Kernel Programming

• Learning Objectives
  • Explain how control transfers between user-level processes and the kernel.
  • Be able to use privileged instructions when writing kernel code.
  • Be able to complete assignment 6

• Topics
  • Exceptional control transfer
  • A Walkthrough of the OS from Thursday
Exceptional Control Flow

• Normal control flow
  • Left to its own devices, a processor issues instructions sequentially. That is, by default, each time it executes an instruction, it moves the instruction pointer to the next consecutive instruction.
  • Some instructions disrupt this sequential execution, but are still perfectly normal:
    • Jump
    • Call
    • Jump if <condition>

• Exceptional control flow
  • Sometimes, it is useful, imperative to have execution proceed at a location not expressed by the current process’s instructions.
We have seen an example this semester of exceptional control flow. What was it?

- The pop instruction
- The pushing of a return address on the stack
- The jz instruction
- Execution in signal handlers
Exceptions

• From the book: an abrupt change in the control flow in response to some change in the processor’s state.

• Exceptions are triggered by events:
  • A process might do something the processor simply cannot do: divide by 0.
  • A process might request service of the operating system: issue a system call.
  • A process tries to do something that requires help from the operating system: tries to access a page that is valid in its address space, but not currently in memory.
  • A hardware event might occur: a packet arrives on the network.
Exception Handling

• On an exception, the processor makes an indirect function call through a dispatch table.
• The code invoked through this call is an exception handler.
• The exception handler handles the exception:
  • Abort the process (divide by 0; access invalid address)
  • Satisfy the process request (execute the system call)
  • Provide something to the process (satisfy a page fault)
• Unless the process was killed, control returns to one of:
  • The instruction causing the exception (page fault)
  • The instruction following the instruction that caused the exception (system call).
Types of Exceptions

• **Interrupts**
  - Asynchronously with respect to the program.
  - The response to a hardware event, such as, a network packet, a disk request completion.
  - The term *interrupt handler* simply means that exception handler for an interrupt.

• **System calls (Traps)**
  - Synchronous with respect to the program.
  - Intentional request for kernel to do something.

• **Faults**
  - Synchronous with respect to the program.
  - Unintentional
  - If the OS can “fix” the fault, it will; else it will abort

• **Aborts**
  - Detected synchronously
  - Unrecoverable errors
  - Terminate the process
Exceptional Control Transfer: In General

• At boot time, the OS sets up a dispatch table.
  • Index = exception number
  • Contents = address of the exception handler

• On exception, the processor:
  • Changes (or stays in) privileged mode
  • Saves away necessary state
  • Transfers control to the exception handler.

• The handler:
  • Finds a kernel stack (if necessary).
  • Saves any additional state not already saved by the hardware.
  • Invokes whatever kernel functions are necessary.
Exceptional Control Transfer: x86

- At boot time, the OS sets up a dispatch table (Interrupt Descriptor Table or IDT; referenced by the IDT Register IDTR):
  - Index = exception number (IDTR contains size of the IDT)
  - Contents = address of the exception handler
  - LIDT: Loads memory into the IDT; SIDT: Stores IDT into memory
- On exception, the processor:
  - Changes (or stays in) privileged mode (bits 12-13 of the EFLAGS register; disables interrupts bit 9)
  - Saves away necessary state (EFLAGS; CR2 contains the address causing the fault)
  - Transfers control to the exception handler.
- The handler:
  - Finds a kernel stack (if necessary).
  - Saves any additional state not already saved by the hardware.
  - Invokes whatever kernel functions are necessary.
Let’s look at the OS from Thursday

- **Code overview:**
  - `kernel.c/kernel.h`: Main kernel code.
  - `k-exception.S`: Exception (interrupt) handlers.
  - `x86.h`: Hardware specific structures
  - `k-hardware.c`: Connects kernel to the hardware
  - `lib.c/lib.h`: Library code used by both kernel and user processes.
  - `elf.h`: Describes the layout of processes (and, in particular, the kernel).
  - `bootstart.S/boot.c`: Bootloader
Getting Started

• How do we get to running our operating system?
  • File `bootstart.S`
  • The BIOS (Basic input/output system) initializes the hardware and then starts looking for a boot block (512 bytes).
  • It starts reading the first sector off of any disk until it finds one with a valid checksum.
  • It then loads those 512 bytes into physical memory at address 0x7c00-0x7DFF.
  • Then starts executing whatever was in the boot block (which is in `bootstart.S` and `boot.c`).
  • While you are welcome to read all of `bootstart.S`, you need not do so. It is an interesting historical journey.
  • The code in `bootstart.S` basically does everything we need to do in assembly code (e.g., initializes registers and sets up a stack) and then jumps into the C code in `boot.c`.
  • The whole goal of the bootloader is to read the operating system from disk and transfer control to it.
~ [1] cd
l23 [5] ls
COPYRIGHT boot.c build k-exception.S kernel.c lib.c link x86.h
GNUmakefile bootstart.S elf.h k-hardware.c kernel.h lib.h log.txt
l23 [6]  

VM Initialization

- We need to construct a kernel page table.
- In Weensy, we construct the identity page table:
  - Maps virtual pgno N to physical pgno N.
- Then we have the special register CR3 point to the kernel page table.
```c
// `kernel_pagetable`.

static x86_pagetable kernel_pagetable_memory;
static x86_pagetable kernel_level2_pagetable;
x86_pagetable* kernel_pagetable;

void virtual_memory_init(void) {
    kernel_pagetable = &kernel_pagetable_memory;
    memset(kernel_pagetable, 0, sizeof(x86_pagetable));
    kernel_pagetable->entry[0] = (x86_pageentry_t) &kernel_level2_pagetable
                           | PTE_P | PTE_W | PTE_U;

    virtual_memory_map(kernel_pagetable, (uintptr_t) 0, (uintptr_t) 0,
                      MEMSIZE_PHYSICAL, PTE_P | PTE_W | PTE_U);

    // Use special instructions to initialize paged virtual memory.
    lcr3((uintptr_t) kernel_pagetable);
    uint32_t cr0 = rcr0();
    cr0 |= CR0_PE | CR0_PG | CR0_AM | CR0_WP | CR0_NE | CR0_TS
         | CR0_EM | CR0_MP;
```
# Interrupt handlers

.align 2

.globl gpf_int_handler

gpf_int_handler:
    pushl $13       // trap number
    jmp _generic_int_handler

.globl pagefault_int_handler

pagefault_int_handler:
    pushl $14
    jmp _generic_int_handler

.globl timer_int_handler

timer_int_handler:
    pushl $0        // error code
    pushl $32
    jmp _generic_int_handler
Wrapping Up

• Control flow in the OS is, perhaps, a bit more confusing than in regular user processes.
• BUT – code is code is code. You know enough to work your way through it.
• Intentional entry and exit into the kernel on an x86 uses:
  • INT n: generates an interrupt with number n
  • Kernel places return value in %eax
  • Kernel uses the iret instruction to return to unprivileged mode.