Lecture 11: Interrupts and Exceptions

Anton Burtsev
November, 2019
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    cprintf("\ncpu%d: starting xv6\n\n", cpunum());
    picinit(); // another interrupt controller
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
    pinit(); // process table
    tvinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    ideinit(); // disk
    if(!ismp)
        timerinit(); // uniprocessor timer
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    userinit(); // first user process
    mpmain(); // finish this processor’s setup
}
Why do we need interrupts?

Remember:
hardware interface is designed to help OS
Why do we need interrupts?

- Two main use cases:
  - **[Synchronous]** Something bad happened and OS needs to fix it
    - Program tries to access an unmapped page (OS maps the page if its on disk)
  - **[Asynchronous]** Notifications from external devices
    - Network packet arrived (OS will copy the packet from temporary buffer in memory (to avoid overflowing) and may switch to a process waiting on that packet)
    - Timer interrupt (OS may switch to another process)
- A third, special, use-case
  - **[It's also synchronous]** For many years an interrupt, e.g., int 0x80 instruction, was used as a mechanism to transfer control flow from user-level to kernel in a secure manner
    - In other words, to implement system calls
    - Now, a faster mechanism is available (sysenter)
How do we handle an interrupt?
Handling interrupts and exceptions

- In both synchronous and asynchronous cases the CPU follows the **same procedure**
  - Stop execution of the current program
  - Start execution of a handler
  - Processor accesses the handler through an entry in the Interrupt Descriptor Table (IDT)
- Each interrupt is defined by a number
  - E.g., 14 is pagefault, 3 debug
  - This number is an index into the interrupt table (IDT)
There might be two cases

- **Interrupt requires no change** of privilege level
  - i.e., the CPU runs kernel code (privilege level 0) when
    - a timer interrupt arrives, or
    - kernel tries to access an unmapped page

- **Interrupt changes** privilege level
  - i.e., the CPU runs **user** code (privilege level 3) when
    - a timer interrupt arrives, or
    - User code tries to access an unmapped page
Case #1: Interrupt path no change in privilege level

- e.g., we're already running in the kernel
Interrupt descriptor table (IDT)

- Is pointed by the IDTR register
  - Virtual address

- OS configures the value and loads it into the register (normally during boot)
Interrupt descriptor

Interrupt Gate

Offset 31..16

Segment Selector

Offset 15..0
Interrupt descriptor

- We will walk through these fields gradually
  - For now we care about vector offset
  - Pointer to the interrupt handler
Interrupt handlers

- Just plain old code in the kernel
- The IDT stores a pointer to the right handler routine
Processing of interrupt (same PL)

1. Push the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack
2. Push an error code (if appropriate) on the stack
3. Load the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers
4. If the call is through an interrupt gate, clear the IF flag in the EFLAGS register (disable further interrupts)
5. Begin execution of the handler
Interrupted Procedure’s and Handler’s Stack

Stack Usage with No Privilege-Level Change

- ESP Before Transfer to Handler
- ESP After Transfer to Handler
Interrupt path

Interrupt Vector #

Timer: IRQ0 -> vector 32

Kernel code

vector32

EBP

Kernel stack

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc
EFLAGS
CS
EIP
Error code

CS : #1
SS : #2
GDT: gdt
IDT: idt
EIP: <kernel>
ESP: <kernel>
TSS: tss
CR3: pt
Processing of interrupt (cross PL)

• Need to change privilege level...
Detour:
What are those privilege levels?
Recap: Can a process overwrite kernel memory?
Privilege levels

- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3
Privilege levels

- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3
Privilege levels

- Currently running code also has a privilege level
  - “Current privilege level” (CPL): 0-3
  - It is saved in the %cs register
    - It was loaded there when the descriptor for the currently running code was loaded into %cs
Privilege level transitions

