

CSE 120 Principles of Operating Systems

Fall 2000

Lecture 2: Architectural Support for Operating Systems

Geoffrey M. Voelker

Why Start With Architecture?

- Operating system functionality fundamentally depends upon the architectural features of the computer
- Architectural support can greatly simplify (or complicate) OS tasks
 - ◆ Early PC operating systems (DOS, MacOS) lacked virtual memory in part because the architecture did not support it
 - ◆ Early Sun 1 computers used two M68000 CPUs, which do not support VM, to implement it

Types of Arch Support

- Manipulating privileged machine state
 - ◆ Protected instructions
 - ◆ Manipulate device registers, TLB entries, etc.
- Generating and handling “events”
 - ◆ Interrupts, exceptions, system calls, etc.
 - ◆ Respond to external events
 - ◆ CPU requires software intervention to handle fault or trap

Protected Instructions

- A subset of instructions of every CPU is restricted to use by the OS
 - ◆ Known as protected (privileged) instructions
- Only the operating system can
 - ◆ Directly access I/O devices (disks, printers, etc.)
 - » Security, fairness (why?)
 - ◆ Manipulate memory management state
 - » Page table pointers, page protection, TLB management, etc.
 - ◆ Manipulate protected control registers
 - » Kernel mode, interrupt level
 - ◆ Halt instruction (why?)

OS Protection

- How do we know if we can execute a protected instruction?
 - ♦ Architecture must support (at least) two modes of operation: **kernel** mode and **user** mode
 - » VAX, x86 support four modes; earlier archs (Multics) even more
 - » Why? Protect the OS from itself (software engineering)
 - ♦ Mode is indicated by a status bit in a protected control register
 - ♦ User programs execute in user mode
 - ♦ OS executes in kernel mode (OS == “kernel”)
- Protected instructions only execute in kernel mode
 - ♦ The CPU checks mode bit when protected instr. executes
 - ♦ Setting mode bit must be a protected instruction

Memory Protection

- OS must be able to protect programs from each other
- OS must protect itself from user programs
- May or may not protect user programs from OS
- Memory management hardware provides memory protection mechanisms
 - ♦ Base and limit registers
 - ♦ Page table pointers, page protection, TLB
 - ♦ Virtual memory
 - ♦ Segmentation
- Manipulation of memory management hardware are protected (privileged) operations

Events

- An event is an “unnatural” change in control flow
 - Events immediately stop current execution
 - Changes mode or context (machine state) or both
- The kernel defines a handler for each event type
 - Event handlers always execute in kernel mode
 - The specific types of events are defined by the machine
- Once the system is booted, all entry to the kernel occurs as the result of an event
 - In effect, the operating system is one big event handler

Categorizing Events

- Two kinds of events, interrupts and exceptions
- Exceptions are caused by executing instructions
 - CPU requires software intervention to handle a fault or trap
- Interrupts are caused by an external event
 - Device finishes I/O, timer expires, etc.

	Unexpected	Deliberate
Exceptions (sync)	fault	syscall trap
Interrupts (async)	interrupt	software interrupt

- S/I – also async system trap (AST), async or deferred procedure call (APC or DPC)
- Terms may be used slightly differently by various OS, CPUs

Faults

- Hardware detects and reports “exceptional” conditions
 - ◆ Page fault, unaligned access, divide by zero
- Upon exception, hardware faults
 - ◆ Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
- Modern Oses use VM faults for many functions
 - ◆ Debugging, distributed VM, GC, copy-on-write
- Fault exceptions are a performance optimization
 - ◆ Could detect faults by inserting extra instructions into code (at a significant performance penalty)

Handling Faults

- Some faults are handled by “fixing” the exceptional condition and returning to the faulting context
 - ◆ Page faults cause the OS to place the missing page into memory
 - ◆ Fault handler resets PC of faulting context to re-execute instruction that caused the page fault
- Some faults are handled by notifying the process
 - ◆ Fault handler munges the saved context to transfer control to a user-mode handler on return from fault
 - ◆ Handler must be registered with OS
 - ◆ Unix **signals** or NT **user-mode Async Procedure Calls (APCs)**
 - » SIGHUP, SIGTERM, SIGSEGV, etc.

