SO2 Lecture 02 - System calls

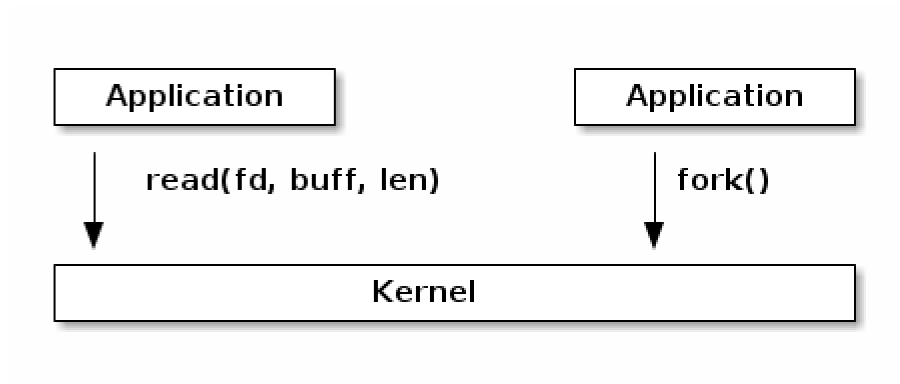
View slides

Lecture objectives:

- Linux system calls implementation
- VDSO and virtual syscalls
- Accessing user space from system calls

Linux system calls implementation

At a high level system calls are "services" offered by the kernel to user applications and they resemble library APIs in that they are described as a function call with a name, parameters, and return value.



However, on a closer look, we can see that system calls are actually not function calls, but specific assembly instructions (architecture and kernel specific) that do the following:

- setup information to identify the system call and its parameters
- trigger a kernel mode switch
- retrieve the result of the system call

In Linux, system calls are identified by numbers and the parameters for system calls are machine word sized (32 or 64 bit). There can be a maximum of 6 system call parameters. Both the system call number and the parameters are stored in certain registers.

For example, on 32bit x86 architecture, the system call identifier is stored in the EAX register, while parameters in registers EBX, ECX, EDX, ESI, EDI, EBP.

