Protected Mode and System Calls

CS101C.1 Operating Systems
Operating System Kernel

• OS kernel can do a lot of things that user-mode programs shouldn’t be allowed to do

• Make some tasks easier for programs
  – Kernel interacts directly with hardware resources, on behalf of programs
  – Provide helpful abstractions such as multitasking

• Must also protect programs from each other
  – Maintain isolated address spaces between programs
  – Ensure access to shared resources is coordinated
Operating System Kernel (2)

• Kernel must be able to access hardware, and manage resources for other programs...
  – Definitely don’t want other programs to have this capability!

• Processors provide hardware support for this:
  – Multiple privilege levels, arranged in a hierarchy
  – Frequently called protection rings
  – Each ring allows a specific set of operations
  – Can only transfer control between rings in a controlled manner
IA32 Privilege Levels

• IA32 has four protection rings:
  – Ring 0 has greatest capability, intended for kernel
  – Ring 3 has least capability, intended for user applications
  – (Rings 1 and 2 can be used for drivers, other OS services, but aren’t always used by OSes on IA32)

• Also called privilege levels
  – e.g. code running in ring 0 is at privilege level 0
IA32 Control Registers

• IA32 has several control registers
  – cr0, cr2, cr3, ...
  – Control low-level capabilities of processor
  – cr0 used to enable/disable protected mode
  – cr3 is the Page Directory Base Register (PDBR) – specifies virtual-to-physical memory mapping
  – On a page fault, cr2 holds address being accessed

• Clearly don’t want user-mode programs manipulating these registers...
IA32 Control Registers (2)

• IA32 control registers: cr0, cr2, cr3, ...
• Can only access and manipulate these control registers when in protection ring 0
  – At boot, CPU is at privilege level 0
  – Bootloader and kernel can manipulate control registers, perform other CPU setup for OS
  – Kernel can setup the system to run itself in ring 0, and user programs in ring 3
IA32 Protected Memory Access

• IA32 segmented memory model can also be used with protection levels

• Global Descriptor Table (GDT) describes various memory segments:
  – Where the segment starts, what its length is
  – Protection level required to access the segment

• (Can also create Local Descriptor Tables for different processes; usu. not done in practice)
IA32 Protected Memory Access (2)

• Can use GDT to create multiple segments for the operating system to use
• Create segments that only code in ring 0 can access and manipulate
  – Store kernel code and data structures in this area
  – Address range is called “kernel space”
• Create other segments that code in ring 3 can access and manipulate
  – User programs reside in this area
  – Address range is called “user space” or “userland”
IA32 Protected Memory Access (3)

- Example: Pintos (see userprog/gdt.c)
- Creates a code and data segment for ring 3, for user programs
  - Virtual address range is [0..PHYS_BASE)
  - PHYS_BASE defaults to 0xC0000000 (3GB)
    - See threads/vaddr.h
- Creates another code and data segment for ring 0
  - Address range [PHYS_BASE..0x100000000) (4GB)
  - Only code at protection level 0 can access this range!
Protection Levels and Stacks

• Protection levels also have their own stacks

• Reason 1:
  – Keep user-mode programs from manipulating or corrupting the stack being used by the kernel
  – Stack for ring 0 only accessible by code in ring 0

• Reason 2:
  – Ensure that the kernel always has enough stack space to service requests from user programs
  – Even if a user program is being a memory hog, kernel can still properly handle interrupts, system calls, etc.
IA32 Protected-Mode Stacks

- IA32 provides basic multitasking support
  - Tasks: units of work that the processor can dispatch, execute and suspend
  - Generally not used by operating systems to implement multitasking
- Important for managing protected-mode stacks
  - IA32 Task Register points to a “task-state segment” (TSS) data structure
  - Structure describes the stack to use for protection levels 0, 1, 2
IA32 Protected-Mode Stacks (2)

• During initialization, kernel can set up a single task, specifying memory areas for each stack
  – Creates a Task-State Segment structure, then records this in the IA32 Task Register

• When the CPU switches between protection levels, this task is used to switch stacks

• Pintos: see userprog/tss.c
IA32, GDT, TSS, and IDT

- Pointer to Global Descriptor Table is stored in the Global Descriptor Table Register (GDTR)
  - Manipulated with LGDT and SGDT instructions
- Task Register is manipulated with the LTR and STR instructions
- Also: Interrupt Descriptor Table (used for dispatching hardware and software interrupts)
  - Stored in the Interrupt Descriptor Table Register
  - Manipulated with LIDT and SIDT instructions
- All of these instructions can only be executed when in privilege level 0 (duh)
Kernel Space and User Space

• Processor provides extensive capabilities to isolate kernel code and data from user-mode programs
  – User-mode programs can’t access kernel code or data without causing general protection fault
  – User-mode programs don’t even share the same stack with the kernel, and can’t affect kernel stack
  – User-mode programs can’t manipulate the CPU control registers that govern this behavior
System Calls

• However: kernel manages the hardware on behalf of user-mode programs
  – Still need mechanism to allow user programs to make requests to the kernel: system calls

