W4118 Operating Systems

Junfeng Yang

Outline

- Linux overview
- Interrupt in Linux
- System call in Linux

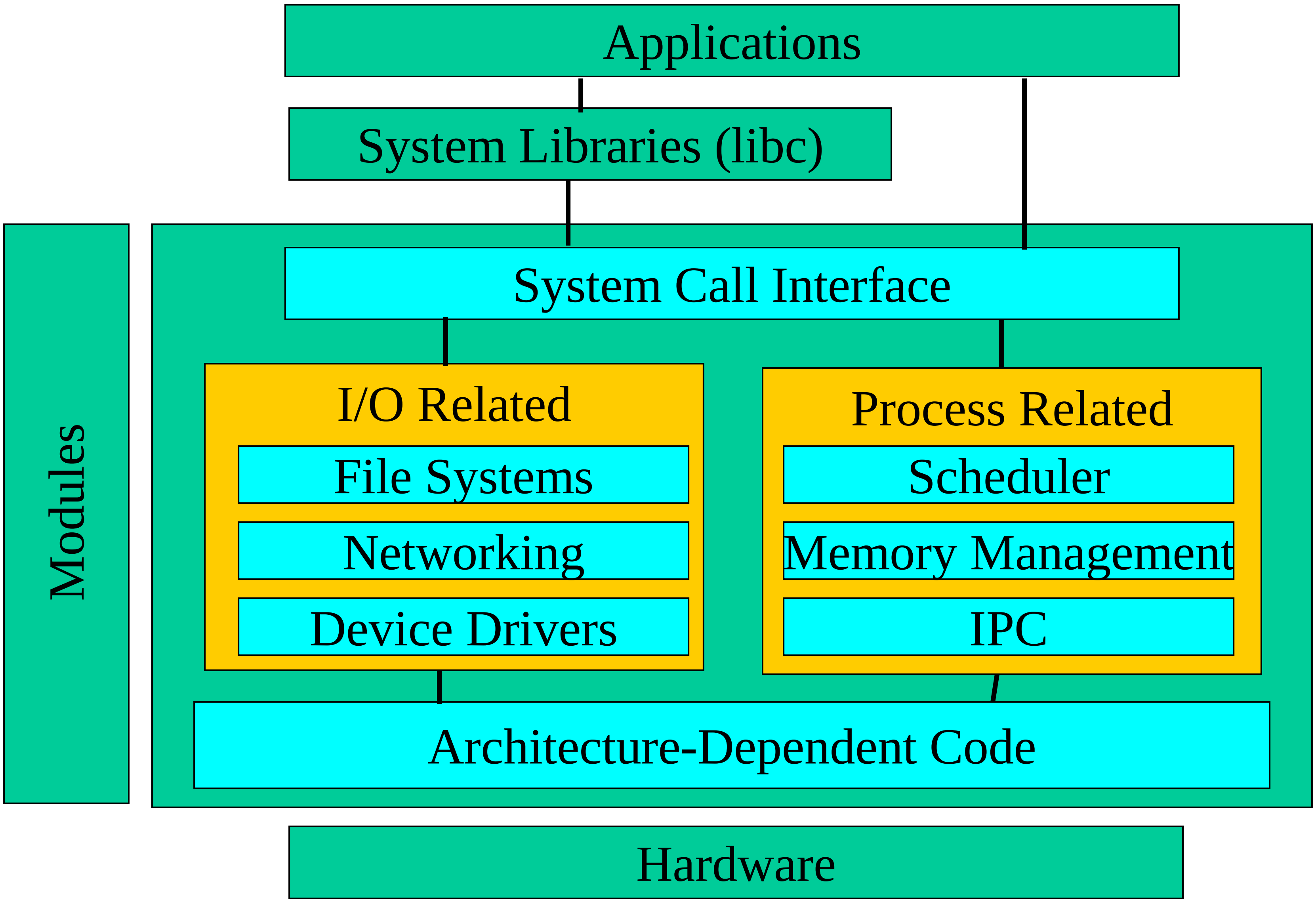
What is Linux

- A modern, open-source OS, based on UNIX standards
  - 1991, 0.1 MLOC, single developer
    - Linus Torvalds wrote from scratch
    - Main design goal: UNIX compatibility
  - Now, 10 MLOC, developers worldwide
    - Unique source code management model
- Linux distributions: ubuntu, redhat, fedora, Gentoo, CentOS, …
  - Kernel is Linux
  - Different set of user applications and package management systems

Linux Licensing

- The GNU General Public License (GPL)
- Anyone creating their own derivative of Linux may not make the derived product proprietary; software released under GPL may not be redistributed as a binary-only product

Linux kernel structure



Linux kernel structure (cont.)

