Copyright Notice

- These slides are distributed under the Creative Commons Attribution 3.0 License

- You are free:
  - to share—to copy, distribute and transmit the work
  - to remix—to adapt the work

- under the following conditions:
  - Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows: “Courtesy of Mark P. Jones, Portland State University”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
General theme for the next two weeks

• In a complex system …

• Question: how can we protect individual programs from interference with themselves, or with one another, either directly or by subverting lower layers?

• General approach: leverage programmable hardware features!

Diagrams and Code

• There are a lot of diagrams on these slides
  • Many of these are taken directly from the “Intel® 64 and IA-32 Architectures Software Developer’s Manual”, particularly Volume 3
  • There is a link to the full pdf file in the Reference section
• There is also a lot of code on these slides
• Remember that you can study these more carefully later if you need to!
### Taking stock: Code samples ... so far

<table>
<thead>
<tr>
<th>vram</th>
<th>video RAM simulation</th>
<th>vram.tar.gz</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>boot and say hello on bare metal, via GRUB</td>
<td>hello.tar.gz</td>
</tr>
<tr>
<td>simpleio</td>
<td>a simple library for video RAM I/O</td>
<td>baremetal.tar.gz</td>
</tr>
<tr>
<td>bootinfo</td>
<td>display basic boot information from GRUB</td>
<td></td>
</tr>
<tr>
<td>mimg</td>
<td>memory image bootloader &amp; make tool</td>
<td></td>
</tr>
<tr>
<td>example-mimg</td>
<td>display basic boot information from mimgload</td>
<td></td>
</tr>
<tr>
<td>example-gdt</td>
<td>basic demo using protected mode segments (via a Global Descriptor Table)</td>
<td>prot.tar.gz</td>
</tr>
<tr>
<td>example-idt</td>
<td>context switching to user mode (via an Interrupt Descriptor Table)</td>
<td></td>
</tr>
</tbody>
</table>

### Segmentation
(or: where do “seg faults” come from?)
Breaking the 64KB barrier …

• The 8086 and 8088 CPUs in the original IBM PCs were 16 bit processors: in principle, they could only address 64KB.

• Intel used \textit{segmentation} to increase the amount of addressable memory from 64KB to 1MB:

\begin{align*}
16 \text{ bit segment} + 16 \text{ bit address} &= 20 \text{ bit address} \\
&= \text{Not an Intel diagram!}
\end{align*}

3.4.1 General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

• Operands for logical and arithmetic operations
• Operands for address calculations
• Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for another purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

\textbf{Figure 3-4. General System and Application Programming Registers}

\begin{center}
\includegraphics[width=\textwidth]{figure3-4.png}
\end{center}

\textbf{An Intel diagram!}
How are segments chosen

• The default choice of segment register is determined by the specific kind of address that is being used:

<table>
<thead>
<tr>
<th>Reference Type</th>
<th>Register Used</th>
<th>Segment Used</th>
<th>Default Selection Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>CS</td>
<td>Code Segment</td>
<td>All instruction fetches.</td>
</tr>
<tr>
<td>Stack</td>
<td>SS</td>
<td>Stack Segment</td>
<td>All stack pushes and pops.</td>
</tr>
<tr>
<td>Local Data</td>
<td>DS</td>
<td>Data Segment</td>
<td>Any memory reference which uses the ESP or EBP register as a base register.</td>
</tr>
<tr>
<td>Destination Strings</td>
<td>ES</td>
<td>Data Segment</td>
<td>Destination of string instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pointed with the ES register</td>
<td></td>
</tr>
</tbody>
</table>

• If a different segment register is required, a single byte “segment prefix” can be attached to the start of the instruction.

Table 3-5. Default Segment Selection Rules

Back to breaking the 64KB barrier …

640KB User Memory

- Programs can be organized to use multiple segments:
- For example:
  - One segment for the stack
  - One segment for code
  - One segment for data
- We can relocate these segments to different physical addresses, just by adjusting the segment registers.
Back to breaking the 64KB barrier …

640KB User Memory

BIOS, Video RAM, etc..