• Kernel will carefully examine the request
  – If allowed, kernel will perform the operation on behalf of the user-mode program
  – If not allowed, kernel can either return an error, or it can terminate the program if necessary
System Calls (2)

• Kernel code and data are present in the virtual address space...
  – Can’t just jump to it though: general protection fault!
  – Also, won’t properly switch to kernel stack

• Additional wrinkle:
  – Sometimes kernel gets invoked from user-mode programs, and sometimes kernel is invoked while other kernel code is running!
  – e.g. kernel accesses memory, causing a page fault
    • Kernel page-fault interrupt handler gets invoked
Switching Protection Levels

- IA32 provides “gate” abstraction for changing protection levels
  - Several kinds of gates: call gates, trap gates, interrupt gates, and task gates
- Purpose:
  - Only allow access from lower privilege levels at a few specific, carefully controlled access points
  - Makes it much easier to build a bulletproof kernel
Interrupt/Trap Gates

• Operating systems usually expose system calls via interrupt/trap gates
• At boot, kernel installs an interrupt gate
  – Allows transfer into privilege level 0
• User-space programs can execute an \texttt{int \text{xx}} instruction to make a request of the kernel
  – CPU will automatically change stacks if the privilege level is changing
Interrupt/Trap Gates (2)

- Examples:
  - Windows uses `int $0x2e` for system calls
  - Linux uses `int $0x80` for system calls
  - Pintos uses `int $0x30` for system calls
System Calls: Issues

• Problems with system calls:
  – Can’t pass arguments to the system call on the stack – kernel will have a whole separate stack
  – Often, arguments to a system call will be pointers to data or code in the user-mode address space
    • Kernel may need to read or write userspace code/data

• Registers are used for passing arguments to system calls
  – Example: On Windows and Linux, eax is set to the specific operation being requested
Dispatching System Calls

• Usually, ID of system call is used to dispatch to a specific kernel function that implements the call
  – Called a *system call service routine*

• Kernel uses a system-call table that stores an array of function-pointers to service routines

• Important checks:
  – Make sure ID of system call is within table (duh)
  – For unimplemented syscalls, use a placeholder implementation that simply reports an error
Example: Linux System Calls

• Linux has a sophisticated mechanism for providing system calls
• User-space C API is syscall(id, ...)
• Other operations, e.g. write(), fork(), kill() are wrappers to syscall()
• Because kernel stack is different from user stack, all arguments are passed in registers
  – eax = ID, ebx = arg 1, ecx = arg 2, edx = arg 3, esi = arg 4, edi = arg 5
• syscall() stores args in registers, then int $0x80
Example: Linux System Calls (2)

- Linux int $0x80$ handler is implemented by a kernel function `system_call()`
  - Stores all regs onto the kernel stack, in cdecl order
  - Uses eax to dispatch to appropriate service routine
  - Or, report an error if no routine corresponding to the specified ID

- System call service routines can access their args on the stack using cdecl convention
  - Can implement service routines in C!
Example: Linux System Calls (2)

• System call service routines are usually named after their user-mode entry points
  – `sys_write()` implements `write()`
  – `sys_fork()` implements `fork()`
  – etc.
Example: Linux System Calls (3)

• Snippet from Linux system_call() handler:

...  # Save registers onto stack

# Make sure it's a valid syscall ID
cmpl $(NR_syscalls), %eax
jb nobadsys

# Return-value of syscall() will be in eax
# as usual, so set value of eax stored on
# kernel stack to ENOSYS to indicate error
movl $(-ENOSYS), 24(%esp)
jmp ret_from_sys_call

nobadsys:
...
Example: Linux System Calls (4)

• Linux system_call() handler, continued:

    ... 

nobadsys:
    # Dispatch to the function in the 
    # system-call table corresponding to 
    # the specified ID 
    call *sys_call_table(, %eax, 4)

    # Store return-value from routine into 
    # location of eax on the kernel stack 
    movl %eax, 24(%esp) 
    jmp ret_from_sys_call
Finally, the system_call() handler restores the registers and returns back to user program via iret

- int $0x80 was invoked from syscall() wrapper
- eax is set to the appropriate return-value

syscall() wrapper returns to the user program, which sees the kernel’s response in eax

- If kernel wrote data to a buffer in user space, etc., this would also be visible to the user program
System Calls: Security Holes?

- Are there potential security holes in accepting pointers as arguments to system calls?
System Calls: Security Holes?

• Are there potential security holes in accepting pointers as arguments to system calls?
  – e.g. read() reads bytes from a file descriptor into a buffer
  – Caller specifies:
    • The file-descriptor to read
    • A pointer to the buffer to store the data in
    • A number of bytes to read

• What if the user-mode program specifies an address in the kernel’s address space?
System Calls: Security Holes?!

• What if the user-mode program specifies an address in the kernel’s address space?
  – As long as user-mode program doesn’t access this address, it won’t cause a general protection fault

• The kernel is allowed to write to this address!
  – If kernel naively uses the address from the user-mode program, it could overwrite critical data
  – e.g. change root password, grant access to user programs, other exploits…
System Calls: Security Holes

• Extremely important to verify addresses that come from user-mode programs:
  – Address **must** be in userspace! If it’s in kernel space, it’s an access violation.