# Isolation and System Calls

Xiaoguang Wang

# Summary of the last lecture

Get, build, and explore the Linux kernel

- `git, tig, make, make modules, make modules_install, make install, vim, emacs, LXR, cscope, ctags, tmux`

Don’t try to master them at once. Instead, gradually get used to them.

# Questions

Other vim tools?

- vim-plug

```
git clone --depth=1 https://github.com/xgwang9/.vim.git ~/.vim
cd ~/.vim
sh install.sh
```

How do I read Linux kernel code?

# How to read kernel code

**E.g., ext4 file system**

1. General understanding of OS file systems ← any OS textbook
2. File system in Linux kernel
3. Check kernel Documentation and Ext4 on-disk layout
4. Read the ext4 kernel code

- module by module; start from a system call (e.g., how `sys_write()` is implemented?)

5. Search LWN to check the latest changes → E.g., ext4 encryption support

# How to read kernel code

Use function tracer

- [ftrace](): function tracer framework
- [perf tools](): `ftrace` front end
    - Try [kernel/funcgraph]()
- Try [bpftrace]()!

```
cs594-s24/kernel/perf-tools  master ✔
▶ sudo ./kernel/funcgraph -HtPp 22842 vfs_read
Tracing "vfs_read" for PID 22842... Ctrl-C to end.
# tracer: function_graph
#
#     TIME        CPU  TASK/PID         DURATION                  FUNCTION CALLS
#      |          |     |    |           |   |                     |   |   |   |
 2664.970859 |    2)  zsh-22842    |               |          finish_task_switch.isra.0() {
 2664.970862 |    2)  zsh-22842    |               |            raw_spin_rq_unlock() {
 2664.970863 |    2)  zsh-22842    |   0.431 us    |              _raw_spin_unlock();
 2664.970863 |    2)  zsh-22842    |   1.450 us    |            }
 2664.970865 |    2)  zsh-22842    |   0.525 us    |            irq_enter_rcu();
 2664.970866 |    2)  zsh-22842    |               |            __sysvec_irq_work() {
 2664.970866 |    2)  zsh-22842    |               |              __wake_up() {
 2664.970866 |    2)  zsh-22842    |               |                __wake_up_common_lock() {
 2664.970867 |    2)  zsh-22842    |   0.288 us    |                  _raw_spin_lock_irqsave();
 2664.970867 |    2)  zsh-22842    |               |                  __wake_up_common() {
 2664.970868 |    2)  zsh-22842    |               |                    autoremove_wake_function() {
 2664.970868 |    2)  zsh-22842    |               |                      default_wake_function() {
 2664.970869 |    2)  zsh-22842    |               |                        try_to_wake_up() {
 2664.970869 |    2)  zsh-22842    |   0.484 us    |                          _raw_spin_lock_irqsave();
 2664.970870 |    2)  zsh-22842    |               |                          select_task_rq_fair() {
 2664.970870 |    2)  zsh-22842    |   0.257 us    |                            __rcu_read_lock();
```

# Browse/navigate kernel code

```
$ make cscope tags -j2
# cscope – build cscope database; tags – build ctag database
$ vim
# :tag <symbol>         # search symbol definition
# :cs find s <symbol> # find uses of symbol
# Ctrl - ]              # search symbol definition at the cursor
# Ctrl – t              # return to the previous cursor point
```

# Today: isolation and system calls

How to isolate user applications from the kernel?

How to safely access the kernel from user application?

How does the Linux system call work?

# The unit of isolation: "process"

Prevent process X from accessing or spying on process Y

- e.g., memory, address space, FDs, cpu, etc.

Prevent a process from maliciously accessing the operating system itself

- e.g, a buggy or malicious program

How to isolate a process from kernel?

