CS 477
Advanced Operating System

Lecture 02: Isolation & System Call

General advice about the course

Guidelines for interaction

- Please drop an email to me or use ED for all discussions
  - No direct email to TAs (let's use ED efficiently)
- Lecture discussion
  - Ask any question on ED or in class
- Labs discussion
  - Use only ED
  - Check all previous threads from ED, else it will be marked as a duplicate
  - You also need to do an online search to figure out the issues, if there are any

Self assessment quiz

- Quiz on the basics of OS fundamentals and C programming
- Not graded
- An hour long

Ways to read kernel code

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

1. General understanding of file systems in OS → any OS text book
2. File system in Linux kernel ← Check Linux programming book
3. Check kernel documentation and Ext4 on-disk layout
4. Read the ext4 kernel code
   - Module by module (e.g., dir, file, block management)
   - Start from a system call (e.g., how **`write()`** is implemented?)

Ways to read kernel code

- **Use function tracer**
- ftrace: function tracer framework
- perf tools: ftrace front end
- kernel/funcgraph
  - Trace a graph of kernel function calls, showing children and times
- bpftrace

From philosophy to implementation

- Last week: Monolithic vs. Microkernel debate
- Linux chose practicality over elegant design
- Consequence: Everything in Ring 0
- **Q.** How to safely trust the boundary?

- The challenge:
  - Monolithic = Fast but dangerous (one bug → kernel panics)
  - User/kernel boundary is the ONLY protection
  - This lecture: How to go about implementing this critical boundary

Focus of today's lecture

- Isolating user applications from the kernel
  - Process: a machine abstraction
  - Isolation mechanisms: X86 rings
- System calls: Safely access the kernel from a user application
  - Implementation details

Famous kernel vulnerabilities from missing checks

- Dirty COW (2016): Race condition in copy-on-write
  - Result: privilege escalation in Android/Linux
- Stack clash (2017): Stack overflow → kernel memory
  - Result: Remote code execution
- Meltdown (2018): CPU speculation breaks isolation
  - Result: User can read kernel memory

Key insight: One mistake in isolation = Game Over!

- In microkernel: Crash one server
- In monolithic (Linux): Crash/compromise everything

The Isolation Challenge

The unit of isolation: "Process"

- Prevent process **X** from wrecking or spying on **Y**
  - (e.g., memory, CPU, FDs, resource exhaustion)

- Prevent a process from wrecking the OS kernel
  - (i.e., from preventing the kernel from enforcing isolation)

- In the face of a buggy or malicious program
  - (e.g., a bad process may try to trick the h/w or kernel)

Q. How to isolate a process from the kernel execution?

Isolation mechanisms in OS

1. Isolate execution context: **User/kernel mode flag (aka ring)**
2. Safe user ←→ kernel transition**: System call interface**
3. Isolate memory: **Address space** (later)
4. Ensure scheduling of tasks: **Timeslicing** (later)

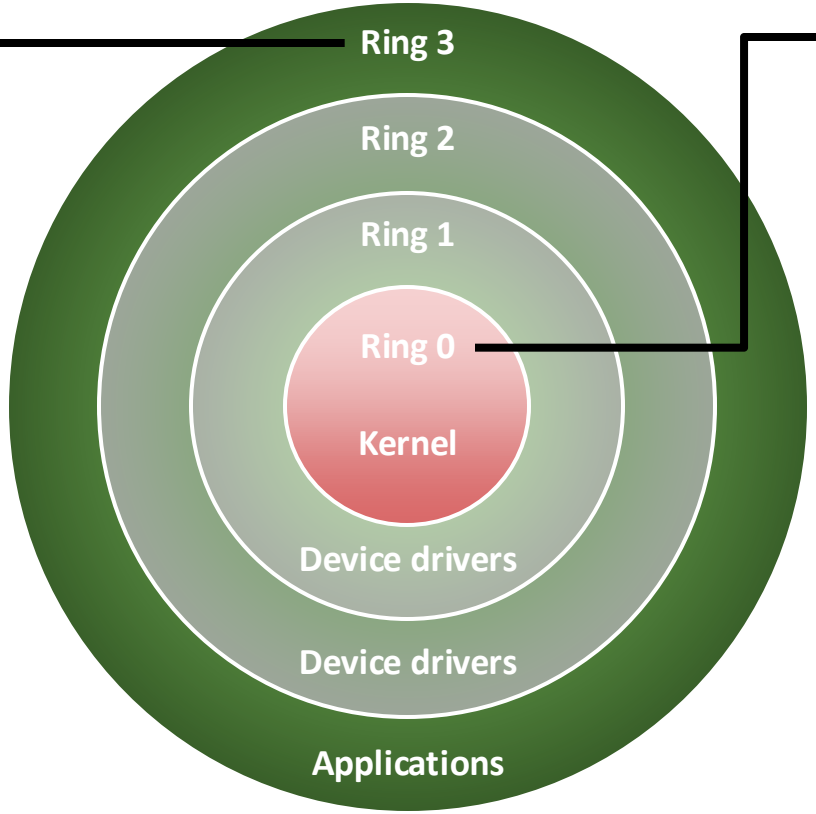
Hardware-Enforced Isolation

Hardware isolation in X86

Uses protection rings to distinguish among various modes of execution

User mode

Least privileged (Ring 3)