• CPL can access only less privileged segments
  - E.g., 0 can access 1, 2, 3

• Some instructions are “privileged”
  • Can only be invoked at CPL = 0
  • Examples:
    - Load GDT
    - MOV <control register>
      - E.g. reload a page table by changing CR3
Xv6 example: started boot (no CPL yet)
Xv6 example: prepare to load GDT entry #1

ljmp 1, $start32
Privilege levels

- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3
How GDT is defined

9180  # Bootstrap GDT
9181  .p2align 2  # force 4 byte alignment
9182  gdt:
9183   SEG_NULLASM  # null seg
9184   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)  # code seg
9185   SEG_ASM(STA_W, 0x0, 0xffffffff)  # data seg
9186
9187  gdtdesc:
9188   .word (gdtdesc - gdt - 1)  # sizeof(gdt) - 1
9189   .long gdt
Now CPL=0. We run in the kernel.
iret: return to user, load GDT #4
Run in user, CPL=3

User stack of a process (can grow up to 2GBs)

Code, data, heap

Kernel Stack of a process (4K)

GDT
- NULL: 0x0
- KCODE: DPL=0, 0 - 4GB
- KDATA: DPL=0, 0 - 4GB
- K_CPU: DPL=0, 4 bytes
- CODE: DPL=3, 0 - 4GB
- DATA: DPL=3, 0 - 4GB
- TSS: sizeof(ts)

CS : CPL=4, 0-4GB  EIP:
SS : ESP:
GDT: gdt  TSS:
IDT: tdt  CR3:
Real world

- Only two privilege levels are used in modern OSes:
  - OS kernel runs at 0
  - User code runs at 3
- This is called “flat” segment model
  - Segments for both 0 and 3 cover entire address space
How GDT is initialized in xv6?
main(void) {
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
  lapicinit(); // interrupt controller
  seginit(); // segment descriptors
  cprintf("cpu%d: starting xv6\n\n", cpunum());
  picinit(); // another interrupt controller
  ioapicinit(); // another interrupt controller
  consoleinit(); // console hardware
  uartinit(); // serial port
  pinit(); // process table
  tvinit(); // trap vectors
  binit(); // buffer cache
  fileinit(); // file table
  ideinit(); // disk
  if(!ismp)
    timerinit(); // uniprocessor timer
  startothers(); // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit(); // first user process
  mpmain(); // finish this processor’s setup
}
1712 // Set up CPU's kernel segment descriptors.
1713 // Run once on entry on each CPU.
1714 void
1715 seginit(void)
1716 {
1717   struct cpu *c;
1718
1719   // Map "logical" addresses to virtual addresses using identity map.
1720   // Cannot share a CODE descriptor for both kernel and user
1721   // because it would have to have DPL_USR, but the CPU forbids
1722   // an interrupt from CPL=0 to DPL=3.
1723   c = &cpus[cpuid()];
1724   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1725   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1726   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1727   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1728   lgdt(c->gdt, sizeof(c->gdt));
1729 }
struct cpu {
    uchar apicid; // Local APIC ID
    struct context *scheduler; // swtch() here to enter scheduler
    struct taskstate ts; // Used by x86 to find stack for interrupt
    struct segdesc gdt[NSEGS]; // x86 global descriptor table
    volatile uint started; // Has the CPU started?
    int ncli; // Depth of pushcli nesting.
    int intena; // Were interrupts enabled before pushcli?
    struct proc *proc; // The process running on this cpu or null
};

extern struct cpu cpus[NCPU];
# Segment descriptor

<table>
<thead>
<tr>
<th>31</th>
<th>24 23 22 21 20 19</th>
<th>16 15 14 13 12 11</th>
<th>8 7</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 31:24</td>
<td>G</td>
<td>D / B</td>
<td>L</td>
<td>AVL</td>
<td>Seg. Limit 19:16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>16 15</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Address 15:00</td>
<td>Segment Limit 15:00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **L** — 64-bit code segment (IA-32e mode only)
- **AVL** — Available for use by system software
- **BASE** — Segment base address
- **D/B** — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- **DPL** — Descriptor privilege level
- **G** — Granularity
- **LIMIT** — Segment Limit
- **P** — Segment present
- **S** — Descriptor type (0 = system; 1 = code or data)
- **TYPE** — Segment type
Segment Descriptor

