

CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 3

System Calls

January 30, 2025

Huaicheng Li

Recap: Kernel vs. User Programming

- No libc or standard headers
 - Instead, the kernel implements lots of libc-like functions
- Examples:
 - `#include <string.h>` → `#include <linux/string.h>`
 - `printf("hello!\n")` → `printk(KERN_INFO "hello")`
 - `malloc(64)` → `kmalloc(64, GFP_KERNEL)`
- Linux kernel code uses many GCC extensions
- Inline functions: “static inline void func()”
- Inline assembly: <2%
 - `asm volatile("rdtsc" : "=a" (l), "=d" (h));`
- Branch annotation: hint for better optimization
 - `if (unlikely(error)) { ... }`
 - `if (likely(success)) { ... }`

- No (easy) use of floating point
- Small, fixed-size stack: 8KB (2 pages) in x86
- No memory protection
 - “SIGSEGV” → kernel panic (oops)
- Example of kernel panic: [here](#)

```

SyncMaster 2433
[113975.736571] CPU: 0 PID: 363 Comm: jbd2/md2-8 Tainted: G      D      4.1.6-1-ARCH #1
[113975.782878] Hardware name: Gigabyte Technology Co., Ltd. To be filled by O.E.M./F2a88XM-D3H, BIOS FS 01/09/2014
[113975.830317] task: ffff88007fc7a8c0 ti: ffff88030e6d0000 task.ti: ffff88030e6d0000
[113975.877770] RIP: 0010:[<ffffffffffa013cfb0>] [<ffffffffffa013cfb0>] __find_stripe+0x30/0xc0 [raid456]
[113975.925822] RSP: 0018:ffff88030e6d3978 EFLAGS: 00010006
[113975.973622] RAX: 00ff88030dc6b000 RBX: 0000000000000000 RCX: ffff88030e6d3a10
[113976.021557] RDX: 000000000000006a RSI: 00000000aca346a0 RDI: ffff88030e0a7400
[113976.069029] RBP: ffff88030e6d3998 R08: 0000000000000000 R09: 0000000000000001
[113976.116475] R10: 0000000000000400 R11: 0000000000000001 R12: 00000000aca346a0
[113976.164070] R13: ffff88030e0a7400 R14: ffff88030e0a7608 R15: 0000000000000000
[113976.211211] FS:  00007f3ef27fc700(0000) GS:ffff88031ec00000(0000) knlGS:0000000000000000
[113976.258678] CS:  0010 DS:  0000 ES:  0000 CR0: 000000008005003b
[113976.306880] CR2: 00007f066313f000 CR3: 0000000309f51000 CR4: 0000000000406f0
[113976.353918] Stack:
[113976.401436] ffffffff81812a48 ffff88030e0a7400 00000000aca346a0 ffff88030e0a7410
[113976.450365] ffff88030e6d3a58 ffffffff8142802 ffff880100fb5400 0000000000000246
[113976.499466] ffff88030e6d39f8 0000000000000202 ffff880300000000 00000004810bc249
[113976.548487] Call Trace:
[113976.597097] [<ffffffffffa0142802>] get_active_stripe+0x132/0x730 [raid456]
[113976.646679] [<ffffffffffa0115e0a>] ? md_write_start+0xda/0x1c0 [md_mod]
[113976.696240] [<ffffffffffa0146ec0>] make_request+0x1b0/0xd10 [raid456]
[113976.745790] [<ffffffffff81175a78>] ? mark_page_accessed+0xb8/0xc0
[113976.795430] [<ffffffffff810bc720>] ? wake_atomic_t_function+0x60/0x60
[113976.845197] [<ffffffffffa0112713>] md_make_request+0x103/0x260 [md_mod]
[113976.895063] [<ffffffffff81168aa5>] ? mempool_alloc_slab+0x15/0x20
[113976.945042] [<ffffffffff8129bd00>] generic_make_request+0xe0/0x130
[113976.994959] [<ffffffffff8129bd00>] submit_bio+0x78/0x180
[113977.044854] [<ffffffffffa0204765>] ? jbd2_journal_write_metadata_buffer+0x2d5/0x470 [jbd2]
[113977.095647] [<ffffffffff8121806c>] _submit_bh+0x11c/0x190
[113977.145590] [<ffffffffff812180f0>] submit_bh+0x10/0x20
[113977.194699] [<ffffffffffa01fb37>] jbd2_journal_commit_transaction+0x6b7/0x19f0 [jbd2]
[113977.243946] [<ffffffffff810b2451>] ? put_prev_entity+0x31/0x450
[113977.292964] [<ffffffffff810e527e>] ? try_to_del_timer_sync+0x5e/0x90
[113977.342214] [<ffffffffffa02021da>] kjournald2+0xca/0x260 [jbd2]
[113977.390715] [<ffffffffff810bc720>] ? wake_atomic_t_function+0x60/0x60
[113977.438453] [<ffffffffffa0202110>] ? commit_timeout+0x10/0x10 [jbd2]
[113977.486105] [<ffffffffff81097868>] kthread+0xd8/0xf0
[113977.533435] [<ffffffffff81097790>] ? kthread_worker_fn+0x170/0x170
[113977.579999] [<ffffffffff8150c3a2>] ret_from_fork+0x42/0x70
[113977.625344] [<ffffffffff81097790>] ? kthread_worker_fn+0x170/0x170
[113977.669670] Code: 55 48 89 e5 41 55 41 54 53 49 89 fd 49 89 f4 89 d3 48 83 ec 08 f6 05 19 ed 00 00 04 75 5e 49 8b 45 00 4c 8
9 e2 81 e2 f8 0f 00 00 <48> 8b 04 02 48 85 c0 75 2d 31 db f6 05 d0 ec 00 00 04 75 57 48
[113977.760011] RIP [<ffffffffffa013cfb0>] __find_stripe+0x30/0xc0 [raid456]
[113977.803125] RSP [<ffff88030e6d3978>]
[113977.844910] ---[ end trace 2cb4bf6d46eccf9d ]---
[113977.885740] note: jbd2/md2-8[363] exited with preempt_count 1