# Isolation mechanisms in OS

User/kernel mode flag (a.k.a., rings)

Address spaces (later)

Time-slicing (later)

System call interface

# Hardware isolation in x86

Ring 0: most privileged CPU mode – kernel

Ring 3: most unprivileged CPU mode -- user

**Q: What's the meaning of "rings" here?**

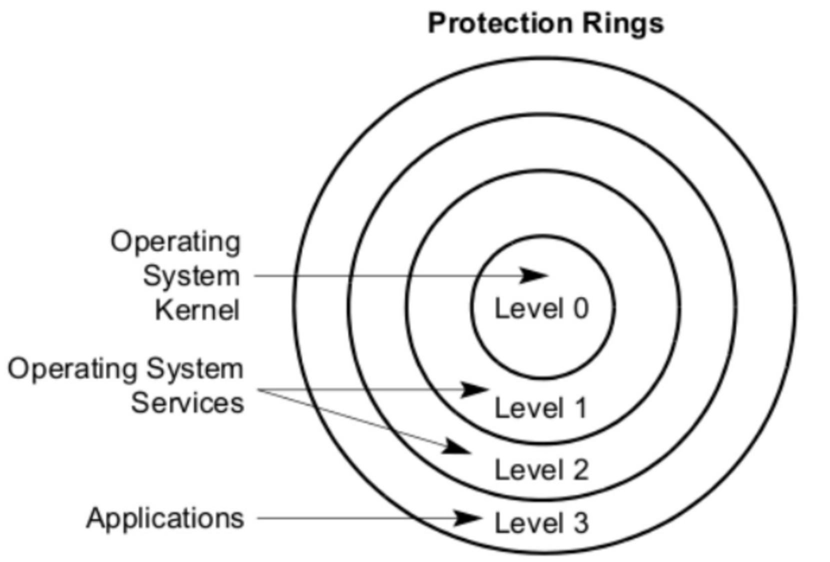



Figure 5-3. Protection Rings

# Segmentation in x86_64

Logical address:

- segment base + offset

Linear address:

- via page tables ➔ physical address

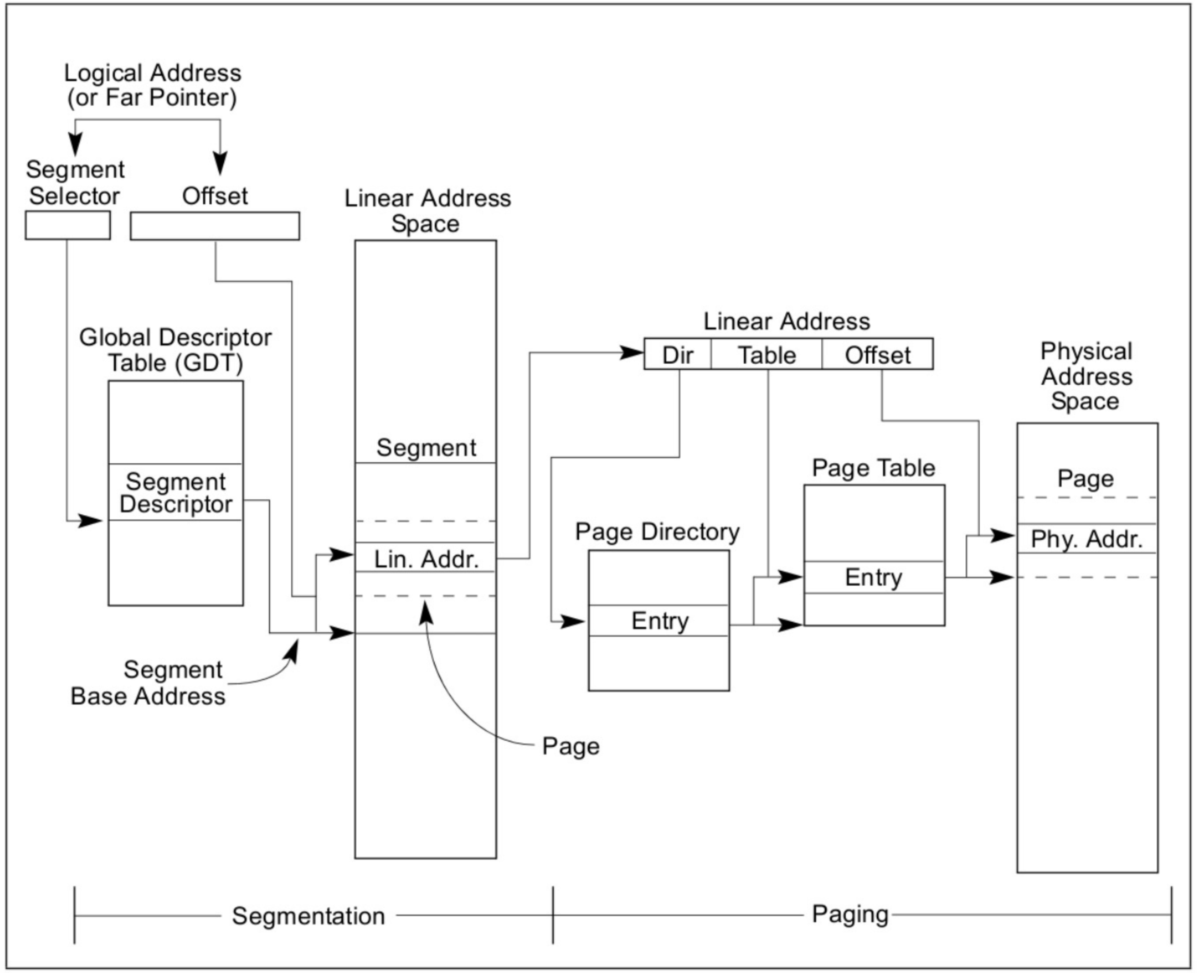


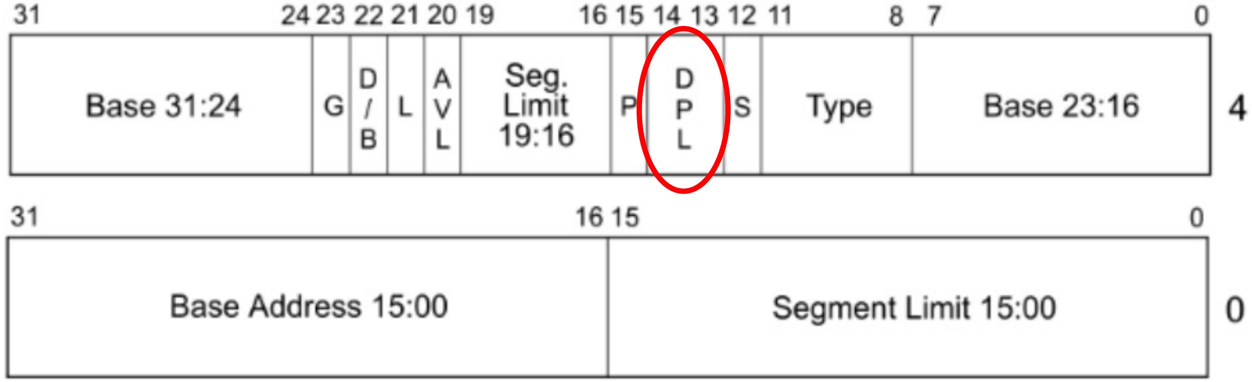

Figure 3-1. Segmentation and Paging

# Segmentation in x86_64

Protected/long mode:

- 16-bit segment registers store a selector
  - `%cs, %ds, %ss, %es, %fs, %gs`
- a selector contains an index to a segment descriptor table