0724 // Segment Descriptor
0725 struct segdesc {
0726   uint lim_15_0 : 16; // Low bits of segment limit
0727   uint base_15_0 : 16; // Low bits of segment base address
0728   uint base_23_16 : 8; // Middle bits of segment base address
0729   uint type : 4;     // Segment type (see STS_ constants)
0730   uint s : 1;       // 0 = system, 1 = application
0731   uint dpl : 2;     // Descriptor Privilege Level
0732   uint p : 1;       // Present
0733   uint lim_19_16 : 4; // High bits of segment limit
0734   uintavl : 1;      // Unused (available for software use)
0735   uint rsv1 : 1;    // Reserved
0736   uint db : 1;      // 0 = 16-bit segment, 1 = 32-bit segment
0737   uint g : 1;       // Granularity: limit scaled by 4K when set
0738   uint base_31_24 : 8; // High bits of segment base address
0739 };
Real world

• Only two privilege levels are used in modern OSes:
  • OS kernel runs at 0
  • User code runs at 3
• This is called “flat” segment model
  • Segments for both 0 and 3 cover entire address space
• But then... how the kernel is protected?
Real world

• Only two privilege levels are used in modern OSes:
  • OS kernel runs at 0
  • User code runs at 3
• This is called “flat” segment model
  • Segments for both 0 and 3 cover entire address space
• But then... how the kernel is protected?
  • Page tables
Page table: user bit

- Each entry (both Level 1 and Level 2) has a bit
  - If set, code at privilege level 3 can access
  - If not, only levels 0-2 can access
- Note, only 2 levels, not 4 like with segments
- All kernel code is mapped with the user bit clear
  - This protects user-level code from accessing the kernel
End of detour:
Back to handling interrupts
Recap: interrupt path, no PL change

Kernel stack

Interrupt Vector #

Timer: IRQ0 -> vector 32

Kernel code
Processing of an interrupt when change of a privilege level is required
Processing of interrupt (cross PL)

• Assume we're at CPL = 3 (user)
Interrupt descriptor (an entry in the IDT)

Interrupt is allowed if...
- current privilege level (CPL) is less or equal to descriptor privilege level (DPL)
- The kernel protects device interrupts from user
Interrupt descriptor (an entry in the IDT)

- Note that this new segment can be more privileged
  - E.g., CPL = 3, DPL = 3, new segment can be PL = 0
  - This is how user-code (PL=3) transitions into kernel (PL=0)
Interrupt path

User stack of a process (can grow up to 2GBs)

Interrupt Vector #

Timer: IRQ0 -> vector 32

GDT

IDT

Page table
Level 1
Level 2

Kernel code

vector 32

CS : #4 (user)  EIP: <user>
SS : #5 (user)  ESP: <user>
GDT: gdt  TSS: tss
IDT: idt  CR3: pt
Can we continue on the same stack?
But how hardware knows where it is?
TSS: Task State Segment (yet another table)
Task State Segment

- Another magic control block
  - Pointed to by special task register (TR)
- Lots of fields for rarely-used features
- A feature we care about in a modern OS:
  - Location of kernel stack (fields SS/ESP)
    - Stack segment selector
    - Location of the stack in that segment
Processing of interrupt (cross PL)

1. Save ESP and SS in a CPU-internal register
2. Load SS and ESP from TSS
3. Push user SS, user ESP, user EFLAGS, user CS, user EIP onto new stack (kernel stack)
4. Set CS and EIP from IDT descriptor's segment selector and offset
5. If the call is through an interrupt gate clear interrupts enabled EFLAGS bit
6. Begin execution of a handler
Stack Usage with Privilege-Level Change

Interrupted Procedure’s Stack

ESP Before Transfer to Handler

Handler’s Stack

SS
ESP
EFLAGS
CS
EIP
Error Code

ESP After Transfer to Handler
Stack Usage with No Privilege-Level Change

Interrupted Procedure's and Handler's Stack

- EFLAGS
- CS
- EIP
- Error Code

ESP Before Transfer to Handler

ESP After Transfer to Handler
Complete interrupt path
Interrupt descriptor table (IDT)
x86 interrupt descriptor table