- Programs can be organized to use multiple segments:
  - For example:
    - One segment for the stack
    - One segment for code
    - One segment for data
  - We can relocate these segments to different physical addresses, just by adjusting the segment registers

Variations on the theme

- Programs can have multiple code and data segments
  - Programmers could use a standard “memory model”
  - Or use custom approaches to suit a specific application
- The machine provides special “far call” and “far jump” instructions that change CS and EIP simultaneously, allowing control transfers between distinct code segments
- There are six segment registers, so programs can have up to 6 active segments at a time (and more by loading new values in to the segment registers)
- Segments do not have to be exactly 64KB
- If segments do not overlap, then a stack overflow will not corrupt the contents of other segments - protection!
Accommodating multiple programs

640KB User Memory

- BIOS, Video RAM, etc..

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>B</th>
<th>C</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>SS</td>
<td>CS</td>
<td>DS</td>
<td>CS</td>
<td>DS</td>
<td>SS</td>
<td>SS</td>
<td>DS</td>
</tr>
</tbody>
</table>

• Now we can have multiple programs in memory at the same time, each with distinct code, data, and stack segments

• But what is to stop the code for one program from accessing and/or changing the data for another?

• Nothing!

• We would like to “protect” programs for interfering with one another, either by accident or design …

Protection!

- Ring 0 is sometimes called “supervisor” or “kernel mode”
- Ring 3 is often called “user mode”
The current mode

- The current mode is saved in the two least significant bits of the CS register.
- The value in CS can only be changed by a limited set of instructions (e.g., it cannot be the target of a movw), each of which performs a privilege check, if necessary, triggering a CPU exception if a violation occurs.
- End result: user mode code cannot change its own privilege level to move out of Ring 3!
Segments in protected mode

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

Figure 3-8. Segment Descriptor

Figure 5-1. Descriptor Fields Used for Protection
Segment registers hold segment selectors

![Segment Selector Diagram](image)

Figure 3-6. Segment Selector

![Logical Address to Linear Address Translation Diagram](image)

Figure 3-5. Logical Address to Linear Address Translation

The descriptor cache

![Segment Registers Table](image)

Figure 3-7. Segment Registers
Achieving protection

- The global and local descriptor tables are created by the kernel and cannot be changed by user mode programs.

- The CPU raises an exception if a user mode program attempts to access:
  - a segment index outside the bounds of the GDT or LDT
  - a segment that is not marked for user mode access
  - an address beyond the limit of the associated segment

- The kernel can associate a different LDT with each process, providing each process with a distinct set of segments.
Segments and capabilities

- The GDT and LDT for a given user mode program determine precisely which regions of memory that program can access.
- As such, these entries are an example of a capability mechanism.
- The user mode program refers to segments by their index in one of these tables, but it has no access to the table itself:
  - It cannot, in general, determine which regions of physical memory they are accessing.
  - It cannot “fake” access to other regions of memory.
- **The principle of least privilege**: limit access to the minimal set of resources that are required to perform a task.

What if we don’t want to use segments?

- Segmentation cannot be disabled in protected mode.
- But we can come pretty close by using segments with:
  - base address 0
  - length = 4GB
- A common GDT structure: (e.g., in Linux, etc., with no LDT)

```
<table>
<thead>
<tr>
<th>Segment</th>
<th>DPL = 0</th>
<th>DPL = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSS segment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KERNEL_CS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KERNEL_DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER_CS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER_DS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- all span the 4GB address space.

Intel Reqd
Storage for the GDT

```
.globl  gdt
.set   GDT_ENTRIES, 8
.set   GDT_SIZE, 8*GDT_ENTRIES  # 8 bytes for each descriptor
.data
.align 128
.gdt:  .space  GDT_SIZE, 0
.align 8
.gdtpt: .short GDT_SIZE-1
.long gdt
```