Kernel mode

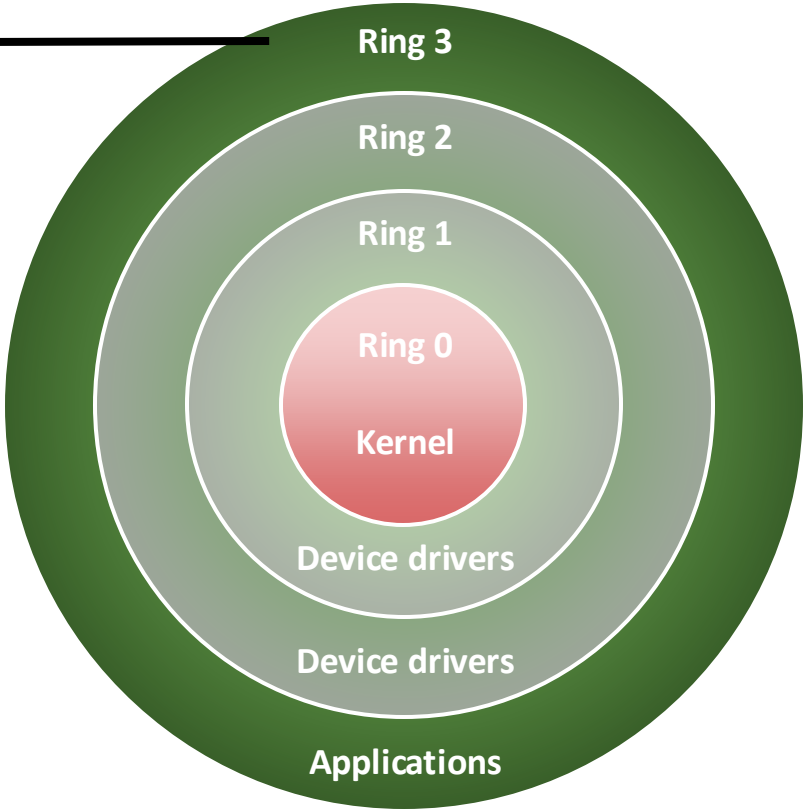
Most privileged (Ring 0)

Hardware isolation in X86

Uses protection rings to distinguish among various modes of execution

User mode

Least privileged (Ring 3)



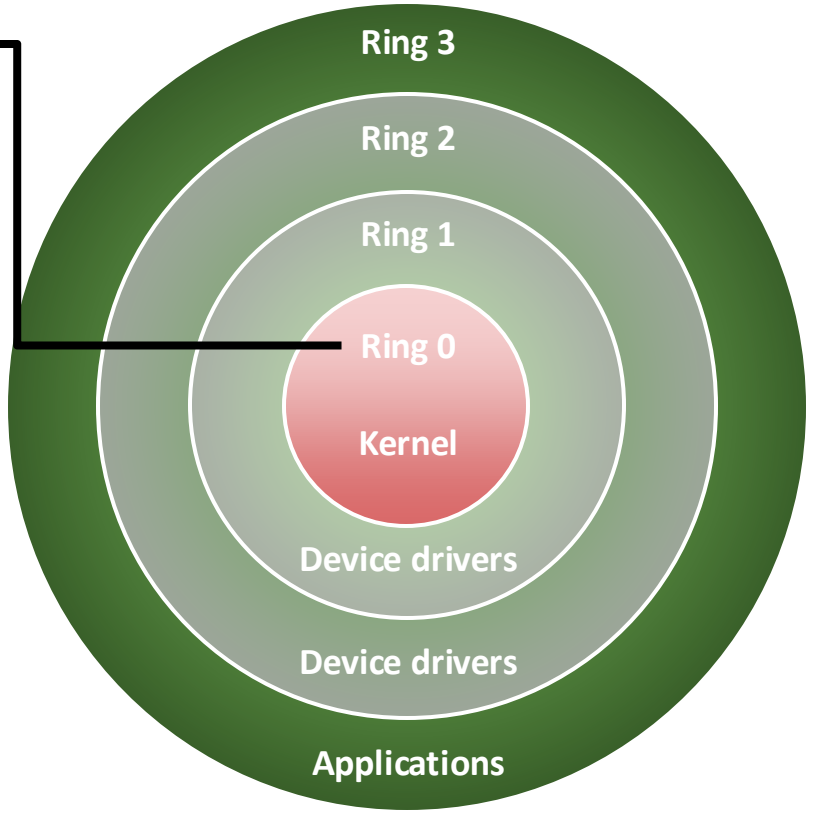
- CPU checks each instruction before executing it
- Executes a limited set of instructions – only allowed instructions
  - Example: no IO requests are allowed!

Hardware isolation in X86

Uses protection rings to distinguish among various modes of execution

Kernel mode

Most privileged (Ring 0)



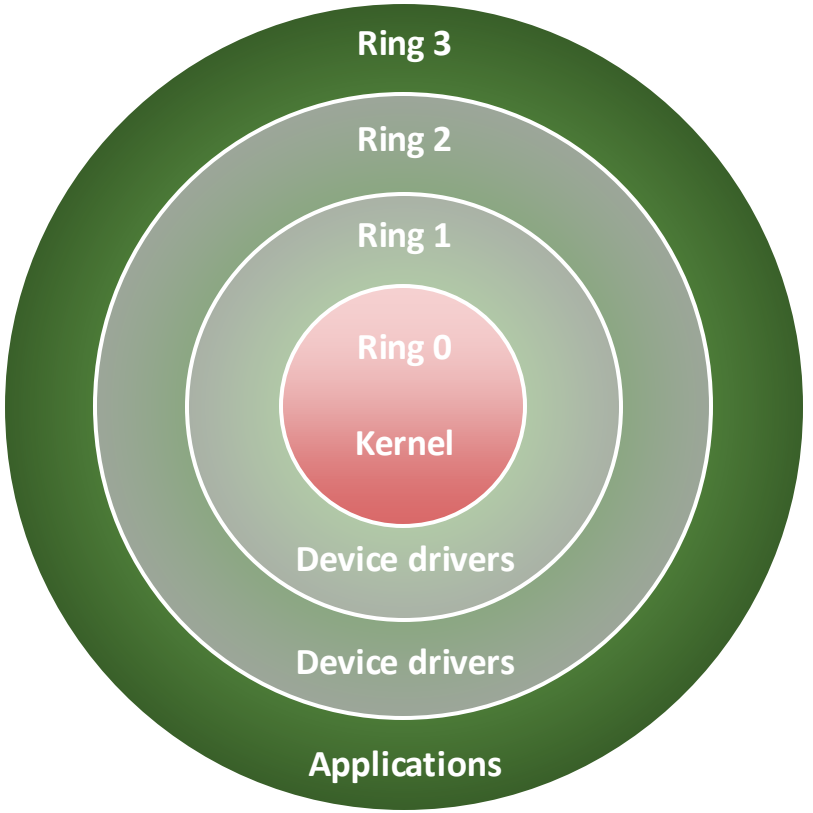
- CPU executes instructions without any checks
- Can execute all instructions
  - Example: all IO requests are allowed!

