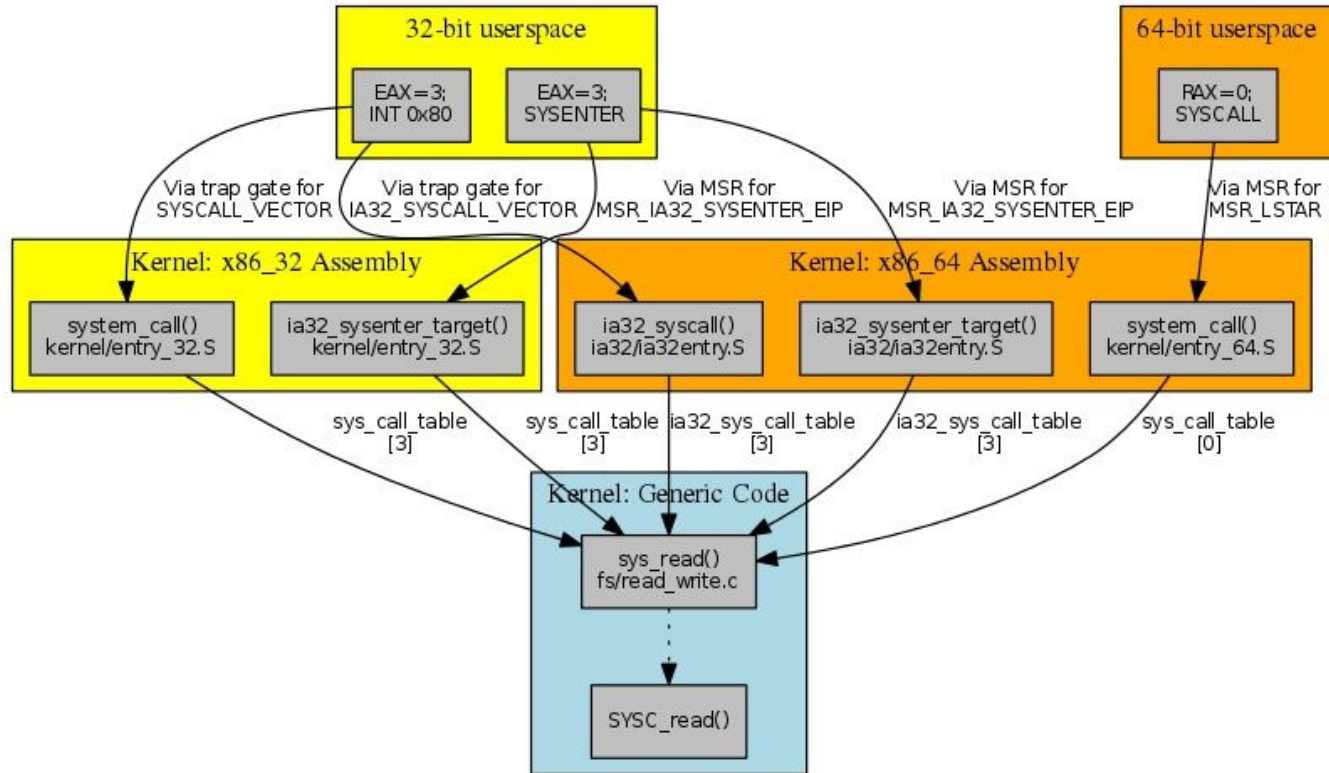
Studies show that the reduction of clock cycles for system calls can be in the order of 75%

4.5

4. System Calls

# Conclusions

# Epilogue



# Advanced Operating Systems and Virtualization

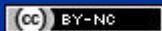
[4] System Calls

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

<http://www.ce.uniroma2.it/~pellegrini/>



gpm.name · [proiettimattia@diag.uniroma1.it](mailto:proiettimattia@diag.uniroma1.it)

DIAG