```

- Synchronization and concurrency
 - preemptive multitasking → synchronization among tasks
 - » A task can be scheduled and re-scheduled at any time
 - Multi-core processor → synchronization among tasks
 - » A kernel code can execute on two more processors
 - Interrupt → synchronization with interrupt handlers
 - » can occur in the midst of execution (e.g., accessing resources)
 - » need to synchronize with interrupt handler

Linux Kernel Coding Style

- Indentation: 1 tab → 8-character width (not 8 spaces)
- No CamelCase, use underscores: SpinLock → spin_lock
- Use C-style comments: /* use this style*/ not //
- Line length: 80 column
- Write code in a similar style with other kernel code
 - Reference: [Documentation/process/coding-style.rst](#)

```
/*
 * a multi-lines comment
 * (no C++ '//' !)
 */

struct foo {
    int member1;
    double member2;
}; /* no typedef ! */

#ifdef CONFIG_COOL_OPTION
int cool_function(void) {
    return 42;
}
#else
int cool_function(void) { }
#endif /* CONFIG_COOL_OPTION */
```

```
void my_function(int the_param, char
*string, int a_long_parameter,
                int another_long_parameter)
{
    int x = the_param % 42;

    if (!the_param)
        do_stuff();

    switch (x % 3) {
    case 0:
        do_some_stuff();
        cool_function();
        break;
    case 1:
        /* Fall through */
    default:
        do_other_stuff();
        cool_function();
    }
}
```