```
ready to begin?
lgdt gdtpt
```

Calculating GDT descriptors

```
.macro  gdtset name, slot, base, limit, gran, dpl, type

.set   \name, \(\slot<3) | \dpl
.globl \name
movl  $\base, $eax  # \eax = bhi # bmd # blo
movl  $\limit, $ebx  # \ebx = ~ # lhi # llo
mov   $\eax, %edx  # \edx = base
shl   $16, %eax  # \eax = blo # 0
mov   %bx, %ax  # \eax = blo # llo
movl  %eax, gdt+(8*\slot)
shr   $16, %edx  # \edx = 0 # bhi # bmd
mov   %edx, %ecx  # \ecx = 0 # bhi # bmd
andl  $0xff, %ecx  # \ecx = 0 # 0 # bmd
xorl  %ecx, %edx  # \edx = 0 # bhi # bmd
shr   $16, %edx  # \edx = bhi # 0
orl   %ecx, %edx  # \edx = bhi # 0 # bmd
andl  $0xf0000, %ebx  # \ebx = 0 # lhi # 0
orl   %ebx, %edx  # \edx = bhi # 0 # lhi # 0
orl   $((\gran<15) | 0x4080 | (\dpl<5) | \type)<<8), %edx
movl  %edx, gdt + (4 + 8*\slot)
.endm
```
Deja vu?

```assembly
movl    base, %eax
movl    limit, %ebx
mov     %eax, %edx
shl     $16, %eax
mov     %bx, %ax
movl    %eax, low
shr     $16, %edx
mov     %edx, %ecx
andl    $0xff, %ecx
xorl    %ecx, %edx
shl     $16, %edx
orl     %ecx, %edx
andl    $0xf0000, %ebx
orl     %ebx, %edx
orl     $0x503200, %edx
movl    %edx, high
```
Activating the GDT

```
lgdt gadget
ljmp $KERN_CS, $1f        # load code segment
1:
  mov $KERN_DS, %ax         # load data segments
  mov %ax, %ds
  mov %ax, %es
  mov %ax, %ss
  mov %ax, %gs
  mov %ax, %fs
  mov $TSS, %ax             # load task register
  ltr %ax
  ret
```

The Task State Segment

Figure 7-1. Structure of a Task
The Task State Segment

The processor updates dynamic fields when a task is suspended during a task switch. The following are dynamic fields:

- General-purpose register fields — State of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers prior to the task switch.
- Segment selector fields — Segment selectors stored in the ES, CS, SS, DS, FS, and GS registers prior to the task switch.
- EFLAGS register field — State of the EFLAGS register prior to the task switch.
- EIP (instruction pointer) field — State of the EIP register prior to the task switch.
- Previous task link field — Contains the segment selector for the TSS of the previous task (updated on a task switch that was initiated by a call, interrupt, or exception). This field (which is sometimes called the back link field) permits a task switch back to the previous task by using the IRET instruction.

The processor reads the static fields, but does not normally change them. These fields are set up when a task is created. The following are static fields:

- LDT segment selector field — Contains the segment selector for the task’s LDT.

Implementing the TSS