The hidden rings

- Modern X86: More than 4 rings
  - Ring 3: Application (least privilege)
  - Ring 2: [unused] Originally for drivers
  - Ring 1: [unused] Originally for drivers
  - Ring 0: Kernel (most privilege)

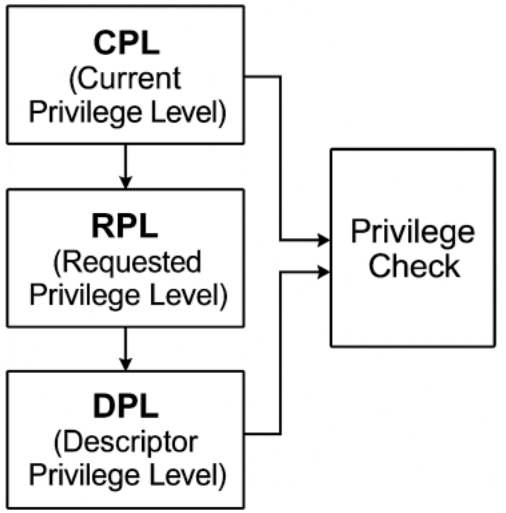
------ [What Intel doesn't advertise] ------

  - Ring -1: Hypervisor (VMX root mode)
  - Ring -2: System management mode (SMM)
  - Ring -3: Management engine (ME)



Q. Why does Linux only use Rings 0 and 3?

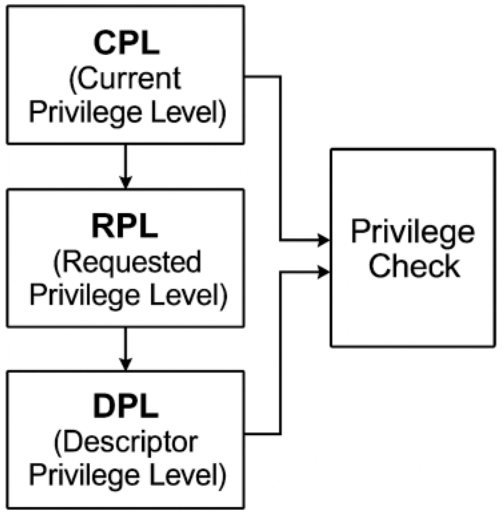
X86_64 segmentation stores privilege levels



- Segmentation is mostly disabled in 64-bit mode, but:
  - Still used for **privilege checks** and **thread-local storage (fs, gs)**

- 3 privilege notions:
  - CPL (current privilege level)
    - Comes from %cs (code segment) → CPL is 0 for kernel and 3 for user mode
  - DPL (descriptor privilege level)
    - Privilege required to access a segment, set in the GDT/LDT entry
    - User segments: DPL is 3; for the kernel segment, DPL is 0
  - RPL (requested privilege level)
    - Encoded in the last two bits of the segment selector, Linux sets RPL to CPL

Enforcing isolation on X86 hardware



- Access is granted if **`RPL >= CPL and DPL >= CPL`**
- **In Linux, `RPL = CPL`**

Q. What are the CPL, RPL, and DPL in the following assembly code?

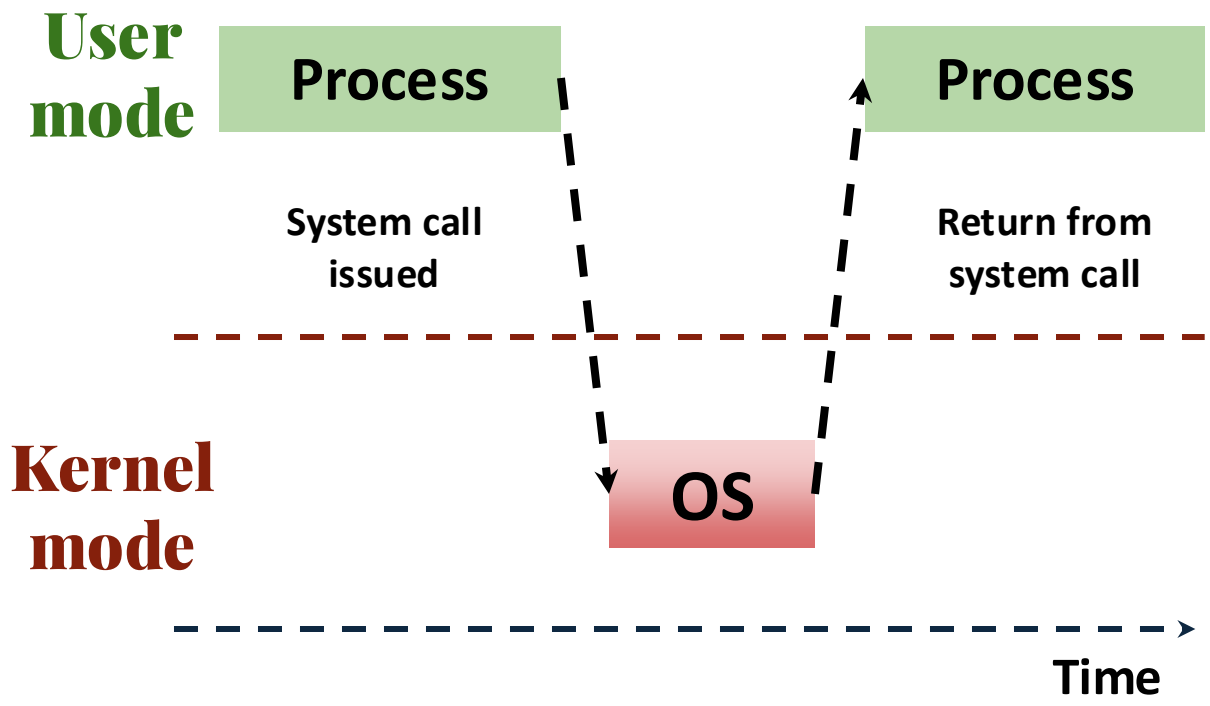
```
mov $42, %fs:(%eax) /* fs[eax] = 42; */
```