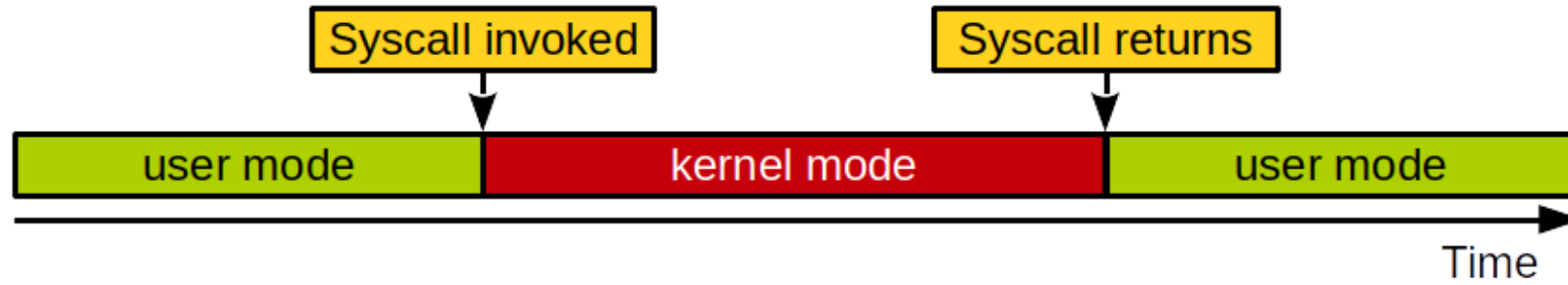
Summary of Tools

- Version control: git, tig
- Configure the kernel: make oldconfig / menuconfig
- Build the kernel: make -j8; make modules -j8
- Install the kernel: make install; make modules_install
- Explore the code: make cscope tags -j8; cscope, ctags
- Editor: *vim*, emacs
- Screen: tmux
- The missing cs education: <https://missing.csail.mit.edu/>
 - watch the videos

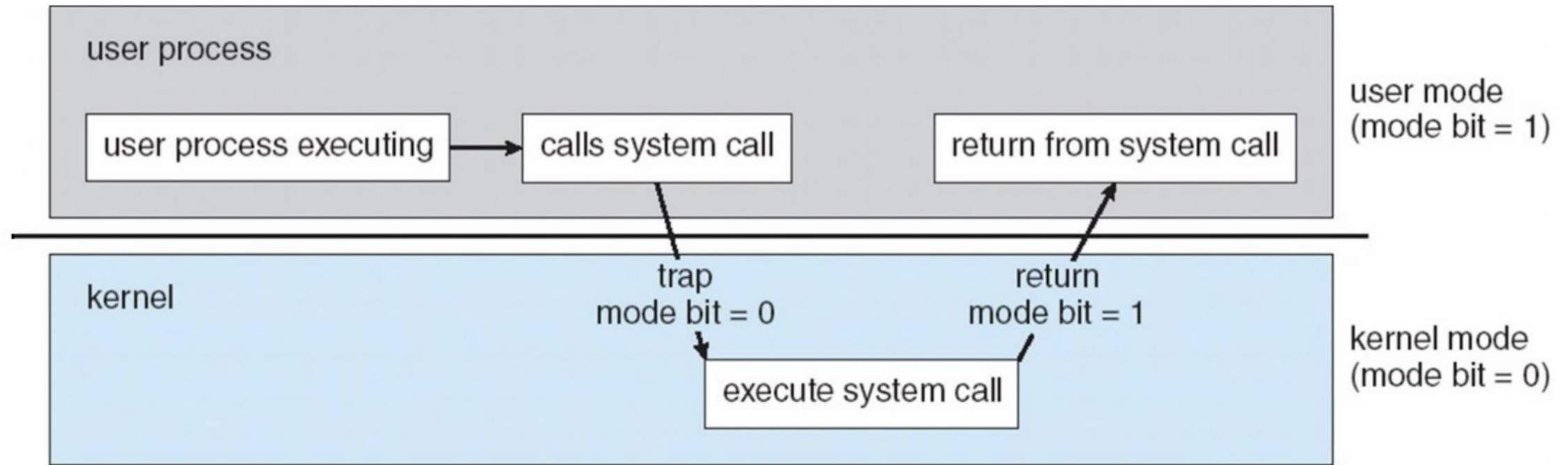
Useful Online Resources

- [The Linux Kernel Documentation](#): the extensive documents extracted from kernel source
- [Linux Weekly News \(LWN\)](#): easy explanation of recently added kernel features
- [Linux Inside](#): textbook-style description on kernel subsystems
- [Kernel newbies](#): useful information for new kernel developers
- [Linux Kernel API Manual](#)
- [Kernel Recipes](#)
- [Kernel Planet](#)