```assembly
.data
tss: .short 0, RESERVED            # previous task link
esp0: .long 0                      # esp0
.short KERN_DS, RESERVED           # ss0
.long 0                           # esp1
.short 0, RESERVED                 # ss1
.long 0                           # esp2
.short 0, RESERVED                 # ss2
.long 0, 0, 0                      # cr3 (pdbr), eip, eflags
.long 0, 0, 0, 0, 0                # eax, ecx, edx, ebx, esp
.long 0, 0, 0                       # ebp, esi, edi
.short 0, RESERVED                 # es
.short 0, RESERVED                 # cs
.short 0, RESERVED                 # ss
.short 0, RESERVED                 # ds
.short 0, RESERVED                 # fs
.short 0, RESERVED                 # gs
.short 0, RESERVED                 # ldt segment selector
.short 0                           # T bit
.short 1000                        # I/O bit map base address
.set tss_len, .-tss
```

Figure 7.2. 32-Bit Task-State Segment (TSS)
Exceptions

• What happens if the program you run on a conventional desktop computer attempts:
  • division by zero?
  • to use an invalid segment selector?
  • to reference memory beyond the limits of a segment?
  • etc…
• What happens when there is no operating system to catch you?
### Table 6-1. Protected-Mode Exceptions and Interrupts

<table>
<thead>
<tr>
<th>Vector No.</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>Yes</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>RESERVED</td>
<td>Fault/ Trap</td>
<td>No</td>
<td>For Intel use only.</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.</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 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 access 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 protection checks.</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>
<tr>
<td>15</td>
<td>—</td>
<td>(Intel reserved. Do not use.)</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>x87 FPU Floating-Point Error (Math Fault)</td>
<td>Fault</td>
<td>No</td>
<td>x87 FPU floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Fault</td>
<td>Yes</td>
<td>Any data reference in memory.3</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Abort</td>
<td>No</td>
<td>Error codes (if any) and source are model dependent.4</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>Fault</td>
<td>No</td>
<td>SSE/SE2/55SE3 floating-point instructions.5</td>
</tr>
<tr>
<td>20</td>
<td>#VE</td>
<td>Virtualization Exception</td>
<td>Fault</td>
<td>No</td>
<td>EPT violations6</td>
</tr>
<tr>
<td>21-31</td>
<td>—</td>
<td>Intel reserved. Do not use.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td>—</td>
<td>User Defined (Non-reserved) Inteptions</td>
<td>Interrupt</td>
<td></td>
<td>External interrupt or INT n instruction.</td>
</tr>
</tbody>
</table>

**Hardware and software interrupts**

- **Hardware**: devices often generate interrupt signals to inform the kernel that a certain event has occurred:
  - a timer has fired
  - a key has been pressed
  - a buffer of data has been transferred
  - ...

- **Software**: User programs often request services from an underlying operating system:
  - read data from a file
  - terminate this program
  - send a message
  - ...

*These can all be handled in the same way …*
INTERRUPT AND EXCEPTION HANDLING

When the processor performs a call to the exception- or interrupt-handler procedure:

• If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
  a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
  b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figures 6-4).
  c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.

• If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
  a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figures 6-4).
  b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

The interrupt vector

![Diagram of interrupt vector and IDT gate descriptors]

**Figure 6-3. Interrupt Procedure Call**

**Figure 6-2. IDT Gate Descriptors**

<table>
<thead>
<tr>
<th>Task Gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 16 15 14 13 12 8 7 0</td>
</tr>
<tr>
<td>Offset 4</td>
</tr>
<tr>
<td>TSS Segment Selector 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interrupt Gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 16 15 14 13 12 8 7 5 4 0</td>
</tr>
<tr>
<td>Offset 31..16 4</td>
</tr>
<tr>
<td>Segment Selector 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trap Gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 16 15 14 13 12 8 7 5 4 0</td>
</tr>
<tr>
<td>Offset 31..16 4</td>
</tr>
<tr>
<td>Segment Selector 0</td>
</tr>
</tbody>
</table>

DPL  Descriptor Privilege Level
Offset  Offset to procedure entry point
P    Segment Present flag
Selector  Segment Selector for destination code segment
D    Size of gate: 1 = 32 bits; 0 = 16 bits
Reserved
### Calculating IDT descriptors

```assembly
.macro idtcalc handler, slot, dpi=0, type=IDT_INTR, seg=KERN_CS
  # type = 0x000 (IDT_INTR) => interrupt gate
  # type = 0x100 (IDT_TRAP) => trap gate
  # The following comments use # for concatenation of bitdata
  #
  mov $\seg$, %ax    # eax = 7 # seg
  shl $16, %eax     # eax = seg # 0
  movl $\handler$, %edx # edx = hhi # hlo
  mov %dx, %ax      # eax = seg # hlo
  mov $(0x8e00 | (\dpi<<13) | \type), %dx
  movl %eax, idt + (8*\slot)
  movl %edx, idt + (4 + 8*\slot)
.endm
```

### Storage for the IDT

```assembly
.data
.align 8
{idt: .space IDT_SIZE, 0}
{idtptr: .short IDT_SIZE-1}
{long idt