Device IRQs

Reserved for the CPU

Software Configurable
<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Any code or data reference.</td>
</tr>
<tr>
<td>2</td>
<td>NMI</td>
<td>NMI Interrupt</td>
<td>Non-maskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (UnDefined Opcode)</td>
<td>UD2 instruction or reserved opcode.</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td>#MF</td>
<td>CoProcessor Segment Overrun (reserved)</td>
<td>Floating-point instruction.</td>
</tr>
<tr>
<td>10</td>
<td>TS</td>
<td>Invalid TSS</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>NP</td>
<td>Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>SS</td>
<td>Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>GP</td>
<td>General Protection</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>PF</td>
<td>Page Fault</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>AC</td>
<td>Alignment Check</td>
<td>Any data reference in memory.</td>
</tr>
<tr>
<td>18</td>
<td>MC</td>
<td>Machine Check</td>
<td>Error codes (if any) and source are model dependent.</td>
</tr>
<tr>
<td>19</td>
<td>XM</td>
<td>SIMD Floating-Point Exception</td>
<td>SIMD Floating-Point Instruction.</td>
</tr>
<tr>
<td>20-31</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td></td>
<td>Maskable Interrupts</td>
<td>External interrupt from INTR pin or INT n instruction.</td>
</tr>
<tr>
<td>Vector No.</td>
<td>Mnemonic</td>
<td>Description</td>
<td>Source</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
<td>---------------------------------------------------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Any code or data reference.</td>
</tr>
<tr>
<td>2</td>
<td>#NMI</td>
<td>NMI Interrupt</td>
<td>Non-maskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (UnDefined Opcode)</td>
<td>UD2 instruction or reserved opcode.</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td>#MF</td>
<td>CoProcessor Segment Overrun (reserved)</td>
<td>Floating-point instruction.</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>#ME</td>
<td>Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Any data reference in memory.</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Error codes (if any) and source are model dependent.</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>SIMD Floating-Point Instruction.</td>
</tr>
<tr>
<td>20-31</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td></td>
<td>Maskable Interrupts</td>
<td>External interrupt from INTR pin or INT n instruction.</td>
</tr>
</tbody>
</table>
Interrupts

• Each type of interrupt is assigned an index from 0—255.
  • 0—31 are for processor interrupts fixed by Intel
  • E.g., 14 is always for page faults
• 32—255 are software configured
  • 32—47 are often used for device interrupts (IRQs)
  • Most device IRQ lines can be configured
  • Look up APICs for more info (Ch 4 of Bovet and Cesati)
• 0x80 issues system call in Linux
  • Xv6 uses 0x40 (64) for the system call
Disabling interrupts

• Delivery of interrupts can be disabled with IF (interrupt flag) in EFLAGS register

• There is a couple of exceptions
  • Synchronous interrupts cannot be disabled
    – It doesn't make sense to disable a page fault
    – INT n – cannot be masked as it is synchronous
  • Non-maskable interrupts (see next slide)
    – Interrupt #2 in the IDT
<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Any code or data reference.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>NMI Interrupt</td>
<td>Non-maskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (UnDefined Opcode)</td>
<td>UD2 instruction or reserved opcode.</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td>#MF</td>
<td>CoProcessor Segment Overrun (reserved)</td>
<td>Floating-point instruction.</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Any data reference in memory.</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Error codes (if any) and source are model dependent.</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>SIMD Floating-Point Instruction</td>
</tr>
<tr>
<td>20-31</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td></td>
<td>Maskable Interrupts</td>
<td>External interrupt from INTR pin or INT n instruction.</td>
</tr>
</tbody>
</table>
Nonmaskable interrupts (NMI)

- Delivered even if IF is clear, e.g. interrupts disabled
  - CPU blocks subsequent NMI interrupts until IRET
- Sources
  - External hardware asserts the NMI pin
  - Processor receives a message on the system bus, or the APIC serial bus with NMI delivery mode
- Delivered via vector #2
void tvinit(void)
{
    int i;

    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
             vectors[T_SYSCALL], DPL_USER);

    initlock(&tickslock, "time");
}
Initialize IDT

- System call interrupt vector (T_SYSCALL)

```c
3316  void
t3317  tvinit(void)
3318  {
3319    int i;
3320
3321    for(i = 0; i < 256; i++)
3322      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
3324      vectors[T_SYSCALL], DPL_USER);
3325    initlock(&tickslock, "time");
3326  }
```
Protection

• Generally user code cannot invoke int X
  • i.e., can't issue int 14 (a page fault)
  • OS configures the IDT in such a manner that invocation of all int X instructions besides 0x40 triggers a general protection fault exception
    – Interrupt vector 13
Remember this slide: interrupt descriptor (an entry in the IDT)