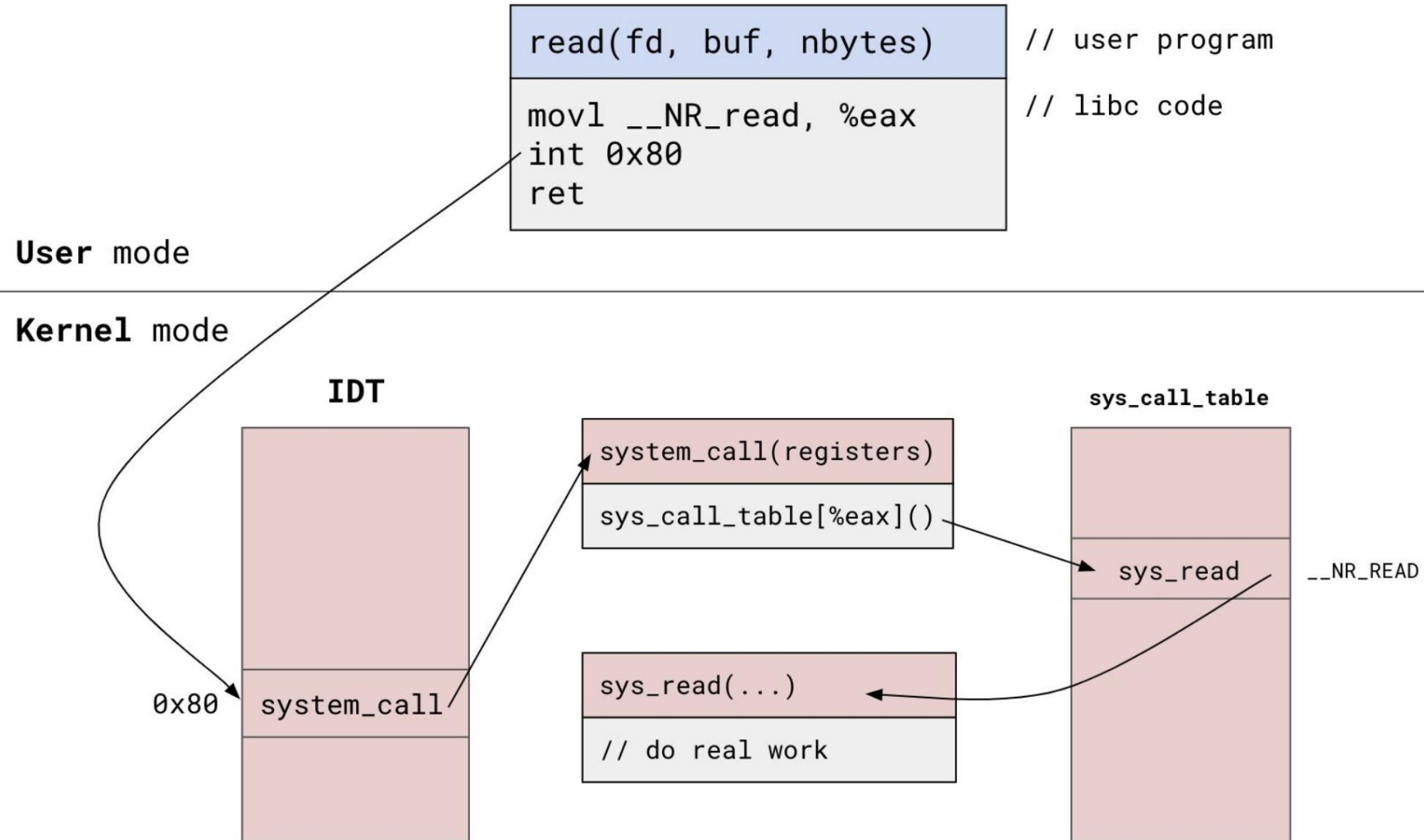
- Ring 0 protects everything relevant to isolation
 - writes to `%cs` (to defend CPL)
 - every memory read/write
 - I/O port accesses
 - control register accesses (`eflags`, `%fs`, `%gs`, ...)
- Controlled transfer: system call
 - `int`, `sysenter` or `syscall` -
 - *Q: How to systematically manage such interfaces?*