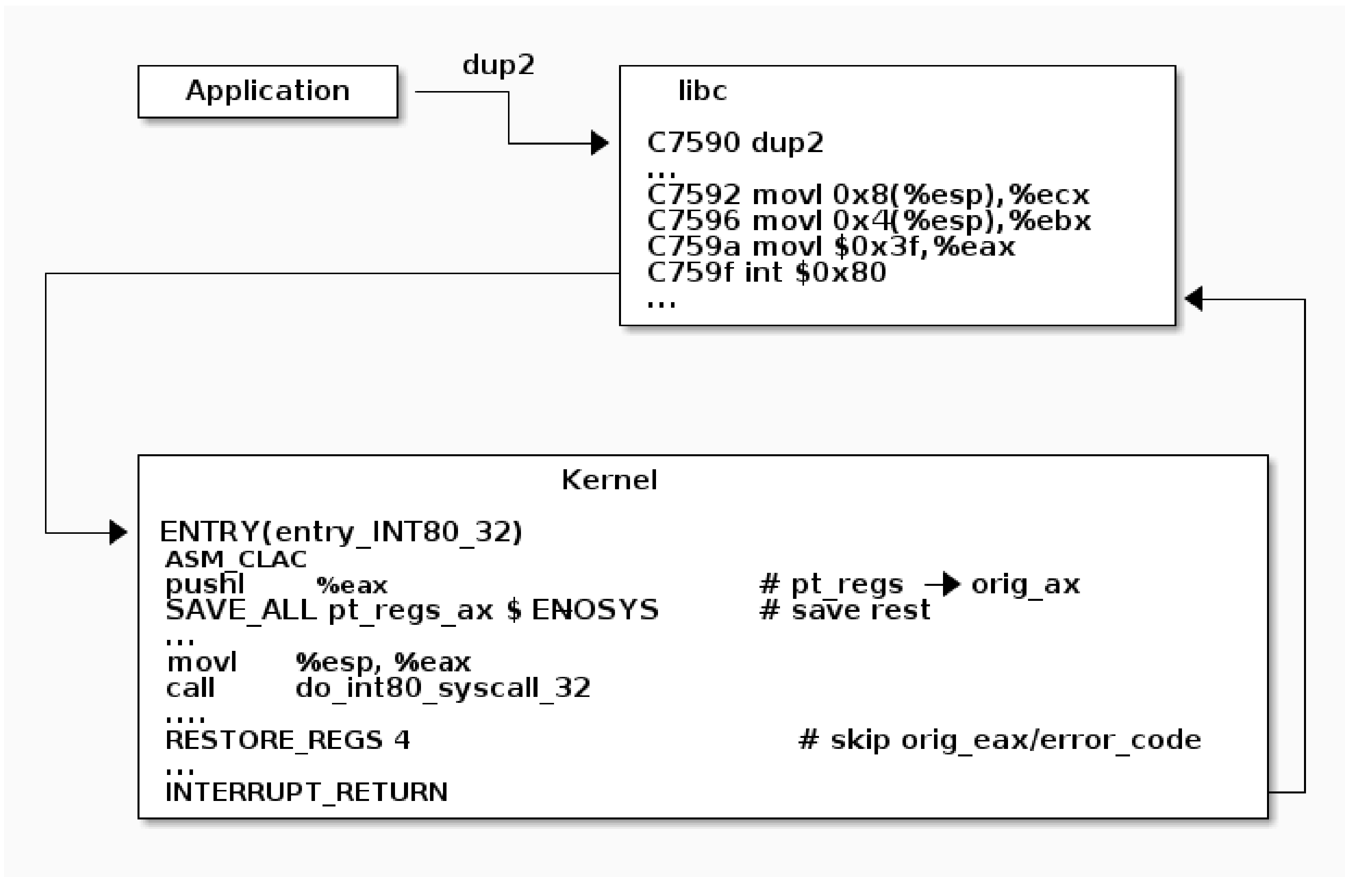
System libraries (e.g. libc) offers functions that implement the actual system calls in order to make it easier for applications to use them.

When a user to kernel mode transition occurs, the execution flow is interrupted and it is transferred to a kernel entry point. This is similar to how interrupts and exceptions are handled (in fact on some architectures this transition happens as a result of an exception).

The system call entry point will save registers (which contains values from user space, including system call number and system call parameters) on stack and then it will continue with executing the system call dispatcher.

Note

During the user - kernel mode transition the stack is also switched from the user stack to the kernel stack. This is explained in more details in the interrupts lecture.



```
C7590 dup2
...
C7592 movl 0x8(%esp),%ecx
C7596 movl 0x4(%esp),%ebx
C759a movl $0x3f,%eax
C759f int $0x80
...
```

```
ENTRY(entry_INT80_32)
 ASM_CLAC
 pushl      %eax                          # pt_regs → orig_ax
 SAVE_ALL pt_regs_ax $ ENOSYS             # save rest
 ...
 movl     %esp, %eax
 call     do_int80_syscall_32
 ....
 RESTORE_REGS 4                            # skip orig_eax/error_code
 ...
 INTERRUPT_RETURN
```

