CSE 451: Operating Systems
Autumn 2019

Module 2
Architectural Support for Operating Systems

John Zahorjan
Low-level architecture affects the OS dramatically

Who’s making sure the app behaves?

Who should get to define what “behaves” means?

(Hardware provides mechanism and OS provides policy.)
Low-level architecture affects the OS dramatically

• The operating system supports sharing of hardware and protection of hardware
  – multiple applications can run concurrently, sharing resources
  – a buggy or malicious application can’t violate other applications or the system

• Those are high level goals
  – There are many mechanisms that can be used to achieve them

• The architecture determines which approaches are viable (reasonably efficient, or even possible)
  – includes instruction set (synchronization, I/O, …)
  – also hardware components like MMU or DMA controllers
Architectural features affecting OS’s

- These hardware features were built primarily to support OS’s:
  - timer (clock) operation
  - synchronization instructions (e.g., atomic test-and-set)
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of execution (kernel vs. user)
  - privileged instructions
  - system calls (and software interrupts)
  - virtualization architectures
Privileged instructions

- Only the OS should be able to:
  - directly access I/O devices (disks, network cards)
    - why?
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
    - why?
  - manipulate special ‘mode bits’
    - interrupt priority level
    - why?

- But users can put any bit strings in memory they want
  - so they can execute the same instructions that the OS does

- So how can this work?
  - some instructions must be “restricted to the OS”
  - known as privileged instructions
OS protection

• So how does the processor know whether to allow execution of a privileged instruction?
  – the architecture must support at least two “privilege levels”: kernel and user
    • x86 supports 4 privilege levels
  – current level is given by status bits in a protected processor register
    • user programs execute in user mode (3, in xk)
    • OS executes in kernel (privileged) mode (0, in xk)

• The hardware assures that privileged instructions can be executed only when the core is at kernel privilege
  – what happens if code running in user mode attempts to execute a privileged instruction?
Crossing protection boundaries

• Q: So how does code running at user level (apps) do something privileged?
  – e.g., how can it write to a disk if it can’t execute the I/O instructions that are needed to do I/O?

• A: Ask code that can (the OS) to do it for you.

• User programs must cause execution of an OS
  – OS defines a set of system calls
  – App code leaves a bunch of arguments to the call somewhere the OS can find them
    • e.g., on the stack or in registers
  – One of the arguments is a name for which system call is being requested
    • usually a syscall number
  – App somehow causes processor to elevate its privilege level to 0
Elevating the CPU privilege level

• Syscall instruction
  – Like a protected procedure call
  – What’s protected?
    • The entry point
  – What about the arguments?
    • Are they valid?
      – Would assuming they are potentially cause an execution error while running the OS?
Dynamic View

HW state: user → privileged → user → privileged

Time:

CSE 451 #2 9
syscall/sysret instructions

• The syscall instruction atomically:
  – Saves the current (user) PC
  – Sets the execution mode to privileged
  – Sets the PC to a handler address (that was established by the OS during boot)

• The sysret instruction atomically:
  – Restores the previously saved user PC
  – Sets the execution mode to unprivileged
“Protected procedure call”

• Similar to local procedure call…
  – Caller puts arguments in a place callee expects (registers or stack)
  – Caller causes jump to OS by executing syscall instruction
    • The OS determines what address to start executing at, not the caller
    • One of the passed args is a syscall number, indicating which OS function to invoke
  – Callee (OS) saves caller’s state (registers, other control state) so it can use the CPU
  – OS function code runs
    • OS must verify caller’s arguments (e.g., pointers)
  – OS (mostly) restores caller’s state
  – OS returns by executing sysret instruction
    • Automatically sets PC to return address and sets execution mode to user
Firefox: read(int fileDescriptor, void *buffer, int numBytes)

Save user PC
PC = trap handler address
Enter kernel mode

Save app state
Verify syscall number
Find sys_read( ) handler in vector table

Verify args
Initiate read
Choose next process to run
Setup return values
Restore app state

SYSRET instruction

PC = saved PC
Enter user mode
System call issues

• What would be wrong if a syscall worked like a regular subroutine call, with the caller specifying the next PC?