- One and the only way for user-space application to enter the kernel to request OS services and privileged operations such as accessing the hardware
 - A layer between the hardware and user-space processes
 - An abstract hardware interface for user-space
 - Ensure system security and stability



An Illustration with “int 0x80”



System Call Examples

- Process management/scheduling:
 - ``fork``, ``exit``, ``execve``, ``nice``, ``{get|set}priority``, ``{get|set}pid``
- Memory management:
 - ``brk``, ``mmap``
- File system:
 - ``open``, ``read``, ``write``, ``lseek``, ``stat``
- Inter-Process Communication:
 - ``pipe``, ``shmget``
- Time management:
 - ``{get|set}timeofday``
- Others:
 - ``{get|set}uid``, ``connect``
- Q: Where are system call implementations in Linux kernel?

Syscall Table and Identifier

- The syscall table for x86_64 architecture
 - ``linux/arch/x86/entry/syscalls/syscall_64.tbl``
- Syscall ID: unique integer ← sequentially assigned

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
0    common  read      sys_read
1    common  write     sys_write
2    common  open      sys_open
...
332  common  statx     sys_statx
```

“sys_call_table”

- `syscall_64.tbl` is translated to an array of function pointers, `sys_call_table`, upon kernel build
 - `linux/scripts/syscalltbl.sh`

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    [0] = sys_read,
    [1] = sys_write,
    [2] = sys_open,
    ...
    ...
    ...
};
```

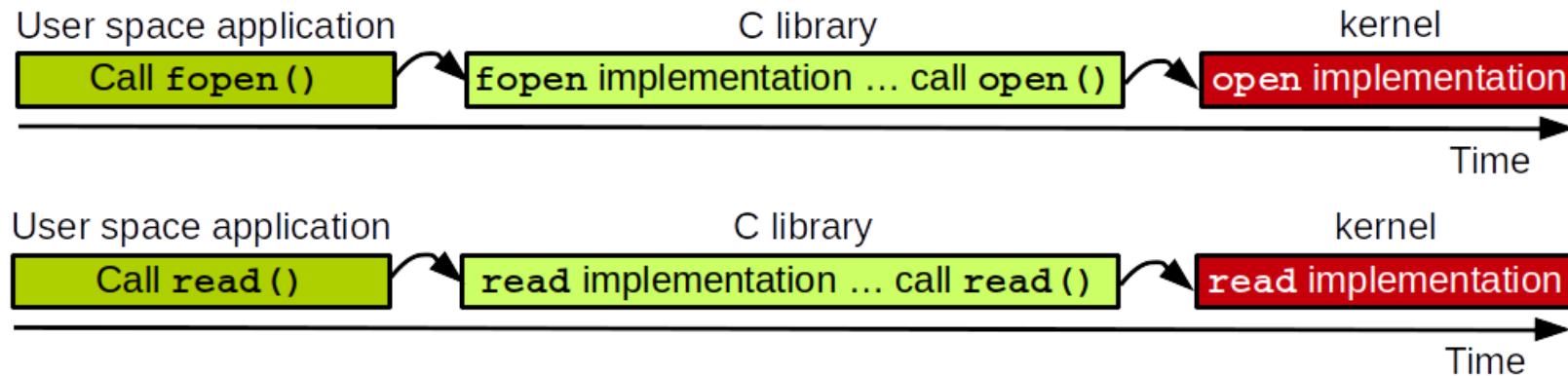
Syscall Implementation

```
# linux/arch/x86/entry/syscalls/syscall_64.tbl
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
0    common  read          sys_read
1    common  write         sys_write

/* linux/fs/read_write.c */
/* ssize_t write(int fd, const void *buf, size_t count); */
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
    return ksys_write(fd, buf, count);
}
```