What does “ring 0” protect?

- Protects everything relevant to isolation
  - Writes to %cs (to defend CPL)
  - Every memory read/write
  - I/O port access
  - control register accesses (eflags, %fs, %gs, …)

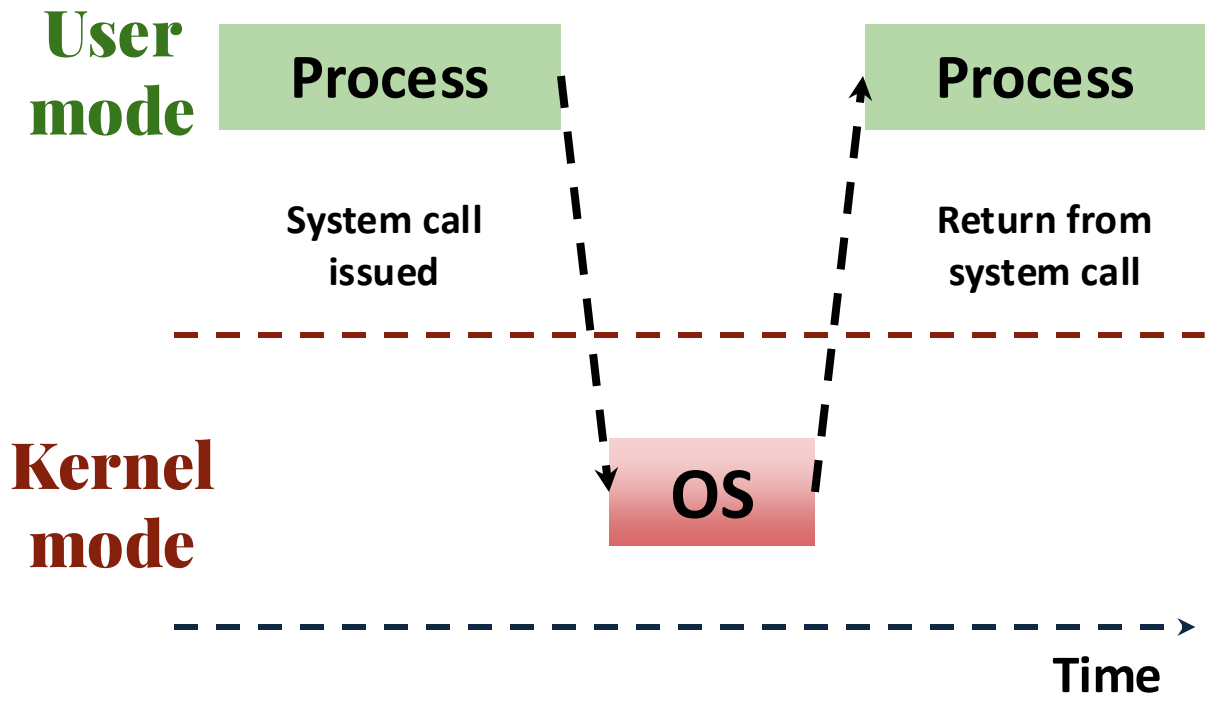
How to switch between rings?

- Controlled transfer: **system call**
- `int`, `sysenter`, or `syscall` instruction 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