.ready to begin?
{lidt idtptr

Calculating IDT descriptors
```

### INTERRUPT AND EXCEPTION HANDLING

The processor handles calls to exception- and interrupt-handlers similar to the way it handles calls with a CALL instruction to a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the IDT. If the index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler in a manner similar to a CALL to a call gate (see Section 5.8.2, "Gate Descriptors," through Section 5.8.6, "Returning from a Called Procedure"). If index points to a task gate, the processor executes a task switch to the exception- or interrupt-handler task in a manner similar to a CALL to a task gate (see Section 7.3, "Task Switching").

### 6.12 Exception and Interrupt Handling

An interrupt gate or trap gate references an exception- or interrupt-handler procedure that runs in the context of the currently executing task (see Figure 6-3). The segment selector for the gate points to a segment descriptor for an executable code segment in either the GDT or the current LDT. The offset field of the gate descriptor points to the beginning of the exception- or interrupt-handling procedure.

**Figure 6-2. IDT Gate Descriptors**

<table>
<thead>
<tr>
<th>Offset 31..16</th>
<th>P</th>
<th>D</th>
<th><code>1</code></th>
<th><code>0</code></th>
<th>D</th>
<th><code>1</code></th>
<th><code>0</code></th>
<th><code>0</code></th>
<th><code>0</code></th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset 15..0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**Segment Selector**

**Offset 15..0**
Initializing and activating the IDT

initIDT: # Fill in IDT entries

# Add descriptors for protected mode exceptions:
.irp num, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,16,17,18,19
.idtcalc exc\num, slot=\num
.endr

# Add descriptors for hardware irqs:
# ... except there aren't any here (yet)

# Add descriptors for system calls:
# These are the only idt entries that we will allow to be
# called from user mode without generating a general
# protection fault, so they are tagged with dpl=3.
.idtcalc handler=kputc, slot=0x80, dpl=3

# Install the new IDT:
.lidt idtptr
.ret

Transferring control to a handler

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines
Contexts

```
struct Iret {
    unsigned error;
    unsigned eip;
    unsigned cs;
    unsigned eflags;
    unsigned esp;
    unsigned ss;
};
```

Automatic (CPU)
Contexts

```
 struct Segments {
     unsigned ds;
     unsigned es;
     unsigned fs;
     unsigned gs;
 }; 
```

```
 struct Iret {
     unsigned error;
     unsigned eip;
     unsigned cs;
     unsigned eflags;
     unsigned esp;
     unsigned ss;
 }; 
```

```
 push %gs
 push %fs
 push %es
 push %ds
```

Automatic (CPU)

```
 pusha 
```

Contexts

```
 struct Registers {
     unsigned edi;
     unsigned esi;
     unsigned ebp;
     unsigned esp;
     unsigned ebx;
     unsigned edx;
     unsigned ecx;
     unsigned eax;
 }; 
```

```
 struct Segments {
     unsigned ds;
     unsigned es;
     unsigned fs;
     unsigned gs;
 }; 
```

```
 struct Iret {
     unsigned error;
     unsigned eip;
     unsigned cs;
     unsigned eflags;
     unsigned esp;
     unsigned ss;
 }; 
```