Invoking a Syscall

- Syscalls are rarely invoked directly
 - Most of them are wrapped by the C library (libc, POSIX API)



Invoking a Syscall

- A syscall can be directly called through `syscall`
 - See “man syscall” → C library uses “syscall”

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

int main(void)
{
    char    msg[] = "Hello, world!\n";
    ssize_t bytes_written;

    /* ssize_t write(int fd, const void *msg, size_t count);    */
    bytes_written = syscall(1, 1, msg, 14);
    /*
       \  \
    /*
       \  +-- fd: standard output
    /*
       \  +-- write syscall id (or SYS_write) */
    return 0;
}
```

System Call Instructions

- x86 instruction for system call
 - “inte \$0x80” raise a software interrupt 128 (old)
 - “sysenter”: fast system call (x86_32)
 - “syscall”: fast system call (x86_64)
- Passing a syscall ID and parameters
 - syscall ID: “%rax”
 - parameters (x86_64)
 - » *passed via registers: rdi, rsi, rdx, r10, r8, r9*
 - » *Use struct pointer to pass in more/larger arguments (e.g., struct sigaction)*
 - » *Need to validate memory! Why?*
 - If a function has more than six arguments, other parameters will be placed on the stack

Invoking a Syscall

- x86_64 architecture has a “syscall” instruction

```
.data
```

```
msg:
```

```
    .ascii "Hello, world!\n"
```

```
    len = . - msg
```

```
.text
```

```
    .global _start
```

```
_start:
```

```
    mov    $1, %rax    # syscall id: write
    mov    $1, %rdi    # 1st arg: fd (standard output)
    mov    $msg, %rsi  # 2nd arg: msg
    mov    $len, %rdx  # 3rd arg: length of msg
    syscall           # switch from user space to kernel space

    mov    $60, %rax   # syscall id: exit
    xor    %rdi, %rdi   # 1st arg: 0
    syscall           # switch from user space to kernel space
```

Handling the Syscall Interrupt

- The kernel syscall interrupt handler, system call handler
 - “entry_SYSCALL_64 at linux/arch/x86/entry/entry_64.S
- “entry_SYSCALL_64” is registered at CPU initialization time
 - A handler of “syscall” is specified at a “IA32_LSTAR MSR” register
 - The address of “IA32_LSTAR MSR” is set to “entry_SYSCALL_64 at boot time: syscall_init() at linux/arch/x86/kernel/cpu/common.c

- “entry_SYSCALL_64” invokes the entry function for the syscall ID
 - call do_syscall_64
 - regs->ax = sys_call_table[nr](regs);

```
# linux/arch/x86/entry/syscalls/syscall_64.tbl
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
0   common  read      sys_read
1   common  write     sys_write
```

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    [0] = sys_read,
    [1] = sys_write,
```

Returning from the Syscall

- x86 instruction for system call
 - "iret": interrupt return (x86-32 bit, old)
 - "sysexit": fast return from fast system call (x86-32 bit)
 - "sysret": return from fast system call (x86-64 bit)

Syscall Example: gettimeofday()

- man gettimeofday

NAME

gettimeofday, settimeofday - get / set time

SYNOPSIS

```
#include <sys/time.h>
```

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

```
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

DESCRIPTION

The functions `gettimeofday()` and `settimeofday()` can get and the time as well as a timezone. The `tv` argument is a `struct timeval` (as specified in `<sys/time.h>`):

```
struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```


Example C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(void)
{
    struct timeval tv;
    int ret;

    ret = gettimeofday(&tv, NULL);
    if(ret == -1)
    {
        perror("gettimeofday");
        return EXIT_FAILURE;
    }

    printf("Local time:\n");
    printf("  sec:%lu\n", tv.tv_sec);
    printf("  usec:%lu\n", tv.tv_usec);