- Interrupt is allowed if...
  - current privilege level (CPL) is less or equal to descriptor privilege level (DPL)
  - The kernel protects device interrupts from user
void tvinit(void) {
    int i;

    for(i = 0; i < 256; i++)
      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

    initlock(&tickslock, "time");
}

Initialize IDT

• A couple of important details
void tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}

- Only int T_SYSCALL can be called from user-level
3316  void
tvinit(void)
3318  {
3319    int i;
3320
3321    for(i = 0; i < 256; i++)
3322      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3324
3325    initlock(&tickslock, "time");
3326  }

Initialize IDT

- Syscall is a “trap”
- i.e., doesn't disable interrupts
Interrupt path through the xv6 kernel
Complete interrupt path

User stack of a process (can grow up to 2GBs)

Code, data, heap

Timer: IRQ0 -> vector 32

Interrupt Vector #
Timer Interrupt (int 0x32)

vector32:

```
pushl $0    // error code
pushl $32   // vector #
jmp alltraps
```

- Automatically generated
- From vectors.pl
  - vector.S
Kernel stack after interrupt

User state (saved by hardware)

- SS
- ESP
- EFLAGS
- CS
- EIP
- 0
- 32

Kernel Stack of a process (4K)

ESP

Call stack: vector32()
alltraps:
3255  # Build trap frame.
3256  pushl %ds
3257  pushl %es
3258  pushl %fs
3259  pushl %gs
3260  pushal
3261
3262  # Set up data segments.
3263  movw $(SEG_KDATA<<3), %ax
3264  movw %ax, %ds
3265  movw %ax, %es
3266
3267  # Call trap(tf), where tf=%esp
3268  pushl %esp
3269  call trap
pusha

- An assembler instruction that saves all registers on the stack

- [https://c9x.me/x86/html/file_module_x86_id_270.html](https://c9x.me/x86/html/file_module_x86_id_270.html)

```
Temporary = ESP;
Push(EAX);
Push(ECX);
Push(EDX);
Push(EBX);
Push(Temporary);
Push(EBP);
Push(ESI);
Push(EDI);
```
Kernel stack after interrupt

User state (saved by hardware)

vector32

alltraps

ESP

Kernel Stack of a process (4K)

Call stack: vector32()
alltraps()

int 0x32
vector32
alltraps
alltraps:
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal

# Set up data and per-cpu segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
movw $(SEG_KCPU<<3), %ax
movw %ax, %fs
movw %ax, %gs

# Call trap(tf), where tf=%esp
pushl %esp
call trap
All interrupts, e.g. timer interrupt end up in a single function: `trap()`
alltraps(): exit from the interrupt
Stack after trap() returns

User state (saved by hardware)

vector32

alltraps

ESP

Kernel Stack of a process (4K)

Trap frame
alltraps(): exiting

- Restore all registers
- Exit into user
- iret
Return from an interrupt

• Starts with IRET

1. Restore the CS and EIP registers to their values prior to the interrupt or exception

2. Restore EFLAGS

3. Restore SS and ESP to their values prior to interrupt
   - This results in a stack switch

4. Resume execution of interrupted procedure
We're back to where we were when timer interrupt was raised
System Calls
(int 0x40)
Software interrupts can be used to implement system calls

- The int N instruction provides a secure mechanism for kernel invocation
  - i.e., user can enter the kernel
  - But through a well-defined entry point
    - System call handler
- Xv6 uses vector 0x40 (or 64)
  - You can choose any other unused vector
  - Linux uses 0x80
    - Well now it uses sysenter instead of int 0x80 as it is faster
Where does IDT (entry 64) point to?

vector64:
```
pushl $0    // error code
pushl $64   // vector #
jmp alltraps
```

- Automatically generated
- From vectors.pl
  - vector.S
Kernel stack inside system call

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- EIP
- 0
- 32

Kernel Stack of a process (4K)