```
 push %gs
 push %fs
 push %es
 push %ds
```

Automatic (CPU)
Interrupts are typically generated by a system-based interrupt controller (8259A), with the interrupts being signaled through the system bus (Pentium 4, Intel Core Duo, P6 family, and Pentium processors) or the APIC serial bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family). When a system interrupt has occurred. The processor reads from the INTR pin or through the local APIC. The primary interrupt pins on Pentium 4, Intel Xeon, P6 family, and Pentium processors are the LINT[1:0] pins, which are connected to the on-chip local APIC. These processors have dedicated NM system bus the interrupt vector number provided by an external interrupt. The LINT[1:0] pins are not available on the Intel486 processor and earlier Pentium processors that do not contain an on-chip local APIC, the APIC indicates the receipt of an illegal vector.

Any external interrupt that is delivered to the processor by means of the INTR pin or through the local APIC is called a non-maskable interrupt (NMI). The primary interrupt pins through the INTR pin include the I and INTR pins. With these I and INTR pins, external interrupts can be delivered to the processor in order.

The processor's local APIC is normally connected to a system-based I/O APIC. Here, external interrupts are received through pins on the processor or through the local APIC. The primary interrupt pins are configured as INTR and NMI pins, respectively. If disabled, the LINT[1:0] pins must be connected to the INTR or NMI pins of the APIC. When the local APIC is global/hardware disabled, these pins are not handled by the interrupt and exception mechanism.

### Table 6-1. Protected-Mode Exceptions and Interrupts

<table>
<thead>
<tr>
<th>Vector No.</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>RESERVED</td>
<td>Fault/ Trap</td>
<td>No</td>
<td>For Intel use only.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>NMI Interrupt</td>
<td>Trap</td>
<td>No</td>
<td>Nonmaskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#GP</td>
<td>Breakpoint</td>
<td>Trap</td>
<td>No</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#DF</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.</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 (Zero)</td>
<td>Any instruction that can generate 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.</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 access 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 protection checks.</td>
</tr>
<tr>
<td>13</td>
<td></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>
<tr>
<td>15</td>
<td></td>
<td>(Intel reserved. Do not use.)</td>
<td>Fault</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>x87 FPU Floating-Point Error (Math Fault)</td>
<td>Fault</td>
<td>No</td>
<td>x87 FPU floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Fault</td>
<td>Yes (Zero)</td>
<td>Any data reference in memory.</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Abort</td>
<td>No</td>
<td>Error codes (if any) and source are model dependent.</td>
</tr>
<tr>
<td>19</td>
<td>#MX</td>
<td>SIMD Floating-Point Exception</td>
<td>Fault</td>
<td>No</td>
<td>SSE/SSE2/SSE3 floating-point instructions.</td>
</tr>
<tr>
<td>20</td>
<td>#VE</td>
<td>Virtualization Exception</td>
<td>Fault</td>
<td>No</td>
<td>EPT violations.</td>
</tr>
<tr>
<td>21-31</td>
<td></td>
<td>(Intel reserved. Do not use.)</td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>32-255</td>
<td>#DF</td>
<td>User Defined (Non-reserved) Interrupt</td>
<td></td>
<td></td>
<td>External interrupt or INT n instruction.</td>
</tr>
</tbody>
</table>

### Notes

- **Faults** can generally be corrected, restarting the program at the faulting instruction.
- **Traps** allow execution to be restarted after the trapping instruction.
- **Aborts** do not allow a restart.
### Exception handler

```
.macro handler num, func, errorcode=0
exc\num:.if \errorcode==0
  subl $4, %esp
  # Fake an error code if necessary
.endif
push %gs
  # Save segments
push %fs
push %es
push %ds
pusha
  # Save registers
push %esp
  # Push pointer to frame for handler
movl $\num, %eax
call \func
addl $4, %esp
  # Call func(struct Context *esp) with num in eax
popa
  # Restore registers
popl %ds
  # Restore segments
popl %es
popl %fs
popl %gs
addl $4, %esp
  # Remove error code
iret
.endm
```

### Some exceptions do not generate an error code ...

- Some exceptions do not generate an error code ...

### Defining a family of (non) handlers