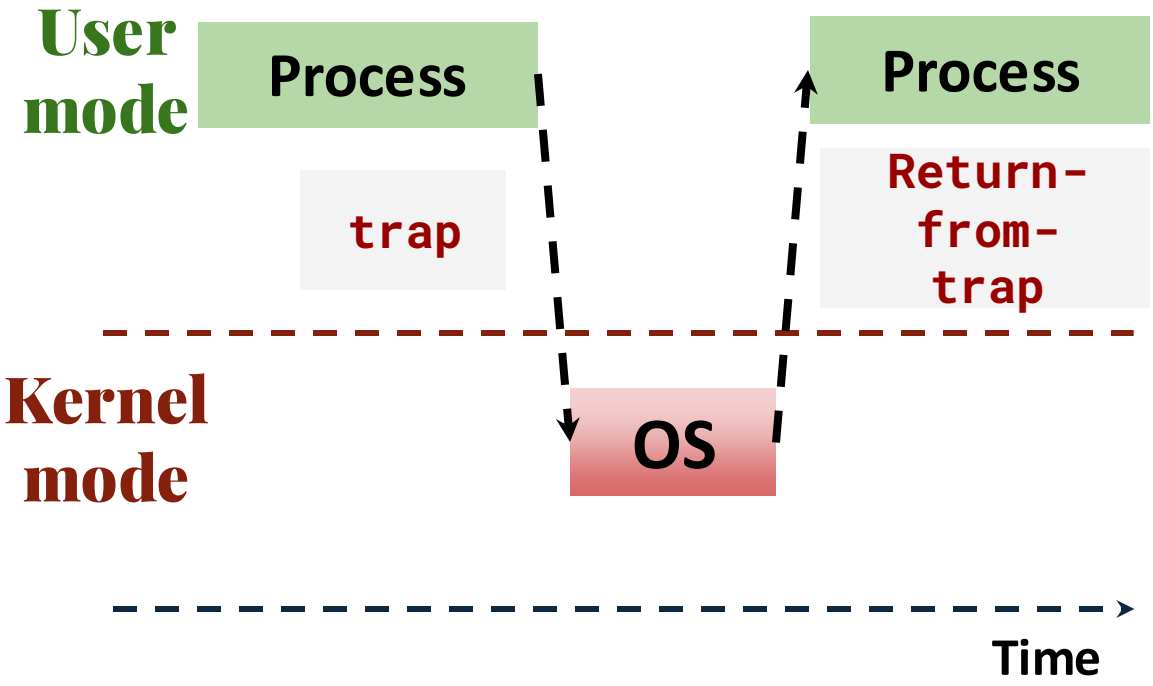
System call

- Process requests for OS services to perform privileged operations
- A layer between hardware and user-space processes
- An abstract hardware interface for user space
- Ensure system security and stability



Putting everything together for a system call

1. A system call is a **trap instruction**
2. OS saves registers to a **per-process/thread stack**
3. Change mode from Ring 3 to Ring 0
4. Execute privileged operations
5. Change mode from Ring 0 to Ring 3
6. Restore the state of the process by popping registers on the **return from the trap**



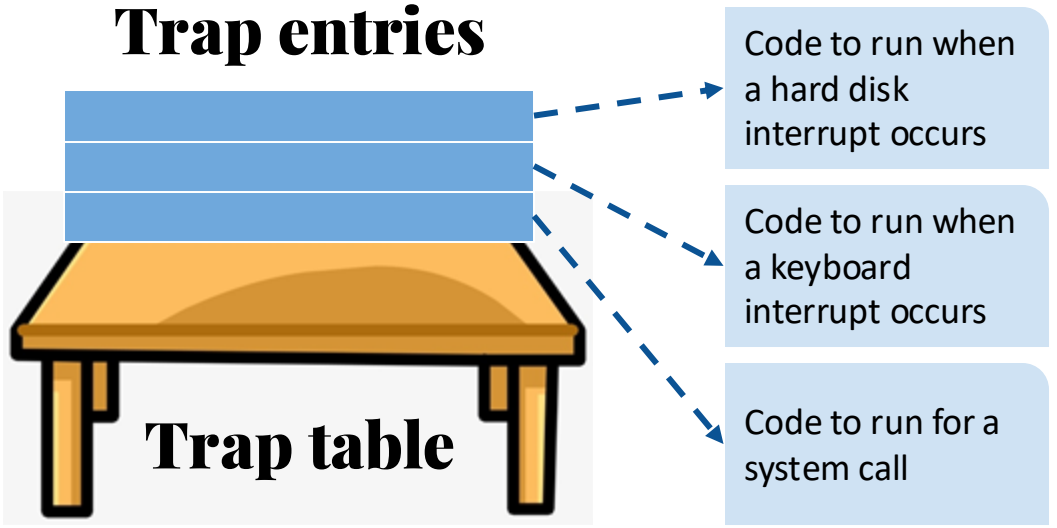
Question

How does a trap know which syscall-specific code to run in the kernel?

OS configures hardware at boot time

During boot …

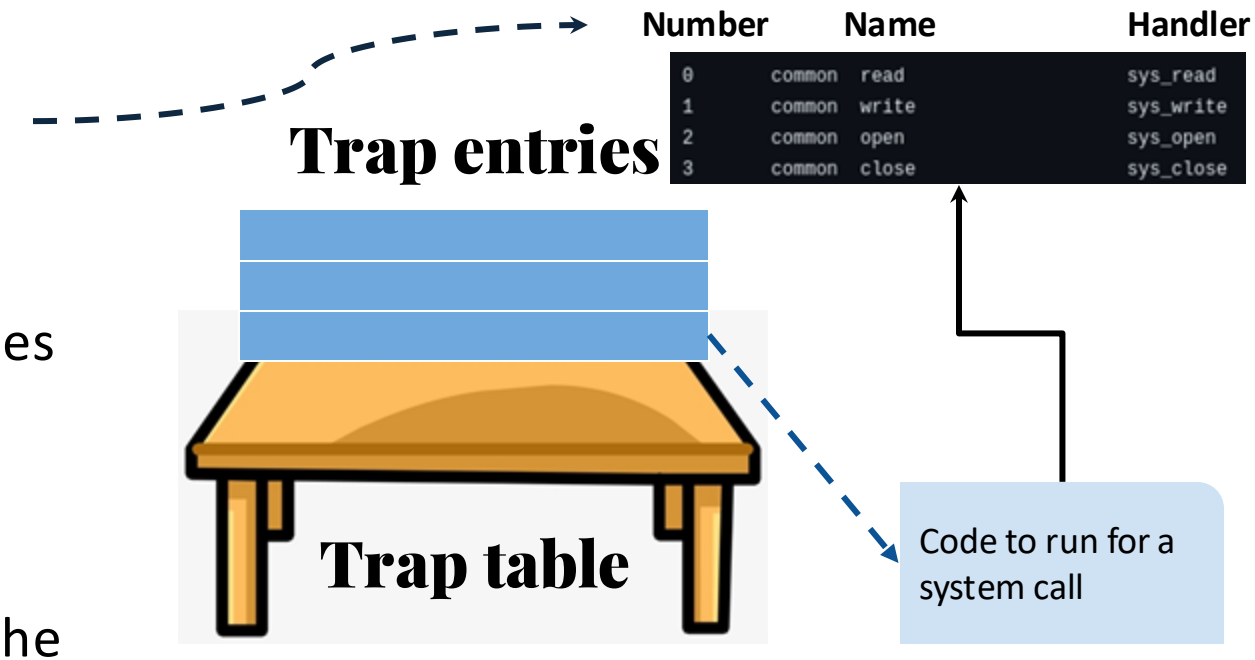
- The OS tells hardware what code to run when certain exceptional events occur
- OS configures specific handlers that hardware remembers
- Hardware then know what to do when certain exceptional events occur
  - System call