- Core + dynamically loaded modules
  - E.g., device drivers, file systems, network protocols

- Modules were originally developed to support the conditional inclusion of device drivers
  - Early OS has to include code for all possible device or be recompiled to add support for a new device

- Modules are now used extensively
  - Standard way to add new functionalities to kernel
  - Reasonably well designed kernel-module interface

Linux kernel source

- Download: kernel.org
- Browse: lxr.linux.no (with cross reference)
- Directory structure
  - include: public headers
  - kernel: core kernel components (e.g., scheduler)
  - arch: hardware-dependent code
  - fs: file systems
  - mm: memory management
  - ipc: inter-process communication
  - drivers: device drivers
  - usr: user-space code
  - lib: common libraries

Outline

- Linux overview
- Interrupt in Linux
- System call in Linux

Privilege level

- Supports four rings (privilege levels); most modern kernels use only two level
  - ring 3: user mode
  - ring 0: kernel mode
- CPU keeps track of the current privilege level (CPL) using the cs segment register
- In Linux
  - __USER_CS: selector for user code segment
  - __KERNEL_CS: selector for kernel code segment
  - include/asm-i386/segment.h

Memory protection

- Segmentation: physical memory is organized as variable-size segments
- Paging: physical memory is organized as equal-size pages
- The (simplified) idea: memory is associated with descriptor privilege level (DPL)
  - if CPL <= DPL, access okay

Interrupt classification

- Interrupts, asynchronous from device
  - Maskable interrupts
  - Non-Maskable interrupts (NMI): hardware error
- Exceptions, synchronous from CPU
  - Intel manual used a bunch of different terms …
  - Faults: instruction illegal to execute
    - Often correctable and instruction retried
  - Traps: instruction intends to switch control to kernel
    - Resume from the next instruction

Interrupt number assignment

- Total 255 possible interrupts
- 0-31: reserved for non-maskable interrupt
  - 0: division by 0
  - 3: breakpoint
  - 14: page fault
- Remaining 224: programmable by OS
  - 0x80: Linux interrupt

Interrupt descriptor table

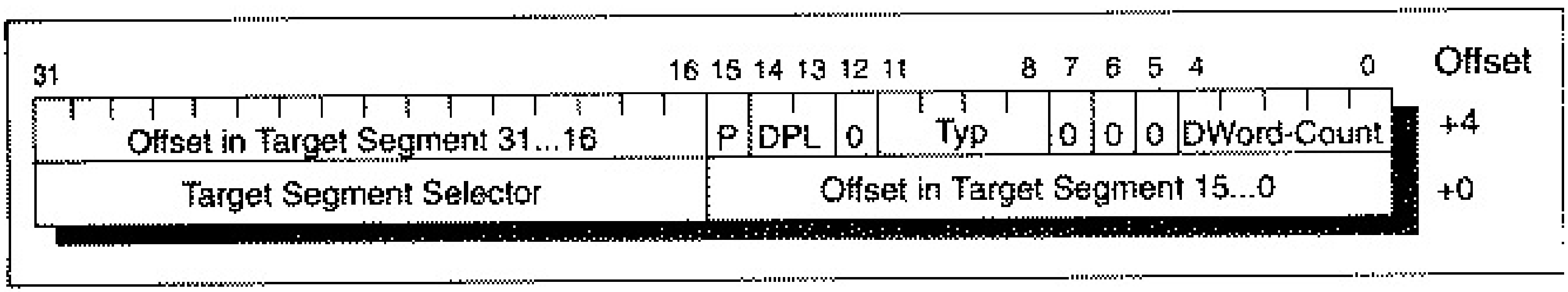


Figure 3.12: Format of an i386 gate descriptor.

