Traps, Exceptions, System Calls, & Privileged Mode

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

P&H Chapter 4.9, pages 509–515, appendix B.7
Operating Systems
Control Transfers to OS

Case 1: Program invokes OS

- eg: sbrk(), mmap(), sleep()
- Like a function call: invoke, do stuff, return results

Attempt #1: OS as a library

- Just a function call: JAL sbrk
- Standard calling conventions
Virtual to physical address translation

Hardware (typical):
- Traverse PageTables on TLB miss, install TLB entries
- Update dirty bit in PTE when evicting
- Flush when PTBR changes

Software (typical):
- Decide when to do context switches, update PTBR
- Decide when to add, remove, modify PTEs and PDEs
  - and invoke MMU to invalidate TLB entries
- Handle page faults: swap to/from disk, kill processes

Hardware (minimal):
- Notify OS on TLB miss; software does everything else
Control Transfers to OS

Case 1: Program invokes OS
- eg: sbrk(), mmap(), sleep()
- Like a function call: invoke, do stuff, return results

Case 2: Hardware invokes OS on behalf of program
- Page fault, divide by zero, arithmetic overflow, ...
- OS takes corrective action; then restarts/kills program

Can CPU simply fake this:
\[
a0 = \text{cause} \\
\text{JAL \ exception\_handler}
\]
Attempt #2: OS as a library + Exception Handler

Program invokes OS: regular calling convention

HW invokes OS:

- New registers: EPC, Cause, Vector*, ...
- On exception, CPU does...
  - EPC ← PC
  - Cause ← error/reason code
  - PC ← Vector
- Code at Vector does...
  - take corrective action based on Cause
  - return to EPC

* x86: via IDTR register and IDT; MIPS used a constant
# MIPS exception vector is 0x80000180
# EPC has offending PC, Cause has errcode
# (step 1) save *everything* but $k0, $k1
lui $k0, 0xB000
sw $1, 0($k0)
sw $2, 4($k0)
sw $3, 8($k0)
sw $4, 12($k0)
...
sw $31, 120($k0)
mflo $1
sw $1, 124($k0)
mfhi $1
sw $1, 128($k0)
...

* approximate
# MIPS exception vector is 0x80000180
.ktext 0x80000180
# EPC has offending PC, Cause has errcode
# (step 1) save *everything* but $k0, $k1
# (step 2) set up a usable OS context
li $sp, 0xFFFFFFFF00
li $fp, 0xFFFFFFFF
li $gp, ...

* approximate
Sketch of Exception Handler

# MIPS exception vector is 0x80000180
.ktext 0x80000180
# EPC has offending PC, Cause has errcode
# (step 1) save *everything* but $k0, $k1
# (step 2) set up a usable OS context
# (step 3) examine Cause register, and take corrective action
mfc0 $t0, Cause # move-from-coprocessor-0
if ($t0 == PAGE_FAULT) {
    mfc0 $a0, BadVAddr # another dedicated register
    jal kernel_handle_pagefault
} else if ($t0 == PROTECTION_FAULT) {
    ...
} else if ($t0 == DIV_BY_ZERO) {
    ...
}

* approximate
# MIPS exception vector is 0x80000180
.ktext 0x80000180
# EPC has offending PC, Cause has errcode
# (step 1) save *everything* but $k0, $k1
# (step 2) set up a usable OS context
# (step 3) examine Cause register, and take corrective action
# (step 4) restore registers and return to where program left off
lui $k0, 0xB000
lw $1, 0($k0)
lw $2, 4($k0)
lw $3, 8($k0)
...lw $31, 120($k0)
...
mfc0 $k1, EPC
jr $k1

* approximate
Hardware Support:

- registers: EPC, Cause, Vector, BadVAddr, ...
- instructions: mfc0, TLB flush/invalidate, cache flush, ...

Hardware guarantees for precise exceptions:

- EPC points at offending instruction
- Earlier instructions are finished
- EPC and later instructions have not started
- Returning to EPC will pick up where we left off
• EPC points at offending inst
• Earlier inst are finished; EPC and later inst not started
• Returning to EPC will pick up where we left off

What could possibly go wrong?