Requesting OS services using system call numbers

Only one handler routine for system call, but multiple system calls are possible!

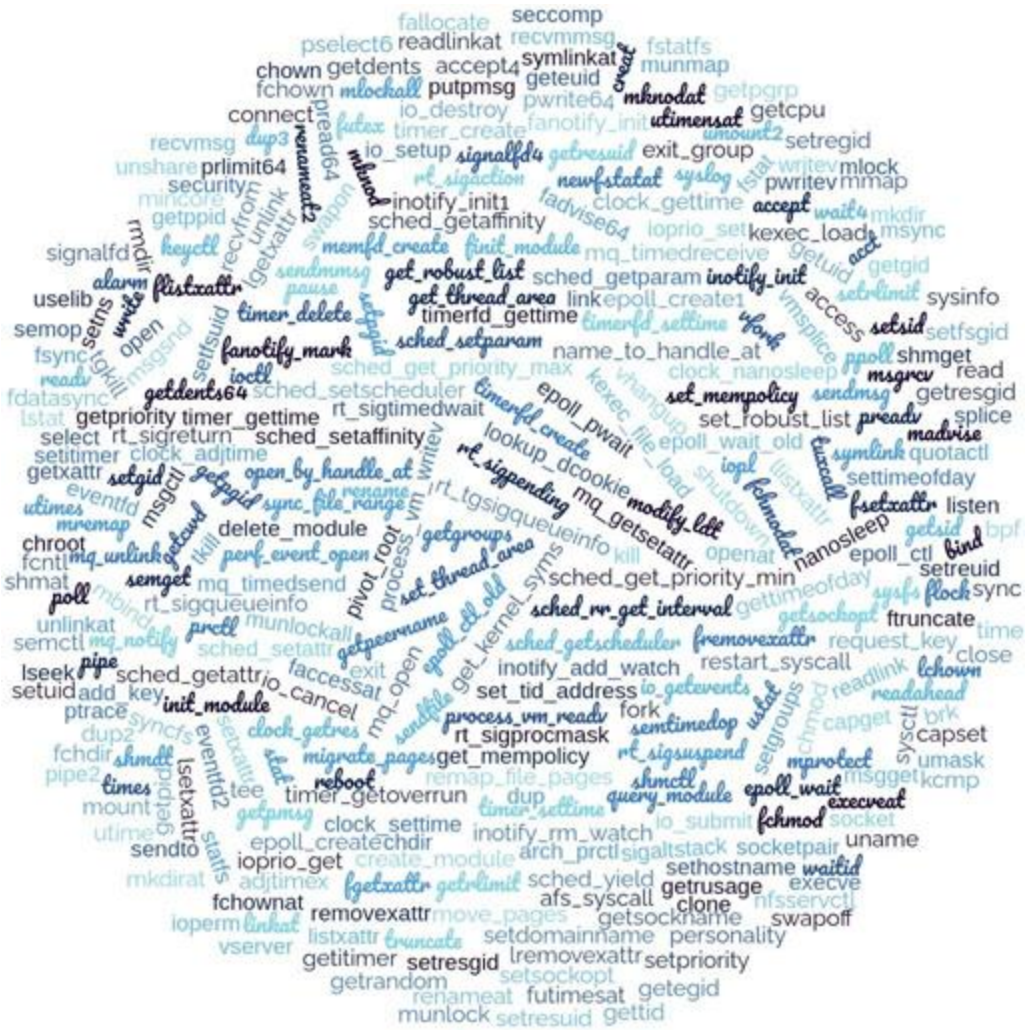
- Each system call has a specific number
- To issue a system call, the process specifies the number in the **stack** or a **register**
- During the trap, the kernel code checks the validity of the number and executes the corresponding system call code



400+ system calls in the Linux kernel

- 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. How to implement syscalls?



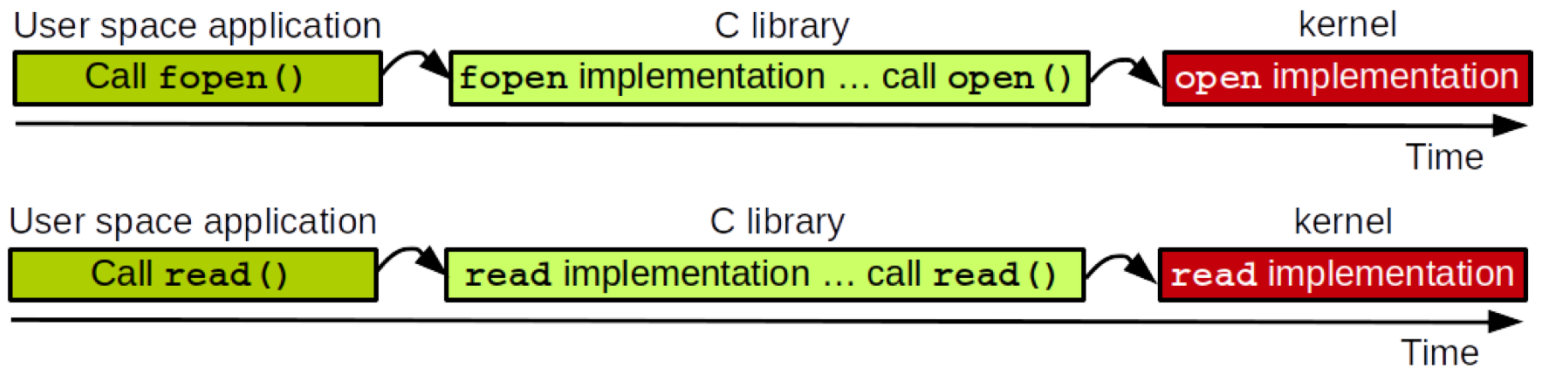
Linux Implementation (via `gettimeofday()`)