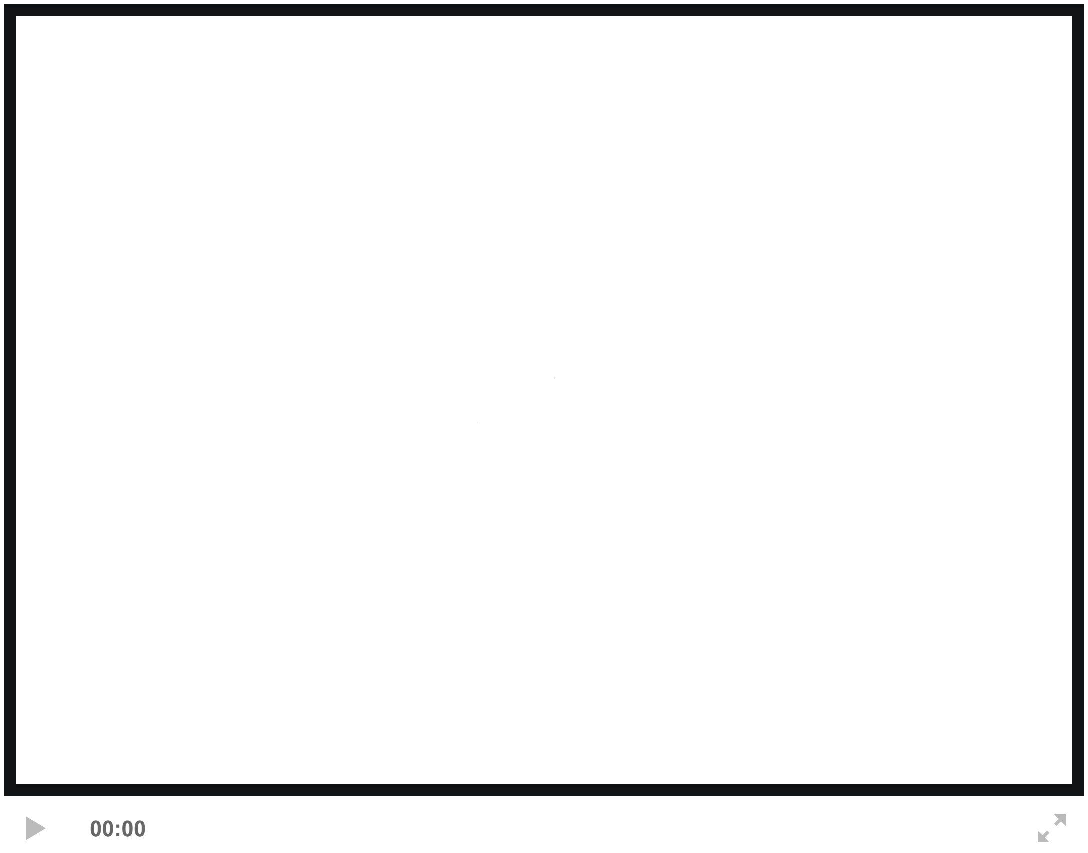
The purpose of the system call dispatcher is to verify the system call number and run the kernel function associated with the system call.

```
/* Handles int $0x80 */
__visible void do_int80_syscall_32(struct pt_regs *regs)
{
    enter_from_user_mode();
    local_irq_enable();
    do_syscall_32_irqs_on(regs);
}

/* simplified version of the Linux x86 32bit System Call Dispatcher */
static __always_inline void do_syscall_32_irqs_on(struct pt_regs *regs)
{
    unsigned int nr = regs->orig_ax;

    if (nr < IA32_NR_syscalls)
        regs->ax = ia32_sys_call_table[nr](regs->bx, regs->cx,
                                           regs->dx, regs->si,
                                           regs->di, regs->bp);
    syscall_return_slowpath(regs);
}
```

To demonstrate the system call flow we are going to use the virtual machine setup, attach gdb to a running kernel, add a breakpoint to the dup2 system call and inspect the state.



In summary, this is what happens during a system call:

- The application is setting up the system call number and parameters and it issues a trap instruction

- The execution mode switches from user to kernel; the CPU switches to a kernel stack; the user stack and the return address to user space is saved on the kernel stack
- The kernel entry point saves registers on the kernel stack
- The system call dispatcher identifies the system call function and runs it
- The user space registers are restored and execution is switched back to user (e.g. calling IRET)
- The user space application resumes