```
# Protected-mode exceptions and interrupts:
#
handler num=0, func=nohandler
  # divide error
handler num=1, func=nohandler
  # debug
handler num=2, func=nohandler
  # NMI
handler num=3, func=nohandler
  # breakpoint
handler num=4, func=nohandler
  # overflow
handler num=5, func=nohandler
  # bound
handler num=6, func=nohandler
  # undefined opcode
handler num=7, func=nohandler
  # nomath
handler num=8, func=nohandler, errorcode=1
  # doublefault
handler num=9, func=nohandler
  # coproc seg overrun
handler num=10, func=nohandler, errorcode=1
  # invalid tss
handler num=11, func=nohandler, errorcode=1
  # segment not present
handler num=12, func=nohandler, errorcode=1
  # stack-segment fault
handler num=13, func=nohandler, errorcode=1
  # general protection
handler num=14, func=nohandler, errorcode=1
  # page fault
handler num=16, func=nohandler
  # math fault
handler num=17, func=nohandler, errorcode=1
  # alignment check
handler num=18, func=nohandler
  # machine check
handler num=19, func=nohandler
  # SIMD fp exception
```
Defining a family of (non) handlers

nohandler:    # dummy interrupt handler
    movl 4(%esp), %ebx  # get frame pointer
    pushl %ebx
    pushl %eax
    pushl $excepted
    call printf
    addl $12, %esp

1:             hlt
    jmp 1b

    ret

excepted:     .asciz "Exception 0x%x, frame=0x%x\n"

Initializing a context

struct Context user;

...  
initContext(&user, userEntry, 0);

...

void initContext(struct Context* ctxt, unsigned eip, unsigned esp) {
    extern char USER_DS[];
    extern char USER_CS[];
    printf("user data segment is 0x%x\n", (unsigned)USER_DS);
    printf("user code segment is 0x%x\n", (unsigned)USER_CS);
    ctxt->segs.ds     = (unsigned)USER_DS;
    ctxt->segs.es     = (unsigned)USER_DS;
    ctxt->segs.fs     = (unsigned)USER_DS;
    ctxt->segs.gs     = (unsigned)USER_DS;
    ctxt->iret.ss     = (unsigned)USER_DS;
    ctxt->iret.esp    = esp;
    ctxt->iret.cs     = (unsigned)USER_CS;
    ctxt->iret.eip    = eip;
    ctxt->iret.eflags = INIT_USER_FLAGS;
}
Initializing the flags

```c
#define INIT_USER_FLAGS (3<<12 | 1<<9 | 1<<1)
```

![Figure 2-5. System Flags in the EFLAGS Register](image)

Switching to a user program

From C:
```
extern int switchToUser(struct Context* ctx);
```

To Assembly:
```
.set CONTEXT_SIZE, 72
.globl switchToUser
switchToUser:
    movl 4(%esp), %eax      # Load address of the user context
    movl %eax, %esp         # Reset stack to base of user context
    addl $CONTEXT_SIZE, %eax
    movl %eax, esp0         # Set stack address for kernel reentry
    popa
    pop %ds                 # Restore registers
    pop %es
    pop %fs
    pop %gs
    addl $4, %esp           # Skip error code
    iret                     # Return from interrupt
```
Entering a system call (kernel view)

Initialize IDT entry:

\[ \text{idtcalc handler=kputc, slot=0x80, dpl=3} \]

Define a stub to handle the interrupt:

```asm
.text

kputc: subl $4, %esp       # Fake an error code
       push %gs            # Save segments
       push %fs
       push %es
       push %ds
       pusha               # Save registers
       leal stack, %esp    # Switch to kernel stack
       jmp kputc_imp
```

Provide a handler implementation:

```c
void kputc_imp() {
    /* A trivial system call */
    putchar(user.regs.eax);
    switchToUser(&user);
}
```

Entering a system call (user view)

From C:

```c
extern void kputc(unsigned);
```

To Assembly:

```asm
.globl kputc

kputc: pushl %eax
       mov 8(%esp), %eax
       int $128
       popl %eax
       ret
```
A recipe for adding a new system call

• Pick an unused interrupt number.

• Add code to initialize the corresponding IDT entry.

• Write and assembly code stub that saves the user program context and jumps to the handler code.

• Write the implementation of the handler. Be sure to use switchToUser (or equivalent) when the handler is done.

• Add user-level code to access the new system call. This often requires an assembly code fragment using the int instruction, and a declaration/prototype in the C code

• Color key for example-idt:
  kernel/init.s  kernel/kernel.c  user/userlib.s  user/user.c

Reflections

• Bare Metal
  • Segmentation, protection, exceptions and interrupts

• Programming/Languages
  • Representation transparency, facilitates interlanguage interoperability
  • Memory areas
    • Vendor-defined layout: GDT, GDTTR, TSS, IDT, IDTR, IRet, Registers, …
    • Self-defined: Context, …
  • “Bitdata”
    • Segment and interrupt descriptors, eflags, cr0, …
  • Does the need for a “recipe” suggest a language weakness?
Let’s see how all the pieces fit together …