System call invocation 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`**
  - return address: **`%rcx`**
  - If a function has more than six arguments, other parameters are placed on the stack

Invoking a system call

- System calls are rarely invoked directly
  - Most of them are wrapped by the C library (**`libc`**, POSIX API)



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

```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;
}
```

Boot-time wiring (handling the syscall interrupt)

- The kernel syscall interrupt handler calls
  - **`entry_SYSCALL_64`** in `arch/x86/entry/entry_64.S`
- **`entry_syscall_64`** is registered for each CPU initialization point
  - **`cpu_init()` → `syscall_init()`**
  - **`wrmsrq(MSR_LSTAR, (unsigned long)entry_SYSCALL_64)`** in **`arch/x86/kernel/cpu/common.c`**

→ **`MSR_LSTAR`** value points to the **`entry_SYSCALL_64`** function address

→ This value is assigned at the CPU initialization time

Dispatching syscalls (handling the syscall interrupt)

- **`entry_syscall_64`** invokes the entry function for the syscall ID
  - **`call do_syscall_64 (arch/x86/entry/syscall_64.c)`**
  - **`regs->ax = x64_sys_call(regs);`**

- Syscall table maps syscall numbers to handler wrappers (defined in **`arch/x86/include/generated/asm/syscalls_64.h`**)
  - Automatically generated when compiling

- Per syscall wrapper stubs decode relevant args from **`struct pt_regs`** that calls **`__do_sys##NAME`** implementation

Where syscalls live (syscall table & ID)

- Syscall table for x86_64 architecture

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

- Syscall ID: unique identifier → 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
```

Building syscall table in Linux: `sys_call_table`

- **`syscall_64.tbl`** is translated to an array of function pointers
- **`sys_call_table`**, upon kernel build
  - **`scripts/syscalltbl.sh`**
  - It generates **`arch/x86/include/generated/asm/syscalls_64.h`**

```
__SYSCALL(0, sys_read)
__SYSCALL(1, sys_write)
__SYSCALL(2, sys_open)
__SYSCALL(3, sys_close)
__SYSCALL(4, sys_newstat)
__SYSCALL(5, sys_newfstat)
__SYSCALL(6, sys_newlstat)
__SYSCALL(7, sys_poll)
__SYSCALL(8, sys_lseek)
__SYSCALL(9, sys_mmap)
```

Kernel implementation of `gettimeofday()`

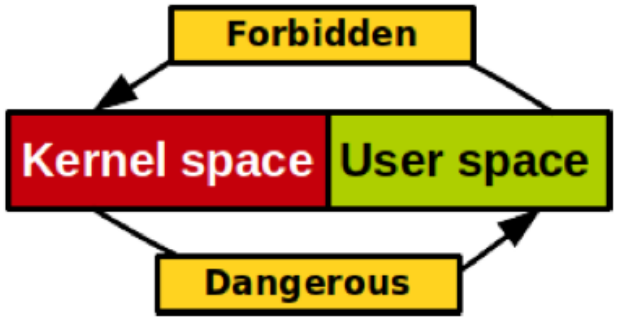
```
SYSCALL_DEFINE2(gettimeofday, struct __kernel_old_timeval __user *, tv,
                struct timezone __user *, tz)
{
        if (likely(tv != NULL)) {
                struct timespec64 ts;

                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;
}
```

Returning from the syscall

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

Security Challenges and Performance Impact

User space vs. kernel space memory



- User space cannot access kernel memory
- Kernel code must never blindly follow a pointer into user space
  - Accessing the incorrect user address can make the kernel crash

Q. How to prevent a user space from accessing kernel-space memory?

Q. How to safely access user-space memory?

Accessing 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);
```

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

The TOCTOU problem

Time-of-check vs. Time-of-use

Broken code

```
if (tv != NULL) { // CHECK: tv is valid
  // ⚠ ATTACKER: Change tv to kernel address!
  tv->tv_sec = seconds; // USE: Writes to kernel!
}
```

Correct code

```
if (tv != NULL) {
  struct timeval safe_tv;
  safe_tv.tv_sec = seconds;
  safe_tv.tv_usec = useconds;
  copy_to_user(tv, &safe_tv, sizeof(safe_tv));
  // copy_to_user checks AGAIN at USE time
}
```

Real exploit: Dirty COW used exactly this race!