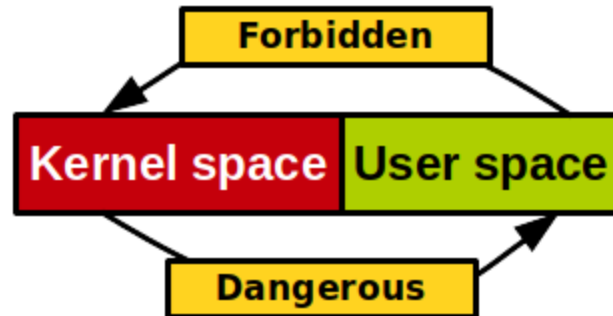
    return EXIT_SUCCESS;
}
```

Kernel Implementation

```
/* linux/kernel/time/time.c */
/* SYSCALL_DEFINE2: a macro to define a syscall with two parameters */
SYSCALL_DEFINE2(gettimeofday, struct timeval __user *, tv,
                struct timezone __user *, tz) /* __user: user-space address */
{
    if (likely(tv != NULL)) { /* likely: branch hint */
        struct timespec64 ts;

        ktime_get_real_ts64(&ts);
        if (put_user(ts.tv_sec, &tv->tv_sec) ||
            put_user(ts.tv_nsec / 1000, &tv->tv_usec))
            return -EFAULT;
    }
    if (unlikely(tz != NULL)) {
        /* memcpy to usr-space memory */
        if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
            return -EFAULT;
    }
    return 0;
}
```

User-space vs. kernel-space Memory



- User space cannot access kernel memory
- Kernel code must never blindly follow a pointer into user-space
 - Accessing incorrect user address can make kernel crash!
- Q: How to prevent a user-space access kernel-space memory?
- Q: How to safely access user-space memory?

copy_from_user() + copy_to_user()

```
/* copy user-space memory to kernel-space memory */  
static inline  
long copy_from_user(void *to, const void __user *from, unsigned long n);  
  
/* copy kernel-space memory to user-space memory */  
static inline  
long copy_to_user(void __user *to, const void *from, unsigned long n);
```

- Is the provided user-space memory legitimate?
 - If not, raise an illegal access error
- Does the user-space memory exist?
 - If swapped out, kernel accesses the user-space memory after swap-in so the process can sleep

Implementing a New System Call

- Write your syscall function
 - Add to the existing file or create a new file
 - Add your new file into the kernel Makefile
- Add it to the syscall table and assign an ID
 - `linux/arch/x86/entry/syscalls/syscall_64.tbl`
- Add its prototype in `linux/include/linux/syscalls.h`
- Compile, reboot, and run
 - Touching the syscall table will trigger the entire kernel compilation

- Example: syscall implemented in linux sources in linux/my_syscall/my_func.c
- Create a linux/my_syscall/Makefile
 - obj-y += my_func.o
- Add “my_syscall” in linux/Makefile
 - core-y += kernel/ certs/ mm/ fs/ my_syscall/

Why Not to Add a New Syscall

- **Pros**
 - Easy to implement and use, fast
- **Cons**
 - Need an official syscall number
 - Interface cannot change after implementation, need to maintain it forever
 - Must support every architecture
 - Probably too much work for small exchanges of information, easily an overkill!
- **Alternative**
 - Create a device node and “read()” / “write()”
 - Use “ioctl()”

Improving Syscall Performance

- System call performance is critical in many applications
 - Web server: `select()`, `poll()`
 - Game engine: `gettimeofday()`
- **Hardware: add a new fast system call instruction**
 - `int 0x80` → `syscall`
- **Software: vDSO (virtual dynamically linked shared object)**
 - A kernel mechanism for exporting a kernel space routines to user space applications
 - No context switching overhead
 - e.g., “`gettimeofday()`”
 - » the kernel allows the page containing the current time to be mapped read-only into user space
- **Software: FlexSC: Exception-less system call, OSDI 2010**
 - Optimizing system call performance for large multi-core systems
 - “FlexSC improves performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 105% while requiring no modifications to the applications

Readings

- LWN: Anatomy of a system call: [part 1](#) and [part 2](#)
- [LWN: On vsyscalls and the vDSO](#)
- [Linux Inside: system calls](#)
- [Linux Performance Analysis: New Tools and Old Secrets](#)