ESP

int 0x64

vector64

Call stack: vector32()
alltraps:
3255  # Build trap frame.
3256  pushl %ds
3257  pushl %es
3258  pushl %fs
3259  pushl %gs
3260  pushal
3261
3262  # Set up data and per-cpu segments.
3263  movw $(SEG_KDATA<<3), %ax
3264  movw %ax, %ds
3265  movw %ax, %es
3266  movw $(SEG_KCPU<<3), %ax
3267  movw %ax, %fs
3268  movw %ax, %gs
3269
3270  # Call trap(tf), where tf=%esp
3271  pushl %esp
3272  call trap
Kernel stack inside system call

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- EIP
- 0
- 32
- DS
- ES
- FS
- GS
- All registers
- ESP

Kernel Stack of a process (4K)

Trap frame

Call stack:
- vector32()
- alltraps()

int 0x32
- vector32
- alltraps
3351 trap(struct trapframe *tf)
3352 {
3353   if(tf->trapno == T_SYSCALL){
3354     if(proc->killed)
3355       exit();
3356     proc->tf = tf;
3357     syscall();
3358     if(proc->killed)
3359       exit();
3360     return;
3361   }
3362
3363   switch(tf->trapno){
3364   case T_IRQ0 + IRQ_TIMER:
3365     int 0x64
3366     vector64
3367     alltraps
3368     trap(*tf)
Syscall number

- System call number is passed in the %eax register
  - To distinguish which syscall to invoke,
    - e.g., sys_read, sys_exec, etc.
- alltrap() saves it along with all other registers
syscall(void) {
    int num;
    num = proc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        proc->tf->eax = syscalls[num]();
    } else {
        cprintf("%d %s: unknown sys call %d\n", 
        proc->pid, proc->name, num);
        proc->tf->eax = -1;
    }
}
syscall(): process a syscall from the table

```c
3625 syscall(void)  
3626 {  
3627   int num;  
3628  
3629   num = proc->tf->eax;  
3630   if(num > 0 && num < NELEM(syscalls) && syscalls[num])  
3631     {  
3632       proc->tf->eax = syscalls[num]();  
3633     } else {  
3634       cprintf("%d %s: unknown sys call %d
",  
3635       proc->pid, proc->name, num);  
3636       proc->tf->eax = -1;  
3637     }  
3638 }  
```
static int (*syscalls[])(void) = {
    SYS_fork sys_fork,
    SYS_exit sys_exit,
    SYS_wait sys_wait,
    SYS_pipe sys_pipe,
    SYS_read sys_read,
    SYS_kill sys_kill,
    SYS_exec sys_exec,
    SYS_fstat sys_fstat,
    SYS_chdir sys_chdir,
    SYS_dup sys_dup,
    SYS_getpid sys_getpid,
    SYS_sbrk sys_sbrk,
    SYS_sleep sys_sleep,
    SYS_uptime sys_uptime,
    SYS_open sys_open,
    SYS_write sys_write,
    SYS_mknod sys_mknod,
    SYS_unlink sys_unlink,
    SYS_link sys_link,
    SYS_mkdir sys_mkdir,
    SYS_close sys_close,
};
How do user programs access system calls?

• It would be weird to write
  
  8410 pushl $argv
  8411 pushl $init
  8412 pushl $0 // where caller pc would be
  8413 movl $SYS_exec, %eax
  8414 int $T_SYSCALL

• … every time we want to invoke a system call
• This is an example for the exec() system call
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
...

user.h
• user.h defines system call prototypes
• Compiler can generate correct system call stacks
  • Remember calling conventions?
  • Arguments on the stack
Example

- From cat.asm

- if (write(1, buf, n) != n)

  A3:  53  push    ebx
  a4:  68 00 0b 00 00  push 0xb00
  a9:  6a 01  push  0x1
  ab:  e8 c2 02 00 00  call 372 <write>
• Note, different versions of gcc
  • and different optimization levels
• Will generate slightly different code
Example

- From cat.asm

```assembly
if (write(1, buf, n) != n)

a0:   89 5c 24 08             mov    %ebx,0x8(%esp)

a4:   c7 44 24 04 00 0b 00    movl   $0xb00,0x4(%esp)

ab:    00

ac:    c7 04 24 01 00 00 00    movl   $0x1,(%esp)

b3:    e8 aa 02 00 00         call    362 <write>
```
Example