**Segment descriptor (in GDT or LDT)**



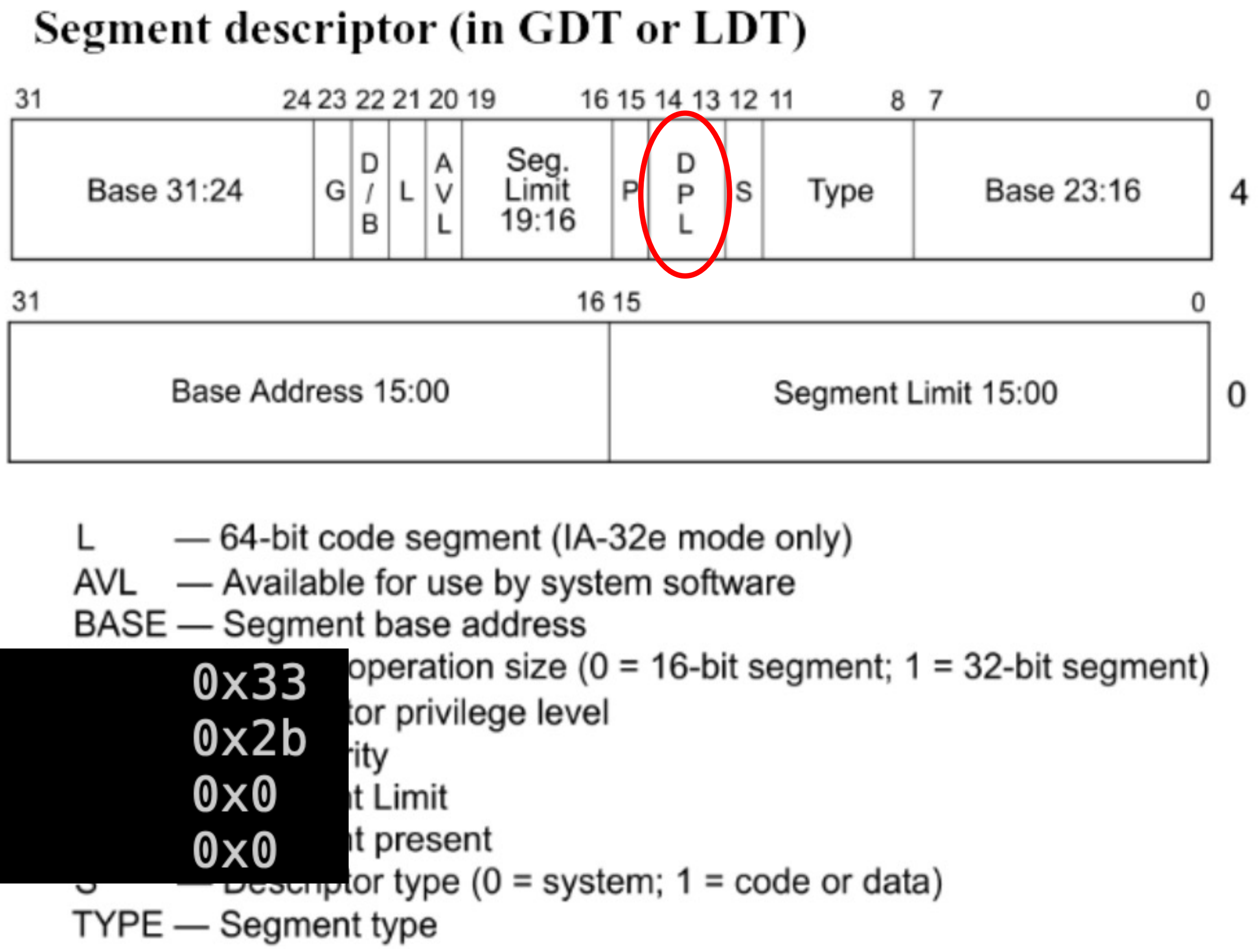

L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

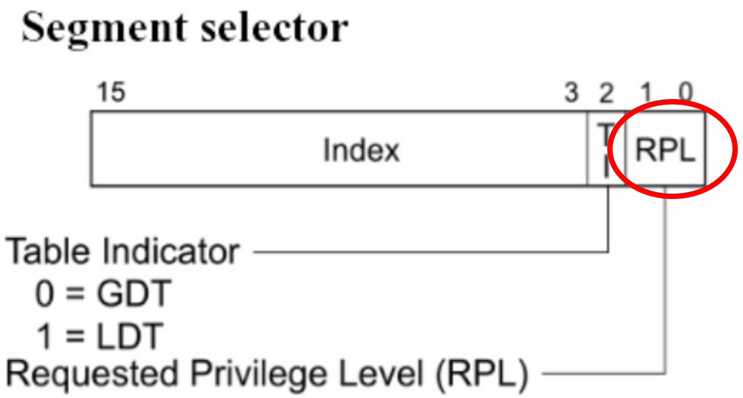
**Segment selector**

# Segmentation in x86_64

Protected/long mode:

- 16-bit segment registers store a selector
  - `%cs, %ds, %ss, %es, %fs, %gs`
- a selector contains an index to a segment descriptor table




```
cs    0x33
ss    0x2b
ds    0x0
es    0x0
```

# Privilege levels of a segment

DPL (descriptor privilege level)

