System Calls

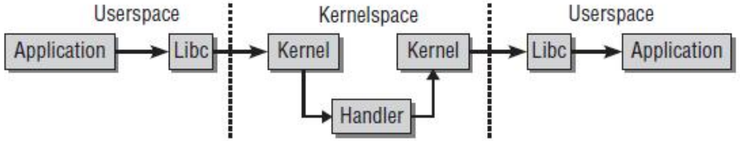
Have a look

- Before getting into **System Calls**, please take a look at
  - LK_Bird's Eye View sessions,

    S3 Virtual space,Privilege Level, System Call, Pagetable, Mapping.

System Calls

- POSIX APIs and the C Library
    - POSIX standard *(Portable Operating System Interface For Unix)* refers to APIs and not to system calls.
    - For instance, in Linux, the ***malloc( ), calloc( ), and free( )*** APIs are implemented in the ***libc*** library.
    - Indeed, the POSIX standard was created to resemble the interfaces provided by earlier Unix systems.
    - From the application programmer's point of view, system calls are irrelevant; all the programmer is concerned with is the API. Conversely, the kernel is concerned only with the system calls.



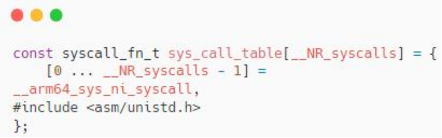
System Calls

- Syscalls, *Cont'd*
  - System Call Numbers
    Each system call is assigned a unique syscall number.

    The kernel keeps a list of all **registered** system calls in the system call table, stored in the architecture specific ***sys_call_table[].***
    typedef long (*syscall_fn_t)(const struct pt_regs *regs);

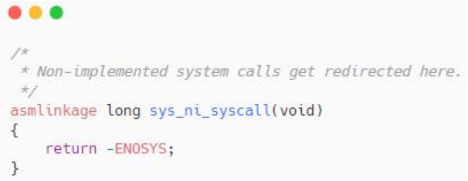
    In case syscall is removed or invalid, Linux provides a "not implemented" system call, ***sys_ni_syscall()***, which returns ***-ENOSYS***, the error corresponding to an invalid syscall.

/arch/arm64/kernel/sys.c



```
const syscall_fn_t sys_call_table[__NR_syscalls] = {
    [0 ... __NR_syscalls - 1] =
__arm64_sys_ni_syscall,
#include <asm/unistd.h>
};
```

/kernel/sys_ni.c



```
/*
 * Non-implemented system calls get redirected here.
 */
asmlinkage long sys_ni_syscall(void)
{
    return -ENOSYS;
}
```

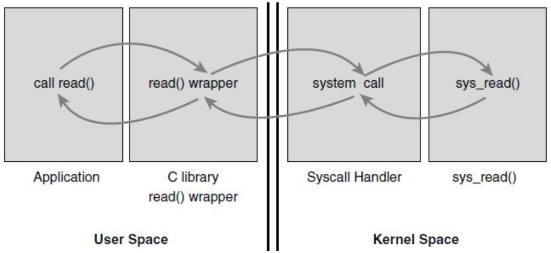
System Calls

- System Call Handling
  - When user-space applications trigger syscall, it signals the kernel through a software interrupt, Incur an exception, and the system will switch to kernel mode and execute the exception handler (**system call handler**).

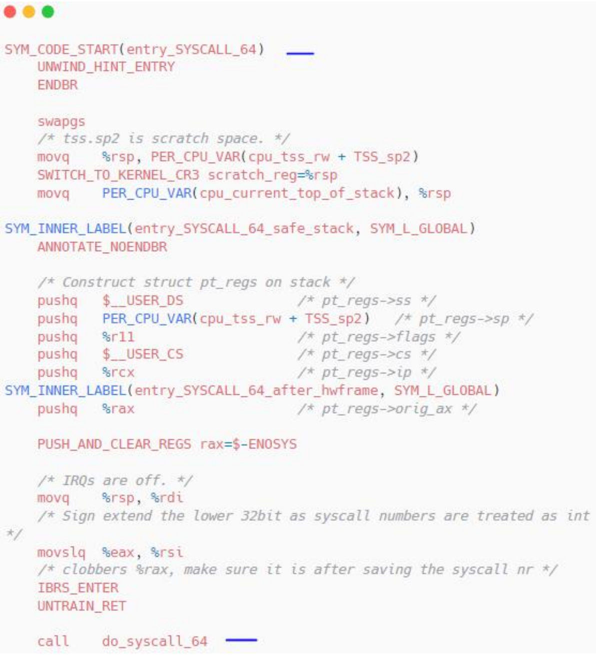
    1) i.e. SWI in x86 is interrupt number 128, which is incurred via the ***int $0x80*** assembly instruction.