Exception happens during exception handler...
original EPC and Cause are lost

• Disable exceptions until current exception is resolved?
  – MIPS: Status register has a bit for enable/disable
  – turn exceptions back on just when returning to EPC
  – works for issues that can be (temporarily) ignored

• Use a “double fault” exception handler for rest
  – BSOD

• And if that faults? Triple fault → instant shutdown
• EPC points at offending inst
• Earlier inst are finished; EPC and later inst not started
• Returning to EPC will pick up where we left off

What could possibly go wrong?

Multiple simultaneous exceptions in pipeline

```
lw $4, 0($0) # page fault
xxx $4, $5, $5 # illegal instruction
add $2, $2, $3 # overflow
```

• need stalls to let earlier inst raise exception first
• even worse with speculative / “out-of-order” execution
• EPC points at offending inst
• Earlier inst are finished; EPC and later inst not started
• Returning to EPC will pick up where we left off

What could possibly go wrong?

Exception happened in delay slot

jal prints
lw $4, 0($0) # page fault

• need more than just EPC to identify “where we left off”
• EPC points at offending inst
• Earlier inst are finished; EPC and later inst not started
• Returning to EPC will pick up where we left off

What could possibly go wrong?

Instructions with multiple faults or side effects

- store-and-update-register
- memory-to-memory-copy
- memory-fill, x86 “string” prefix, x86 “loop” prefix

• need more than just EPC to identify “where we left off”
• or: try to undo effects that have already happened
• or: have software try to finish the partially finished EPC
• or: all of the above
“The interaction between branch delay slots and exception handling is extremely unpleasant and you'll be happier if you don't think about it.”

– Matt Welch
Attempt #2: Recap

Program invokes OS
  • regular calling convention

HW invokes OS:
  • precise exceptions vector to OS exception handler

Drawbacks?
Drawbacks:

- Any program can muck with TLB, PageTables, OS code...
- A program can intercept exceptions of other programs
- OS can crash if program messes up $sp, $fp, $gp, ...

Wrong: Make these instructions and registers available only to “OS Code”

- “OS Code” == any code above 0x80000000
- Program can still JAL into middle of OS functions
- Program can still muck with OS memory, pagetables, ...
Privileged Mode
aka Kernel Mode
Some things not available to untrusted programs:

- Exception registers, HALT instruction, MMU instructions, talk to I/O devices, OS memory, ...

Need trusted mediator: **Operating System (OS)**

- *Safe control transfer*
- *Data isolation*
CPU Mode Bit / Privilege Level / Status Register

Mode 0 = untrusted = user domain
  • “Privileged” instructions and registers are disabled by CPU

Mode 1 = trusted = kernel domain
  • All instructions and registers are enabled

Boot sequence:
  • load first sector of disk (containing OS code) to well known address in memory
  • Mode ← 1; PC ← well known address

OS takes over...
  • initialize devices, MMU, timers, etc.
  • loads programs from disk, sets up pagetables, etc.
  • Mode ← 0; PC ← program entry point

(note: x86 has 4 levels x 3 dimensions, but nobody uses any but the 2 extremes)
CPU Mode Bit / Privilege Level Status Register

Mode 0 = untrusted = user domain
  - “Privileged” instructions and registers are disabled by CPU

Mode 1 = trusted = kernel domain
  - All instructions and registers are enabled

Boot sequence:
  - load first sector of disk (containing OS code) to well known address in memory
  - Mode $\leftarrow 1$; PC $\leftarrow$ well known address

OS takes over...
  - initialize devices, MMU, timers, etc.
  - loads programs from disk, sets up pagetables, etc.
  - Mode $\leftarrow 0$; PC $\leftarrow$ program entry point

(note: x86 has 4 levels x 3 dimensions, but nobody uses any but the 2 extremes)
**Trap**: Any kind of a control transfer to the OS

**Syscall**: Synchronous (planned), program-to-kernel transfer
- SYSCALL instruction in MIPS (various on x86)

**Exception**: Asynchronous, program-to-kernel transfer
- exceptional events: div by zero, page fault, page protection err, ...