System call table

The system call table is what the system call dispatcher uses to map system call numbers to kernel functions:

```
#define __SYSCALL_I386(nr, sym, qual) [nr] = sym,

const sys_call_ptr_t ia32_sys_call_table[] = {
  [0 ... __NR_syscall_compat_max] = &sys_ni_syscall,
  #include <asm/syscalls_32.h>
};
```

```
__SYSCALL_I386(0, sys_restart_syscall)
__SYSCALL_I386(1, sys_exit)
__SYSCALL_I386(2, sys_fork)
__SYSCALL_I386(3, sys_read)
__SYSCALL_I386(4, sys_write)
#ifdef CONFIG_X86_32
__SYSCALL_I386(5, sys_open)
#else
__SYSCALL_I386(5, compat_sys_open)
#endif
__SYSCALL_I386(6, sys_close)
```

System call parameters handling

Handling system call parameters is tricky. Since these values are setup by user space, the kernel can not assume correctness and must always verify them thoroughly.

Pointers have a few important special cases that must be checked:

- Never allow pointers to kernel-space
- Check for invalid pointers

Since system calls are executed in kernel mode, they have access to kernel space and if pointers are not properly checked user applications might get read or write access to kernel space.

For example, let's consider the case where such a check is not made for the read or write system calls. If the user passes a kernel-space pointer to a write system call then it can get access to kernel data by later reading the file. If it passes a kernel-space pointer to a read system call then it can corrupt kernel memory.

Likewise, if a pointer passed by the application is invalid (e.g. unmapped, read-only for cases where it is used for writing), it could "crash" the kernel. Two approaches could be used:

- Check the pointer against the user address space before using it, or
- Avoid checking the pointer and rely on the MMU to detect when the pointer is invalid and use the page fault handler to determine that the pointer was invalid

Although it sounds tempting, the second approach is not that easy to implement. The page fault handler uses the fault address (the address that was accessed), the faulting address (the address of the instruction that did the access) and information from the user address space to determine the cause:

  - Copy on write, demand paging, swapping: both the fault and faulting addresses are in user space; the fault address is valid (checked against the user address space)
  - Invalid pointer used in system call: the faulting address is in kernel space; the fault address is in user space and it is invalid
  - Kernel bug (kernel accesses invalid pointer): same as above

But in the last two cases we don't have enough information to determine the cause of the fault.

In order to solve this issue, Linux uses special APIs (e.g `copy_to_user()`) to accesses user space that are specially crafted:

- The exact instructions that access user space are recorded in a table (exception table)
- When a page fault occurs the faulting address is checked against this table

Although the fault handling case may be more costly overall depending on the address space vs exception table size, and it is more complex, it is optimized for the common case and that is why it is preferred and used in Linux.

Cost	Pointer checks	Fault handling
Valid address	address space search	negligible
Invalid address	address space search	exception table search