Modern CPU security features

- NX bit (2003): No execute – separate code/data
- SMEP (2012): Prevents kernel (NULL pointer) from executing user code
- SMAP (2014): Prevents kernel from read/write to user memory
- CET (2020): Protects from ROP/JOP attacks

Each adds overhead to system calls:

- NX: Must mark pages correctly
- SMEP: Check on every control transfer
- SMAP: Must use copy_to_user (can't optimize!)
- CET: Shadow stack operations

The trend: Security > Performance

Meltdown: When hardware breaks software promises

- The attack (simplified):
  - User code: mov rax, [kernel address] // Will fault
  - CPU: Speculatively executes anyway before checking permissions
    - Fetches kernel data into registers and caches
    - **Side-effect:** Only the microarchitectural state is cleared; the caches remain intact
  - User: Use rax to affect cache
  - User: Time cache access → Read kernel memory!
    - Which cache lines are fast to access, the attacker can infer the kernel value

Meltdown fix: KPTI

- Kernel page table isolation
- Before: User and kernel share page table
- After: Complete separation, swap on every syscall

- Impact on **`gettimeofday`**:
- Before KPTI: 100ns
- After KPTI: 150ns (+50%!)
- Database benchmark: -30% throughput
- Redis: -20% operations/sec

Measuring the ideal cost: trace `gettimeofday()`

- User space:
  - Call `gettimeofday()` [~5 cycles]
  - glibc wrapper prepares registers [~10 cycles]
  - Execute '`syscall`' instruction [~100 cycles]
- Kernel space:
  - `entry_SYSCALL_64` saves context [~20 cycles]
  - `do_syscall_64` dispatches [~15 cycles]
  - `sys_gettimeofday` executes [~30 cycles]
  - `copy_to_user` (with checks!) [~25 cycles]
  - Return path restores context [~20 cycles]
  - '`sysret`' back to user [~50 cycles]
- Total: ~275 cycles = 100ns @ 2.5GHz

Handling syscall expanding attack surface

- Every syscall is an attack vector
  - sendfile (2005): Local root exploit
  - vmsplice (2008): Kernel memory disclosure
  - perf_event (2013): Privilege escalation
  - io_uring (2022): Multiple escapes

Pattern: Complex syscalls = More bugs

Solution: Minimize syscall interface

Problem: Users want features

Handling syscall expanding attack surface

Pattern: Complex syscalls = More bugs

Solution: Minimize syscall interface

Problem: Users want features

Modern defense: seccomp-bpf

- Whitelist only needed syscalls
- Chrome: ~50
- Docker: ~150

Modern Optimizations

Improving system call performance

- **Software:** vDSO (virtual dynamically linked shared object)
  - A kernel mechanism for exporting kernel space routines to user space
  - No context switching overhead
- E.g., **`gettimeofday()`**
  - The kernel allows the page containing the current time to be mapped **read-only** into user space
  - New flow on every **`gettimeofday()`** execution:

1. `gettimeofday()` called	[5 cycles]
2. vDSO function reads mapped data	[15 cycles]
3. Return	[5 cycles]
Total: ~25 cycles	

Security tradeoff: no mode switch, kernel data exposed, KASLR bypass

What else can we optimize?

vDSO functions available:

- gettimeofday() - Just read memory
- clock_gettime() - Multiple clock sources
- time() - Seconds since epoch
- getcpu() - Current CPU number

Cannot vDSO (need kernel):

- getpid() - Need task_struct
- open() - Need file system
- socket() - Need network stack

Pattern: Read-only, frequently-called, public data → vDSO candidate

Other optimization: io_uring (requires rethinking)

Traditional (10 reads):
10 x syscalls @ 150 ns
= 1500 ns

io_uring (10 reads):
1 x setup
0 x syscalls (async!)
= ~200 ns total

io_uring working:

1. Set up shared ring buffers (once)
2. User adds requests to the submission queue
3. Kernel processes asynchronously
4. User polls the completion queue
5. Zero syscalls in steady state

Results:

- Network: 10M packets/sec with one core
- Storage: 3.5M IOPS single thread

Other optimization: eBPF (execute code in the kernel)

Safely run user code in the kernel → with verification

Example: Custom **`gettimeofday()`** with filtering

1. eBPF program → shares a memory with the application and returns the cached time
2. Verifier ensures memory safety and termination guarantee
3. JIT compile and attach to the syscall

Use cases:

- Observability without overhead
- Custom security policies
- Network packet processing

The evolution of a simple system call

`gettimeofday()`: 30 Years of Evolution

- 1991: int 0x80 - 400 cycles
- 1999: sysenter - 200 cycles
- 2003: syscall - 100 cycles
- 2007: vDSO - 25 cycles
- 2018: +KPTI - vDSO unaffected!
- 2024: eBPF custom - 10 cycles possible

Practical Considerations

Implementing a new system call

1. Write your syscall function
   - Add to the existing file or create a new file
   - Add your new file to the kernel Makefile
2. Add to the syscall table and assign ID
   - **`arch/x86/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

When should you not add a system call?

- Pros: Easy to implement and use, fast
- Cons:
  - Needs an official syscall number
  - The interface cannot change once defined and implemented
  - Must be registered for each architecture
  - Probably too much work for a small exchange of information
- Alternative:
  - Create a device node and **`read()`** and **`write()`**
  - Use **`ioctl()`**

The ABI contract

- System call ABI: A 30-year promise
  - Once a system call is added, it's FOREVER
    - **`stat()`** (1991) → still supported
    - **`stat64()`** (1997) → still supported
    - **`fstat()`** (2006) → still supported
    - **`statx()`** (2017) → currently recommended
- Why not depreciate old names?
  - "We don't break user space" – Linus's #1 rule
- Cost: 400+ syscalls, many redundant
- Benefit: Backward compatible, 30-year-old binaries can still run

Open problems

1. Zero-copy for syscalls for all operations?
2. Hardware-accelerated privilege checks?
3. Contention-aware syscall scheduling
4. Automated application code transformation to use **io_uring** or **eBPF**
5. Learned syscall prediction/prefetching

Next actions

- Lab 1 is out!

Next lecture

- Kernel primitives!

Further readings

- LWN: Anatomy of a system call: part1 and part2
- LWN: On vsyscalls and the vDSO
- Linux Inside: System Calls
- Linux Performance Analysis: New Tools and Old Secrets