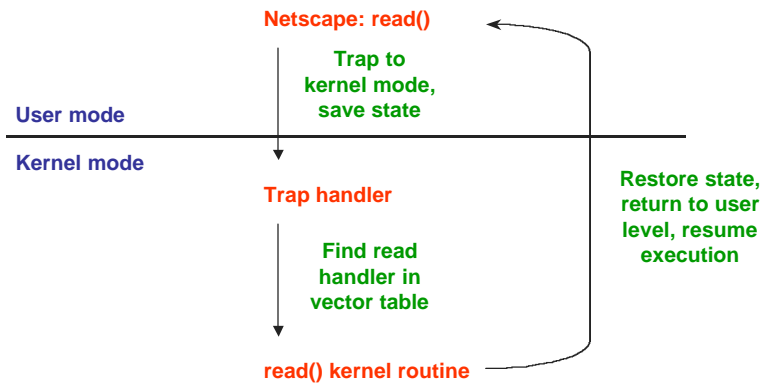
Handling Faults (2)

- The kernel may handle unrecoverable faults by killing the user process
 - ◆ Program fault with no registered handler
 - ◆ Halt process, write process state to file, destroy process
 - ◆ In Unix, the default action for many signals (e.g., SIGSEGV)
- What about faults in the kernel?
 - ◆ Dereference NULL, divide by zero, undefined instruction
 - ◆ These faults considered fatal, operating system crashes
 - ◆ Unix panic, Windows “Blue screen of death”

System Calls

- For a user program to do something “privileged” (e.g., I/O) it must call an OS procedure
 - ◆ Known as **crossing the protection boundary**, or a **protected procedure call**
- Arch provides a *system call* instruction that:
 - ◆ Causes an exception, which vectors to a kernel handler
 - ◆ Passes a parameter determining the system routine to call
 - ◆ Saves caller state (PC, regs, mode) so it can be restored
 - ◆ Returning from system call restores state
- Arch must permit OS to:
 - ◆ Verify input parameters (e.g., valid addresses for buffers)
 - ◆ Restore saved state, return to user mode, resume execution

System Call



September 20, 2000

CSE 120 – Lecture 2 – Arch Support

13

System Call Questions

- What would happen if kernel did not save state?
- What if the kernel executes a system call? What if a user program returns from a system call?
- How to reference kernel objects as arguments or results to/from system calls?
 - ◆ Use integer object handles or descriptors
 - ◆ Also called capabilities (more later)
 - ◆ E.g., Unix file descriptors

September 20, 2000

CSE 120 – Lecture 2 – Arch Support

14

Interrupts

- Interrupts
 - ♦ Precise interrupts – CPU transfers control only on instruction boundaries
 - ♦ Imprecise interrupts – CPU transfers control in the middle of instruction execution
- Interrupts signal asynchronous events
 - ♦ Timer, I/O, etc.

Timer

- The timer is critical for an operating system
- It is the fallback mechanism by which the OS reclaims control over the machine
 - ♦ Timer is set to generate an interrupt after a period of time
 - » Setting timer is a privileged instruction
 - ♦ When timer expires, generates an interrupt
- Prevents infinite loops
 - ♦ OS can always regain control from erroneous or malicious programs that try to hog CPU
- Also used for time-based functions (e.g., *sleep()*)

I/O Control

- I/O issues
 - ◆ Initiating an I/O
 - ◆ Completing an I/O
- Initiating an I/O
 - ◆ Special instructions
 - ◆ Memory-mapped I/O
 - » Device registers mapped into address space
 - » Writing to address sends data to I/O device

I/O Completion

Interrupts are the basis for asynchronous I/O

- OS initiates I/O
- Device operates independently of rest of machine
- Device sends an interrupt signal to CPU when done
- OS maintains a vector table containing a list of addresses of kernel routines to handle various events
- CPU looks up kernel address indexed by interrupt number, context switches to routine

I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack

Synchronization

- Interrupts cause difficult problems
 - ♦ An interrupt can occur at any time
 - ♦ Code can execute that interferes with code that was interrupted
- OS must be able to synchronize concurrent execution
- Need to guarantee that short instruction sequences execute atomically
 - ♦ Disable interrupts – turn off interrupts before sequence, execute sequence, turn interrupts back on
 - ♦ Special atomic instructions – read/modify/write a memory address, test and conditionally set a bit based upon previous value

Next Time...

- Read Chapter 3