- the privilege level of a segment

CPL (current privilege level)

- the privilege level of currently executing program
- bits 0:1 in the `%cs` register

RPL (requested privilege level)

- an override privilege level that is assigned to a segment selector
- a segment selector is a part (16-bit) of segment registers (e.g., `%ds`, `%fs`), which is an index of a segment descriptor and RPL

# How is isolation enforced in x86?

Access is granted if `max(CPL, RPL) ≤ DPL` (x86 segment)

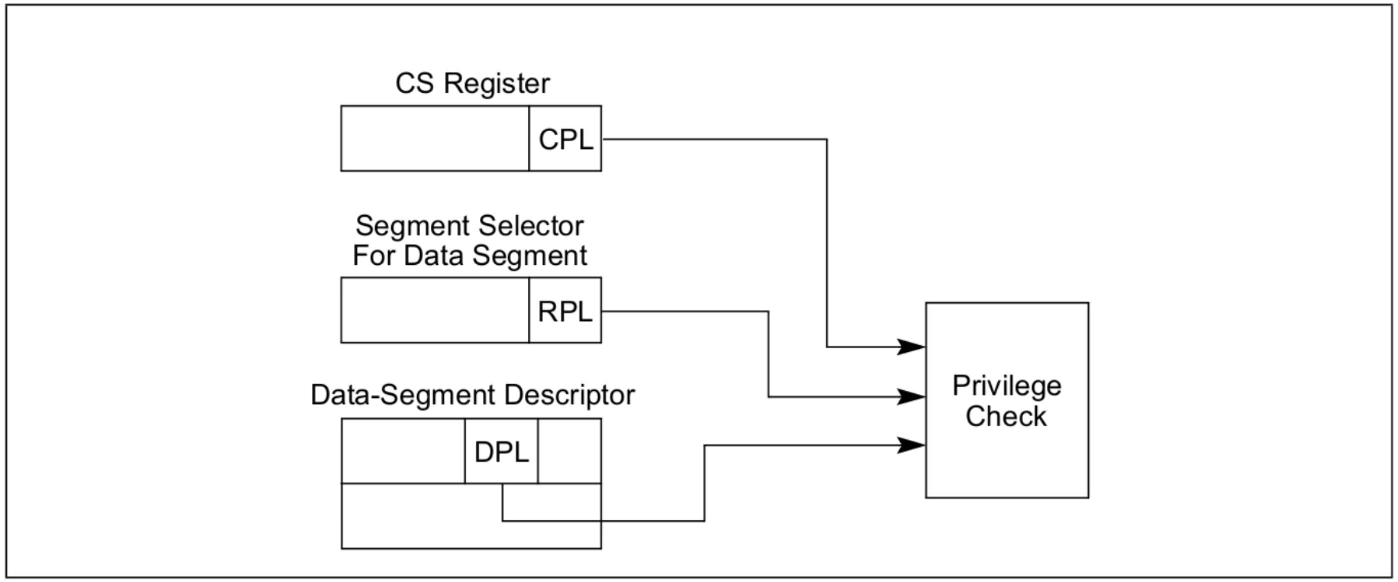



Figure 5-4. Privilege Check for Data Access

# What does "ring 0" protect?

Protects everything relevant to isolation

- Writes to `%cs` (to defend CPL)
- I/O port access
- Control register accesses (`eflags`, `%cr3`, …)

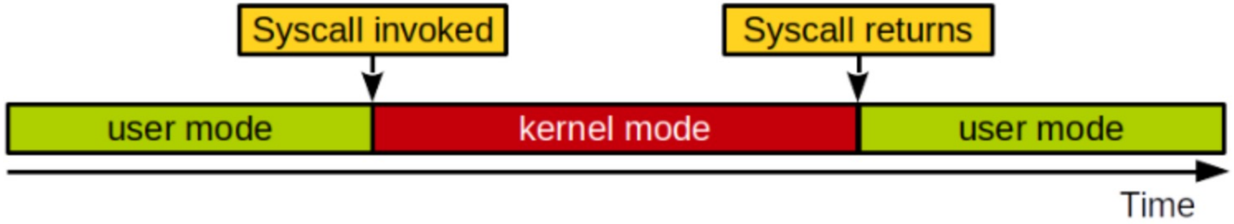
# How to switch between rings?