• What would happen if kernel didn’t save state?

• Why must the kernel verify arguments?

• How can you reference kernel objects as arguments to or results from system calls?
  – What does that question mean?!
Exception Handling and Protection

• *All* entries to the OS occur via the mechanism just shown
  – Acquiring privileged mode and branching to the trap handler are inseparable

• Terminology:
  – **Interrupt**: asynchronous; caused by an external device
  – **Exception**: synchronous; unexpected problem with instruction
  – **Trap**: synchronous; intended transition to OS due to an instruction

• Privileged instructions and resources are the basis for most everything: memory protection, protected I/O, limiting user resource consumption, …
### x86 Interrupt/Trap Handling: Interrupt vector

<table>
<thead>
<tr>
<th>Vector</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
<th>Error Code</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>Fault</td>
<td>No</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug Exception</td>
<td>Fault/ Trap</td>
<td>No</td>
<td>Instruction, data, and I/O breakpoints; single-step; and others.</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>NMI Interrupt</td>
<td>Interrupt</td>
<td>No</td>
<td>Nonmaskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>Trap</td>
<td>No</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>Trap</td>
<td>No</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>Fault</td>
<td>No</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (Undefined Opcode)</td>
<td>Fault</td>
<td>No</td>
<td>UD2 instruction or reserved opcode.¹</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Abort</td>
<td>Yes</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Coprocessor Segment Overrun (reserved)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point instruction.²</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Fault</td>
<td>Yes</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Fault</td>
<td>Yes</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack-Segment Fault</td>
<td>Fault</td>
<td>Yes</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Fault</td>
<td>Yes</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Fault</td>
<td>Yes</td>
<td>Any memory reference.</td>
</tr>
</tbody>
</table>
x86 Interrupt/Trap Handling: Overview
x86 Interrupt/Trap Handling: Finding the IDT

Figure 6-1. Relationship of the IDTR and IDT
x86 Interrupt/Trap Handling: IDT entries

Figure 6-2. IDT Gate Descriptors
x86 Interrupt/Trap Handling: Segment Descriptors

Figure 3-8. Segment Descriptor
x86 Interrupt/Trap Handling: Stacks

![Figure 7-2: 32-Bit Task-State Segment (TSS)](image)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>I/O Map Base Address</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>96</td>
<td>LDT Segment Selector</td>
</tr>
<tr>
<td>92</td>
<td>GS</td>
</tr>
<tr>
<td>88</td>
<td>FS</td>
</tr>
<tr>
<td>84</td>
<td>DS</td>
</tr>
<tr>
<td>80</td>
<td>SS</td>
</tr>
<tr>
<td>76</td>
<td>CS</td>
</tr>
<tr>
<td>72</td>
<td>ES</td>
</tr>
<tr>
<td>68</td>
<td>EDI</td>
</tr>
<tr>
<td>64</td>
<td>ESI</td>
</tr>
<tr>
<td>60</td>
<td>EBP</td>
</tr>
<tr>
<td>56</td>
<td>ESP</td>
</tr>
<tr>
<td>52</td>
<td>EBX</td>
</tr>
<tr>
<td>48</td>
<td>EDX</td>
</tr>
<tr>
<td>44</td>
<td>ECX</td>
</tr>
<tr>
<td>40</td>
<td>EAX</td>
</tr>
<tr>
<td>36</td>
<td>EFLAGS</td>
</tr>
<tr>
<td>32</td>
<td>EIP</td>
</tr>
<tr>
<td>28</td>
<td>CR3 (PDBR)</td>
</tr>
<tr>
<td>24</td>
<td>SS2</td>
</tr>
<tr>
<td>20</td>
<td>ESP2</td>
</tr>
<tr>
<td>16</td>
<td>SS1</td>
</tr>
<tr>
<td>12</td>
<td>ESP1</td>
</tr>
<tr>
<td>8</td>
<td>SS0</td>
</tr>
<tr>
<td>4</td>
<td>ESP0</td>
</tr>
<tr>
<td>0</td>
<td>Previous Task Link</td>
</tr>
</tbody>
</table>

Figure 7-2: 32-Bit Task-State Segment (TSS)
Memory protection

• OS must protect user programs from each other
  – malice, bugs
• OS must also protect itself from user programs
  – integrity and security
  – what about protecting user programs from OS?
• Simplest scheme: base and limit registers
  – (Hey, segments!)
  – are these protected?