    2) i.e. the **system call handler** in x86-64 ***entry_SYSCALL_64*** assembly routine and <u>syscall number</u> is passed to the kernel via the CPU registers.

    3) System Call Binding bind with the coreresponding system call entry inside the table.



/arch/x86/entry/entry_64.S



```
SYM_CODE_START(entry_SYSCALL_64)
    UNWIND_HINT_ENTRY
    ENDBR

    swapgs
    /* tss.sp2 is scratch space. */
    movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
    SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
    movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)
    ANNOTATE_NOENDBR

    /* Construct struct pt_regs on stack */
    pushq   $__USER_DS                  /* pt_regs->ss */
    pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)   /* pt_regs->sp */
    pushq   %r11                        /* pt_regs->flags */
    pushq   $__USER_CS                  /* pt_regs->cs */
    pushq   %rcx                        /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
    pushq   %rax                        /* pt_regs->orig_ax */

    PUSH_AND_CLEAR_REGS rax=$-ENOSYS

    /* IRQs are off. */
    movq    %rsp, %rdi
    /* Sign extend the lower 32bit as syscall numbers are treated as int
*/
    movslq  %eax, %rsi
    /* clobbers %rax, make sure it is after saving the syscall nr */
    IBRS_ENTER
    UNTRAIN_RET

    call    do_syscall_64
```

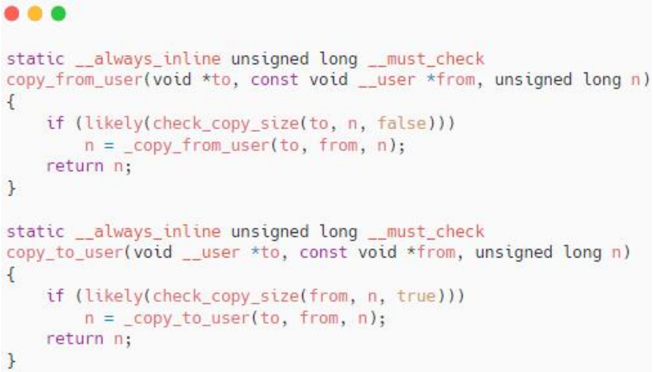
System Calls

- System Call Handling, *Cont'd*
    - Verifying syscall Parameters
      Most important checks are
      - Process **permissions** to acces any memories.

      - The pointer points to a region of memory in the **process's address space.**

      So kernel functions should be used to deal with data from user space ***copy_to_user()***, ***copy_from_user()***

/include/linux/uaccess.h



```
static __always_inline unsigned long __must_check
copy_from_user(void *to, const void __user *from, unsigned long n)
{
    if (likely(check_copy_size(to, n, false)))
        n = _copy_from_user(to, from, n);
    return n;
}

static __always_inline unsigned long __must_check
copy_to_user(void __user *to, const void *from, unsigned long n)
{
    if (likely(check_copy_size(from, n, true)))
        n = _copy_to_user(to, from, n);
    return n;
}
```

System Calls

- Why Not to Implement a System Call
  - System calls are simple to implement and easy to use and performance on Linux is fast.

  - But,
    - You need a syscall number, which needs to be officially assigned to you.

    - After it is written in stone, the interface cannot change without **breaking user-space** applications.

    - Each architecture needs to separately register the system call and support it.

    The alternatives:
    *Implement a device node and **read()** and **write()** to it.*
    *Use **ioctl()** to manipulate specific settings or retrieve specific information.*

    *Add the information as a file to the appropriate location in **sysfs**.*