Controlled transfer: system calls

- `int 0x80`, `sysenter` or `syscall` instructions set CPL to 0; change to `KERNEL_CS` and `KERNEL_DS` segments
- set CPL to 3 before going back to user space; change to USER_CS and USER_DS segments

**Q: How to systematically manage this interface?**

# System calls




One way (the only way) for user-space application to enter the kernel to request OS services

- A layer between the hardware and the processes
- An abstract hardware interface for user-space
- Ensure system security and stability

# Examples of system calls

- 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 the x86_64 architecture

- `arch/x86/entry/syscalls/syscall_64.tbl`

Syscall ID: unique integer (sequentially assigned)

```
vi arch/x86/entry/syscalls/syscall_64.tbl

#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# 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
```

# sys_call_table

The `syscall_64.tbl` will be translated to an array of function pointers (`sys_call_table`) on kernel build

- `scripts/syscalltbl.sh`

```
/* arch/x86/entry/syscall_64.c */
asmlinkage const sys_call_ptr_t sys_call_table[] =
{
#include <asm/syscalls_64.h>
};
```

File `arch/x86/include/generated/asm/syscalls_64.h` will be generated after kernel build

# Syscall implementation

arch/x86/entry/syscalls/syscall_64.tbl

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# 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
```

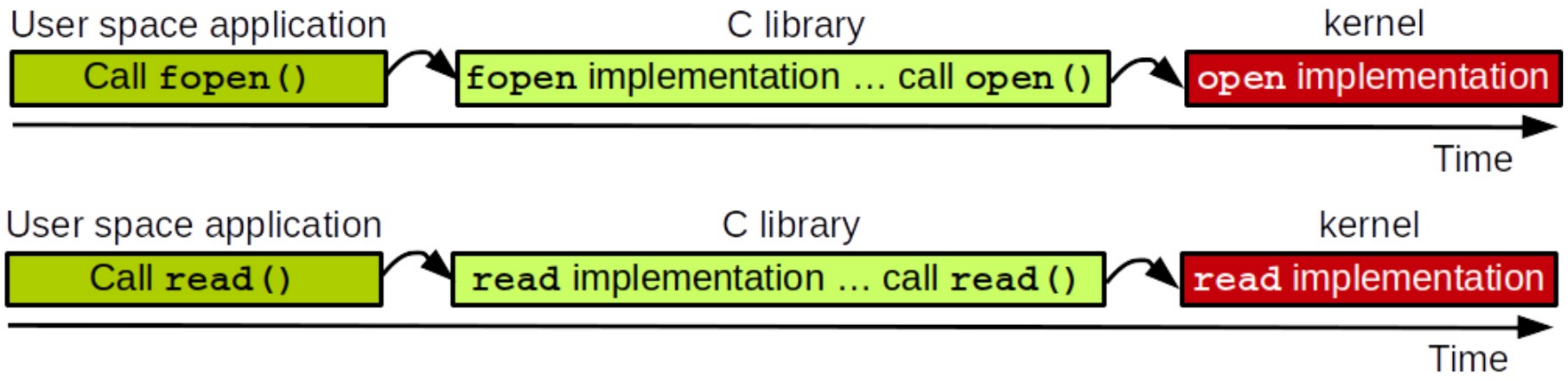
fs/read_write.c

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

# Invoke a syscall

Syscalls are rarely invoked directly

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



# Invoke a syscall

A syscall can be directly called through `syscall()`

- See man syscall → A C library function to directly call syscalls

```
#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

- `int 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): `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`
- If a function has more than six arguments, other parameters will be placed on the stack.