![Diagram of base and limit registers for programs A, B, and C.](image)

Base and limit registers are loaded by OS before starting program.
More sophisticated memory protection

• coming later in the course
  – also coming earlier in your course sequence!

• paging, segmentation, virtual memory
  – page tables, page table pointers
  – translation lookaside buffers (TLBs)
  – page fault handling
I/O control

• Issues:
  – how does the OS start an I/O?
    • special I/O instructions
    • memory-mapped I/O
  – how does the OS notice an I/O has finished?
    • polling
    • Interrupts
  – how does the OS exchange data with an I/O device?
    • Programmed I/O (PIO)
    • Direct Memory Access (DMA)
Asynchronous I/O

• Interrupts are the basis for asynchronous I/O
  – device performs an operation asynchronously to CPU
  – device sends an interrupt signal on bus when done
  – in memory, a vector table contains list of addresses of kernel routines to handle various interrupt types
    • who populates the vector table, and when?
  – CPU switches to address indicated by vector index specified by interrupt signal

• What’s the advantage of asynchronous I/O?
Timers

• How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
  – use a hardware timer that generates a periodic interrupt
  – before it transfers to a user program, the OS loads the timer with a time to interrupt
    • “quantum” – how big should it be set?
  – when timer fires, an interrupt transfers control back to OS
    • at which point OS must decide which program to schedule next
    • very interesting policy question: we’ll dedicate a class to it

• Should access to the timer be privileged?
  – for reading or for writing?
Synchronization

• Interrupts cause a wrinkle:
  – may occur any time, causing code to execute that interferes with code that was interrupted
  – OS must be able to synchronize concurrent processes

• Synchronization:
  – guarantee that short instruction sequences (e.g., read-modify-write) execute atomically
  – one method: turn off interrupts before the sequence, execute it, then re-enable interrupts
    • architecture must support disabling interrupts
      – Privileged???
  – another method: have special complex atomic instructions
    • read-modify-write
    • test-and-set
    • load-linked store-conditional
“Concurrent programming”

- Management of concurrency and asynchronous events is biggest difference between “systems programming” and “traditional application programming”
  - modern “event-oriented” application programming is a middle ground
  - And in a multi-core world, more and more apps have internal concurrency

- Arises from the architecture
  - Can be sugar-coated, but cannot be totally abstracted away

- Huge intellectual challenge
  - Unlike vulnerabilities due to buffer overruns, which are just sloppy programming
Architectures are still evolving

• New features are still being introduced to meet modern demands
  – Support for virtual machine monitors
  – Hardware transaction support (to simplify parallel programming)
  – Support for security (encryption, trusted modes)
  – Increasingly sophisticated video / graphics
  – Other stuff that hasn’t been invented yet…

• In current technology transistors are free – CPU makers are looking for new ways to use transistors to make their chips more desirable

• Intel’s big challenge: finding applications that require new hardware support, so that you will want to upgrade to a new computer to run them
Some questions

• Why wouldn’t you want a user program to be able to access an I/O device (e.g., the disk) directly?
  – Why would you?!

• OK, so what keeps this from happening? What prevents user programs from directly accessing the disk?

• How then does a user program cause disk I/O to occur?
Some questions

• What prevents a user program from scribbling on the memory of another user program?
  – Why might you want to allow it to?!

• What prevents a user program from scribbling on the memory of the operating system?

• What prevents a user program from over-writing its own instructions?
  – Why do you want to prevent that?
  – Why do you want to allow it?!

• What prevents a user program from running away with the CPU?