- From cat.asm

  `if (write(1, buf, n) != n)`

  ```asm
  a0: 89 5c 24 08     mov    %ebx,0x8(%esp)
  a4: c7 44 24 04 00 0b 00  movl   $0xb00,0x4(%esp)
  ab: 00
  ac: c7 04 24 01 00 00 00    movl   $0x1,(%esp)
  b3: e8 aa 02 00 00     call    362 <write>
  ```
Example

- From cat.asm

- if (write(1, buf, n) != n)

  a0:  89 5c 24 08         mov    %ebx,0x8(%esp)
  a4:  c7 44 24 04 00 0b 00 movl    $0xb00,0x4(%esp)
  ab:  00
  ac:  c7 04 24 01 00 00 00 movl    $0x1,(%esp)
  b3:  e8 aa 02 00 00      call    362 <write>
• Still not clear...
  • The header file allows compiler to generate a call side invocation,
    - e.g., push arguments on the stack
  • But where is the system call invocation itself
    - e.g., `int $T_SYSCALL`
• Xv6 uses a SYSCALL macro to define a function for each system call invocation
  • E.g., fork() to invoke the “fork” system call
Example

- Write system call from cat.asm

000000362 <write>:

SYSCALL(write)

362:   b8 10 00 00 00          mov    $0x10,%eax
367:   cd 40                   int    $0x40
369:   c3                      ret
System call arguments

- Where are the system call arguments?
- How does kernel access them?
  - And returns results?
Example

- Write system call

- if (write(1, buf, n) != n)

```c
5876 int
5877 sys_write(void)
5878 {
5879   struct file *f;
5880   int n;
5881   char *p;
5882
5883   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5884     return -1;
5885   return filewrite(f, p, n);
5886 }
```
Example

- Write system call
  - `if (write(1, buf, n) != n)`

    5876 int
    5877 sys_write(void)
    5878 {
    5879   struct file *f;
    5880   int n;
    5881   char *p;
    5882
    5883   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    5884     return -1;
    5885   return filewrite(f, p, n);
    5886 }
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}

fetchint(uint addr, int *ip)
{
    if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }

3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }

argint(int n, int *ip)
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547    return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }

3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519    if(addr >= proc->sz || addr+4 > proc->sz)
3520        return -1;
3521    *ip = *(int*)(addr);
3522    return 0;
3523 }

• Start with the address where current user stack is (esp)
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }

3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }

• Skip return eip
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }

3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }

- Fetch n'th argument

argint(int n, int *ip)
3543  // Fetch the nth 32-bit system call argument.
3544  int
3545  argint(int n, int *ip)
3546  {
3547    return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548  }

3515  // Fetch the int at addr from the current process.
3516  int
3517  fetchint(uint addr, int *ip)
3518  {
3519    if(addr >= proc->sz || addr+4 > proc->sz)
3520      return -1;
3521    *ip = *(int*)(addr);
3522    return 0;
3523  }
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }

3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }

fetchint(uint addr, int *ip)
Any idea for what argptr() shall do?

- Write system call
  - `if (write(1, buf, n) != n)`

```c
int sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return −1;
  return filewrite(f, p, n);
}
```

- Remember, buf is a pointer to a region of memory
  - i.e., a buffer
  - of size n
3550 // Fetch the nth word-sized system call argument as a pointer
3551 // to a block of memory of size n bytes. Check that the pointer
3552 // lies within the process address space.
3553 int
3554 argptr(int n, char **pp, int size)
3555 {
3556   int i;
3557
3558   if(argint(n, &i) < 0)
3559     return −1;
3560   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3561     return −1;
3562   *pp = (char*)i;
3563   return 0;
3564 }
argptr(uint addr, int *ip)

3550 // Fetch the nth word-sized system call argument as a pointer
3551 // to a block of memory of size n bytes. Check that the pointer
3552 // lies within the process address space.

3553 int
3554 argptr(int n, char **pp, int size)
3555 {
3556   int i;
3557
3558   if(argint(n, &i) < 0)
3559     return −1;
3560   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3561     return −1;
3562   *pp = (char*)i;
3563   return 0;
3564 }

argptr(uint addr, int *ip)

• Check that the
  buffer is in user
  memory
Summary

• We've learned how system calls work
Thank you