UIC COMPUTER SCIENCE

# Invoke a syscall

x86_64 architecture has a `syscall` instruction

```
▶ cat hello-asm.S
.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 `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 `arch/x86/kernel/cpu/common.c`

# Handling the syscall interrupt

`entry_SYSCALL_64` invokes the entry function for the syscall ID

- In `arch/x86/entry/entry_64.S`
- `call do_syscall_64`
- `regs->ax = sys_call_table[nr](regs);`

```
/* arch/x86/entry/syscall_64.c */
asmlinkage const sys_call_ptr_t sys_call_table[] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    [0] = sys_read,
    [1] = sys_write,
    ... ...
};
```

# Return 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

- Get the time

```
GETTIMEOFDAY(2)           Linux Programmer's Manual          GETTIMEOFDAY(2)

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);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       settimeofday():
           Since glibc 2.19:
               _DEFAULT_SOURCE
           Glibc 2.19 and earlier:
               _BSD_SOURCE

DESCRIPTION
       The functions gettimeofday() and settimeofday() can get and set 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

```c
#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;
}
```

time.c

# Kernel implementation

```
SYSCALL_DEFINE2(gettimeofday, struct __kernel_old_timeval __user *, tv,
                struct timezone __user *, tz)
{
        if (likely(tv != NULL)) {
                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)) {
                if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
                        return -EFAULT;
        }
        return 0;
}
```

kernel/time/time.c

# User-space vs. kernel-space memory

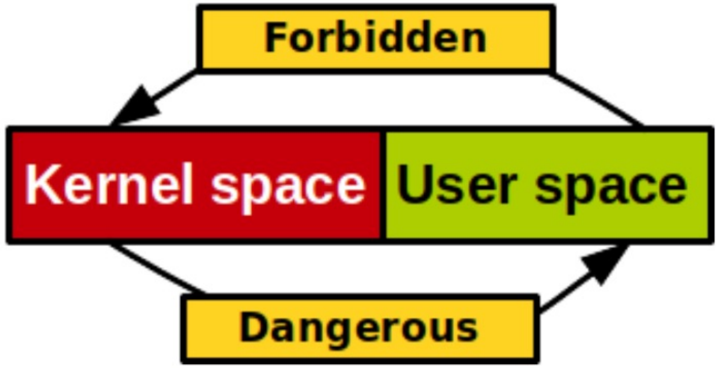
User space cannot access kernel memory

Kernel code must never blindly follow a pointer into user-space

- **Q: Why?**

**Q: How to prevent a user-space access kernel-space memory?**

**Q: How to safely access user-space memory?**



# copy_{from | to}_user

```c
/* 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);
```

Make sure the user-space memory is legitimate

- raise an error if not

… and exist

- wait for user-space memory to swap in

# Implement a new system call

1. Write your syscall function
   - Add to the existing file or create a new file
   - Add your new file into the kernel Makefile

2. Add it to the syscall table and assign an ID
   - `arch/x86/entry/syscalls/syscall_64.tbl`

3. Add its prototype in `include/linux/syscalls.h`

4. Compile, reboot, and run
   - Touching the syscall table will trigger the entire kernel compilation

# Implement a new system call

Example: syscall implemented in linux sources in 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/ ipc/ security/ crypto/ block/ my_syscall/`

# Why *not* add a new syscall

**Pros**: Easy to implement and use, fast

**Cons**:

- Needs an official syscall number
- Interface cannot change after implementation
- Must be registered for each architecture
- Probably too much work for small exchanges of information

**Alternative**:

- Create a device node and `read()` and `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

# Improving syscall performance

**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

# Summary

Isolation: CPU privilege on the x86 architecture

System calls: interface for applications to request OS services

Linux system calls: syscall table, syscall handler, and add a syscall

Improve syscall performance

# Next steps

Paper reading assignment 1 (due this Friday)

- FlexSC: Exception-less system call, OSDI 2010
  - Summary of the paper; do I like it, or dislike it? Why?
  - Strength of the paper
  - Weakness of the proposed approach
  - Questions/comments if any
- Tips to read a research paper:
  - Watch the presentation video first (if available)
  - Read abstract, introduction and evaluation first; read design/implementation with questions in your mind

# Next steps

Hw2 due this Friday

- Linux kernel compilation and boot in a QEMU VM

Hw3 is released (due Jan 26th)

- Modify the Linux kernel (~2 hours)
- On top of the QEMU VM from hw2

# Next lecture

FlexSC: Exception-less system call

- Optimizing system call performance on multi-core systems
- “We show how 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.”

Kernel data structures

# Further reading

LWN: Anatomy of a system call: part 1 and part 2

LWN: On vsyscalls and the vDSO

# Feedback