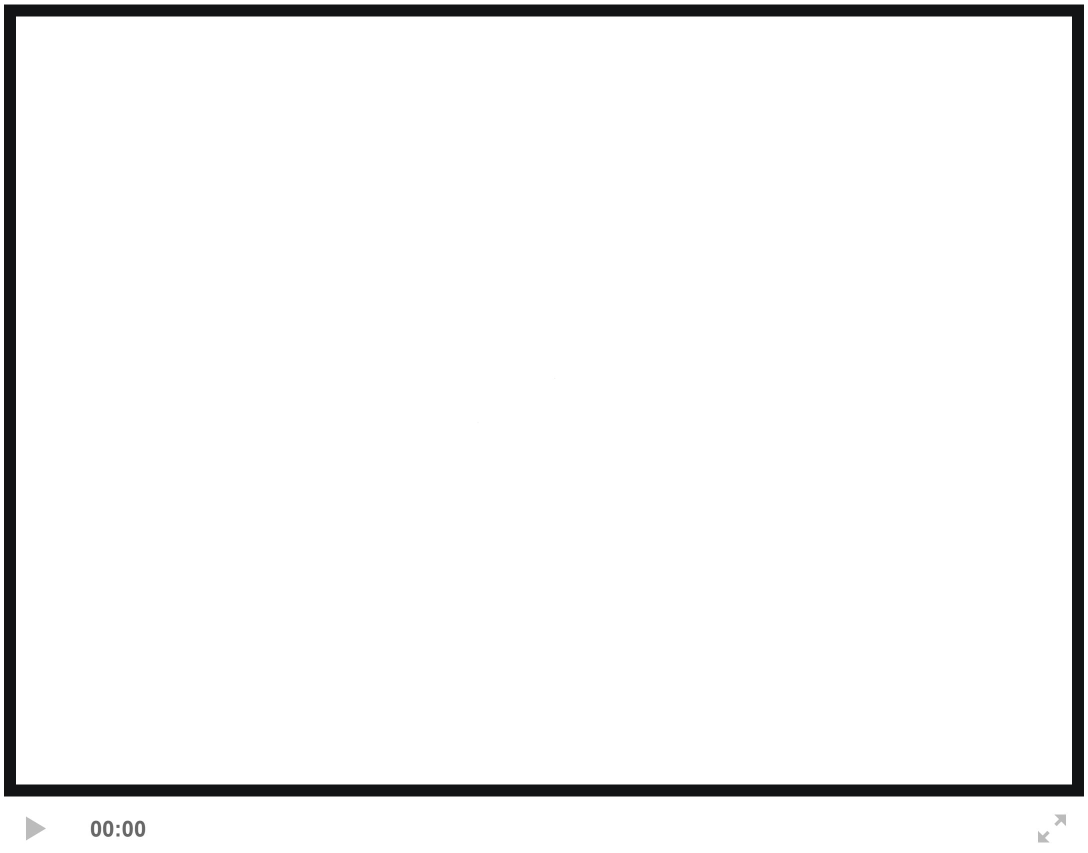
Virtual Dynamic Shared Object (VDSO)

The VDSO mechanism was born out of the necessity of optimizing the system call implementation, in a way that does not impact libc with having to track the CPU capabilities in conjunction with the kernel version.

For example, x86 has two ways of issuing system calls: int 0x80 and sysenter. The latter is significantly faster so it should be used when available. However, it is only available for processors newer than Pentium II and only for kernel versions greater than 2.6.

With VDSO the system call interface is decided by the kernel:

- a stream of instructions to issue the system call is generated by the kernel in a special memory area (formatted as an ELF shared object)
- that memory area is mapped towards the end of the user address space
- libc searches for VDSO and if present will use it to issue the system call



An interesting development of the VDSO is the virtual system calls (vsyscalls) which run directly from user space. These vsyscalls are also part of VDSO and they are accessing data from the VDSO page that is either static or modified by the kernel in a separate read-write map of the VDSO page. Examples of system calls that can be implemented as vsyscalls are: getpid or gettimeofday.

- "System calls" that run directly from user space, part of the VDSO
- Static data (e.g. getpid())
- Dynamic data update by the kernel a in RW map of the VDSO (e.g. gettimeofday(), time(),)

Accessing user space from system calls

As we mentioned earlier, user space must be accessed with special APIs (`get_user()` , `put_user()` , `copy_from_user()` , `copy_to_user()`) that check whether the pointer is in user space and also handle the fault if the pointer is invalid. In case of invalid pointers, they return a non-zero value.

```
/* OK: return -EFAULT if user_ptr is invalid */
if (copy_from_user(&kernel_buffer, user_ptr, size))
    return -EFAULT;

/* NOK: only works if user_ptr is valid otherwise crashes kernel */
memcpy(&kernel_buffer, user_ptr, size);
```

Let's examine the simplest API, get_user, as implemented for x86:

```
#define get_user(x, ptr)                                                  \
({                                                                        \
  int __ret_gu;                                                           \
  register __inttype(*(ptr)) __val_gu asm("%"_ASM_DX);                    \
  __chk_user_ptr(ptr);                                                    \
  might_fault();                                                          \
  asm volatile("call __get_user_%P4"                                      \
               : "=a" (__ret_gu), "=r" (__val_gu),                        \
                  ASM_CALL_CONSTRAINT                                     \
               : "0" (ptr), "i" (sizeof(*(ptr))));                        \
  (x) = (__force __typeof__(*(ptr))) __val_gu;                            \
  __builtin_expect(__ret_gu, 0);                                          \
})
```

The implementation uses inline assembly, which allows inserting ASM sequences in C code and also handles access to/from variables in the ASM code.