- Gate descriptor
- Preventing user code from triggering random interrupts
  - On Trap, if CPL <= Gate DPL, access ok

Seting up IDT in Linux

- Initialization
  - Start by setting all descriptors to ignore_int()
- Then, set up the gate descriptors
  - arch/i386/kernel/traps.c

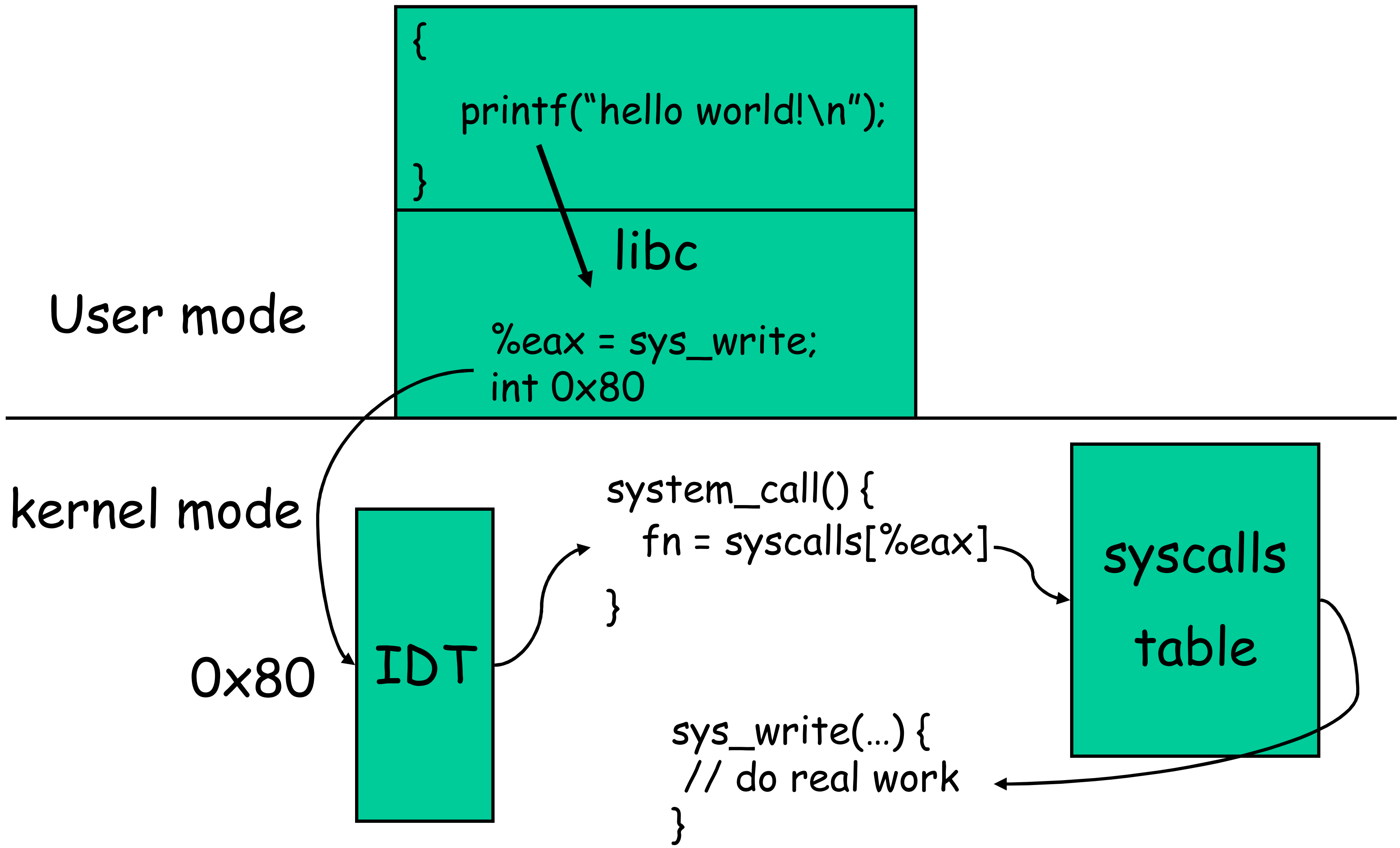
Linux Lingo

- Linux interrupt gate: Intel interrupt, from device
  - DPL = 0
  - Disable interrupt
  - set_intr_gate(2, &nmi)
- System gate: Intel trap, instruction intends to trigger interrupt
  - DPL = 3
  - Often disable interrupt
  - set_system_gate(SYSCALL_VECTOR, &system_call)
- Trap gate: Intel fault, instruction illegal
  - DPL = 0
  - set_trap_gate(0, &divide_error)