**Interrupt**: Asynchronous, device-initiated transfer
- e.g. Network packet arrived, keyboard event, timer ticks

* real mechanisms, but nobody agrees on these terms
System call examples:

putc(): Print character to screen
  • Need to multiplex screen between competing programs

send(): Send a packet on the network
  • Need to manipulate the internals of a device

sbrk(): Allocate a page
  • Needs to update page tables & MMU

sleep(): put current prog to sleep, wake other
  • Need to update page table base register
System call: Not just a function call

- Don’t let program jump just anywhere in OS code
- OS can’t trust program’s registers (sp, fp, gp, etc.)

**SYSCALL instruction**: safe transfer of control to OS

- Mode $\leftarrow 0$; Cause $\leftarrow$ syscall; PC $\leftarrow$ exception vector

**MIPS system call convention:**

- user program mostly normal (save temps, save ra, ...)
- but: $v0 = \text{system call number}$
int getc() {
    asm("addiu $2, $0, 4");
    asm("syscall");
}

char *gets(char *buf) {
    while (...) {
        buf[i] = getc();
    }
}
Compilers do not emit SYSCALL instructions

- Compiler doesn’t know OS interface

Libraries implement standard API from system API

\texttt{libc} (standard C library):

- \texttt{getc()} $\rightarrow$ syscall
- \texttt{sbrk()} $\rightarrow$ syscall
- \texttt{write()} $\rightarrow$ syscall
- \texttt{gets()} $\rightarrow$ \texttt{getc()}
- \texttt{printf()} $\rightarrow$ \texttt{write()}
- \texttt{malloc()} $\rightarrow$ \texttt{sbrk()}
- ...
user

kernel
Kernel code and data lives above 0x80000000

In same virtual address space as user process?

- but... user code can modify kernel code and data!
Kernel code and data lives above 0x80000000
In its own address space?

- all traps switch to a different address space [expensive]
- prints("hi") syscall is tricky [why?]
Kernel code and data lives above 0x80000000

Solution

- map kernel code/data into all processes at same vaddr
- but use supervisor=1 protection bit on PTEs
- VM hardware enforces user/kernel isolation
Interrupts
Recap: Traps

→ Map kernel into every process using *supervisor* PTEs
→ Switch to *kernel mode* on trap, *user mode* on return

**Syscall:** Synchronous, program-to-kernel transfer
- user does caller-saves, invokes kernel via syscall
- kernel handles request, puts result in v0, and returns

**Exception:** Asynchronous, program-to-kernel transfer
- user div/load/store/... faults, CPU invokes kernel
- kernel saves everything, handles fault, restores, and returns

**Interrupt:** Asynchronous, device-initiated transfer
- e.g. Network packet arrived, keyboard event, timer ticks
- kernel saves everything, handles event, restores, and returns
Example: Clock Interrupt*

- Every N cycles, CPU causes exception with Cause = CLOCK_TICK
- OS can select N to get e.g. 1000 TICKs per second

```
.ktext 0x80000180
# (step 1) save *everything* but $k0, $k1 to 0xB0000000
# (step 2) set up a usable OS context
# (step 3) examine Cause register, take action
if (Cause == PAGE_FAULT) handle_pfault(BadVaddr)
else if (Cause == SYSCALL) dispatch_syscall($v0)
else if (Cause == CLOCK_TICK) schedule()
# (step 4) restore registers and return to where program left off
```

* not the CPU clock, but a programmable timer clock
struct regs context[];
int ptbr[];
schedule() {
    i = current_process;
    j = pick_some_process();
    if (i != j) {
        current_process = j;
        memcpy(context[i], 0xB00000000);
        memcpy(0xB00000000, context[j]);
        asm("mtc0 Context, ptbr[j]");
    }
}
Syscall vs. Exceptions vs. Interrupts

Same mechanisms, but...

**Syscall** saves and restores much less state

Others save and restore full processor state

**Interrupt** arrival is unrelated to user code