Based on the type size of the x variable, one of __get_user_1, __get_user_2 or __get_user_4 will be called. Also, before executing the assembly call, ptr will be moved to the first register EAX while after the completion of assembly part the value of EAX will be moved to __ret_gu and the EDX register will be moved to __val_gu.

It is equivalent to the following pseudo code:

```
#define get_user(x, ptr)                \
    movl ptr, %eax                      \
    call __get_user_1                   \
    movl %edx, x                        \
    movl %eax, result                   \
```

The __get_user_1 implementation for x86 is the following:

```
.text
ENTRY(__get_user_1)
    mov PER_CPU_VAR(current_task), %_ASM_DX
    cmp TASK_addr_limit(%_ASM_DX),%_ASM_AX
    jae bad_get_user
    ASM_STAC
1:  movzbl (%_ASM_AX),%edx
    xor %eax,%eax
    ASM_CLAC
    ret
ENDPROC(__get_user_1)

bad_get_user:
    xor %edx,%edx
    mov $(-EFAULT),%_ASM_AX
    ASM_CLAC
    ret
END(bad_get_user)

_ASM_EXTABLE(1b,bad_get_user)
```

The first two statements check the pointer (which is stored in EDX) with the addr_limit field of the current task (process) descriptor to make sure that we don't have a pointer to kernel space.

Then, SMAP is disabled, to allow access to user from kernel, and the access to user space is done with the instruction at the 1: label. EAX is then zeroed to mark success, SMAP is enabled, and the call returns.

The movzbl instruction is the one that does the access to user space and its address is captured with the 1: label and stored in a special section:

```
/* Exception table entry */
# define _ASM_EXTABLE_HANDLE(from, to, handler)           \
  .pushsection "__ex_table","a" ;                         \
  .balign 4 ;                                             \
  .long (from) - . ;                                      \
  .long (to) - . ;                                        \
  .long (handler) - . ;                                   \
  .popsection

# define _ASM_EXTABLE(from, to)                           \
  _ASM_EXTABLE_HANDLE(from, to, ex_handler_default)
```

For each address that accesses user space we have an entry in the exception table, that is made up of: the faulting address(from), where to jump to in case of a fault, and a handler function (that implements the jump logic). All of these addresses are stored on 32bit in relative format to the exception table, so that they work for both 32 and 64 bit kernels.

All of the exception table entries are then collected in the __ex_table section by the linker script:

```
#define EXCEPTION_TABLE(align)                                      \
  . = ALIGN(align);                                                 \
  __ex_table : AT(ADDR(__ex_table) - LOAD_OFFSET) {                 \
          VMLINUX_SYMBOL(__start___ex_table) = .;                   \
          KEEP(*(__ex_table))                                       \
          VMLINUX_SYMBOL(__stop___ex_table) = .;                    \
  }
```

The section is guarded with __start___ex_table and __stop___ex_table symbols, so that it is easy to find the data from C code. This table is accessed by the fault handler:

```
bool ex_handler_default(const struct exception_table_entry *fixup,
                        struct pt_regs *regs, int trapnr)
{
    regs->ip = ex_fixup_addr(fixup);
    return true;
}

int fixup_exception(struct pt_regs *regs, int trapnr)
{
    const struct exception_table_entry *e;
    ex_handler_t handler;

    e = search_exception_tables(regs->ip);
    if (!e)
        return 0;

    handler = ex_fixup_handler(e);
    return handler(e, regs, trapnr);
}
```

All it does is to set the return address to the one in the field of the exception table entry which, in case of the get_user exception table entry, is bad_get_user which return -EFAULT to the caller.