Outline

- Linux overview
- Interrupt in Linux
- System call in Linux

Linux system call overview



Syscall wrapper macros

- Macros with name _syscallN(), where N is the number of system call parameters
  - _syscallN(return_type, name, arg1type, arg1name, …)
  - in linux-2.6.11/include/asm-i386/unistd.h
  - Macro will expands to a wrapper function
- Example:
  - long open(const char *filename, int flags, int mode);
  - _syscall3(long, open, const char *, filename, int, flags, int, mode)
- Note: _syscallN obsolete after 2.6.18; now syscall (…), can take different # of args

Lib call/syscall return codes

- Library calls return -1 on error and place a specific error code in the global variable errno
- System calls return specific negative values to indicate an error
  - Most system calls return –errno
- The library wrapper code is responsible for conforming the return values to the errno convention

System call dispatch (arch/i386/kernel/entry.S)

```
        .section        .text
system_call:
    // copy parameters from registers onto stack…
        call    sys_call_table(, %eax, 4)
        jmp     ret_from_sys_call

ret_from_sys_call:
    // perform rescheduling and signal-handling…
        iret            // return to caller (in user-mode)

// File arch/i386/kernel/entry.S
```

Why jump table? Can't we use if-then-else?

The system-call jump-table

- There are approximately 300 system-calls
- Any specific system-call is selected by its ID-number (it's placed into register %eax)
- It would be inefficient to use if-else tests to transfer to the service-routine's entry-point
- Instead an array of function-pointers is directly accessed (using the ID-number)
- This array is named '**sys_call_table**[]'
  - Defined in file arch/i386/kernel/entry.S

System call table definition

```
        .section        .data
sys_call_table:
        .long           sys_restart_syscall
        .long           sys_exit
        .long           sys_fork
        .long           sys_read
        .long           sys_write
        ...
```

NOTE: should avoid reusing syscall numbers (why?); deprecated syscalls are implemented by a special "not implemented" syscall (sys_ni_syscall)

Syscall naming convention

- Usually a library function "foo()" will do some work and then call a system call ("sys_foo()")
- In Linux, all system calls begin with "sys_"
  - Reverse is not true
- Often "sys_foo()" just does some simple error checking and then calls a worker function named "do_foo()"

Tracing System Calls

- Use the "strace" command (man strace for info)
- Linux has a powerful mechanism for tracing system call execution for a compiled application
- Output is printed for each system call as it is executed, including parameters and return codes
- The ptrace() system call is used to implement strace
  - *Also used by debuggers (breakpoint, singlestep, etc)*
- You can trace library calls using the "ltrace" command

Passing system call parameters

- The first parameter is always the syscall #
  - eax on Intel

- Linux allows up to six additional parameters
  - ebx, ecx, edx, esi, edi, ebp on Intel

- System calls that require more parameters package the remaining parameters in a struct and pass a pointer to that struct as the sixth parameter

- Problem: must validate pointers
  - Could be invalid, e.g. NULL ➔ crash OS
  - Or worse, could point to OS, device memory ➔ security hole

How to validate user pointers?

- Too expensive to do a thorough check
  - Must check that the pointer is within all valid memory regions of the calling process
- Solution: no comprehensive check, but users have to use paranoid routines to access user pointers

Paranoid functions to access user pointers

Function	Action
get_user(), __get_user()	reads integer (1,2,4 bytes)
put_user(), __put_user()	writes integer (1,2,4 bytes)
copy_from_user(), __copy_from_user	copy a block from user space
copy_to_user(), __copy_to_user()	copy a block to user space
strncpy_from_user(), __strncpy_from_user()	copies null-terminated string from user space
strnlen_user(), __strnlen_user()	returns length of null-terminated string in user space
clear_user(), __clear_user()	fills memory area with zeros

Intel Fast System Calls

- int 0x80 not used any more (I lied …)
- Intel has a hardware optimization (sysenter) that provides an optimized system call invocation
- Read the gory details in ULK Chapter 10

Next lecture

- Process
- Homework 2 will be out tonight