x86 and xv6

CS 450: Operating Systems
Michael Saelee <saelee@iit.edu>
To work on an OS kernel, we must be intimately familiar with the underlying ISA, hardware, and system conventions

- x86 ISA
- PC architecture
- Unix, GCC, ELF, etc.
§ x86 ISA
- Intel IA-32 Software Developer’s Manuals (linked on course website) are comprehensive references
- Volume 1: Architectural Overview (e.g., regs, addressing)
- Volume 2: Instruction Set Reference
- Volume 3: Systems Programming Guide (e.g., mechanisms that let operating system control/configure hardware)
- ( Majority of diagrams in slides are from these manuals)
x86 Family of ISAs

- Started with Intel’s 8086 16-bit CPU in 1978
- Followed by 80186, 80286, 80386, 80486 (then Pentium …)
- 80386 introduced 32-bit addressing ("IA-32" architecture)
- CISC-style ISA
  - Large instruction set, complex addressing modes
Backwards Compatibility

- “x86” implies backwards compatibility all the way to 8086
- All x86 CPUs boot into 16-bit “real address mode” (aka “real mode”)
- Supported CPUs can switch into 32-bit “Protected Mode”
- i.e., we need to understand real mode to write an OS!
Instruction Set Overview

- Arithmetic: add, sub, and, etc.
- Moving data: mov, push, pop, etc.
- Control flow: jmp, call, ret, etc.
- I/O: in, out
- Privileged: int, iret, hlt, etc.
Instruction formats:

- 0 operands, e.g., `ret`
- 1 operand, e.g., `pushl %ebp`
- 2 operands — OP SRC, DST — e.g., `movl $0xa, %eax`
- Operands may be immediate values (constants), registers, memory addresses
NB: we’ll be using AT&T syntax for x86 assembly

- output by GCC/GAS

  - Constants are prefixed with $, Register names with %

  - Instruction suffixes (b=8-bit, w=16-bit, l=32-bit, etc.) used to indicate operand sizes

- not the same as official Intel syntax! (output by NASM)
### General-Purpose Registers

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EAX</td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td>BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESP</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 3-5. Alternate General-Purpose Register Names

- **EAX** — Accumulator for operands and results data
- **EBX** — Pointer to data in the DS segment
- **ECX** — Counter for string and loop operations
- **EDX** — I/O pointer
- **ESI** — Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- **EDI** — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
- **ESP** — Stack pointer (in the SS segment)
- **EBP** — Pointer to data on the stack (in the SS segment)
EFLAGS register used for conditional operations
E.g., if last operation resulted in zero/nonzero/neg/etc.

```assembly
movl $0, %eax
.L0:
addl $1, %eax
cmpl $10, %eax  # 10-eax
jne  .L0       # jump if ZF≠0
```

```c
for (i=0; i<10; i++);  
```
As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

### 3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

- **CF (bit 0)** Carry flag
  - Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

- **PF (bit 2)** Parity flag
  - Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

- **AF (bit 4)** Auxiliary Carry flag
  - Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

- **ZF (bit 6)** Zero flag
  - Set if the result is zero; cleared otherwise.

- **SF (bit 7)** Sign flag
  - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

- **OF (bit 11)** Overflow flag
  - Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

![Figure 3-8. EFLAGS Register](image.png)

(only bottom 16-bits in real mode)
IP (16-bit) / EIP (32-bit) is *instruction pointer* register (PC elsewhere)
- always points to next instruction; automatically incremented
- change explicitly with jump, call, ret, etc.
Addressing modes:

- **Direct**: `movl 0x401000, %eax`
  \[ \approx \text{eax} = *(\text{uint32}_t *) (0x401000) \]

- **Indirect**: `movl (%ebx), %eax`
  \[ \approx \text{eax} = *(\text{uint32}_t *) \text{ebx} \]
Addressing modes (continued):

- **Base-Displacement**: `movl 8(%ebx), %eax`
  \[ \approx \text{eax} = *(\text{uint32}_t \*) (\text{ebx} + 8) \]

- **Indexed & Scaled**: `movl (%ebx, %ecx, 4), %eax`
  \[ \approx \text{eax} = *(\text{uint32}_t \*) (\text{ebx} + \text{ecx} \times 4) \]
16-bit addressing modes:
\[
\begin{align*}
\{ &\text{CS : } \\
&\text{DS : } \} + \left[ \begin{array}{l}
&\{BX\} \\
&\{BP\} \\
&\{SI\} \\
&\{DI\} \\
\end{array} \right] + \text{[displacement]} \\
\end{align*}
\]

32-bit addressing modes:
\[
\begin{align*}
\{ &\text{CS : } \\
&\text{DS : } \} + \left[ \begin{array}{l}
&\{EAX\} \\
&\{EBX\} \\
&\{ECX\} \\
&\{EDX\} \\
&\{ESP\} \\
&\{EBP\} \\
&\{ESI\} \\
&\{EDI\} \\
\end{array} \right] * \left[ \begin{array}{l}
&\{EAX\} \\
&\{EBX\} \\
&\{ECX\} \\
&\{EDX\} \\
&\{EBP\} \\
&\{ESI\} \\
&\{EDI\} \\
\end{array} \right] + \text{[displacement]} \\
\end{align*}
\]

(Courtesy WikiMedia Commons)
Real mode addressing

- 8086 has 16-bit registers, but **20-bit physical addresses**
- Use one of four 16-bit segment registers: CS, DS, SS, ES
  - Shift left by 4 bits (i.e., $\times 16$) to obtain a segment *base address*
- Add to virtual address to obtain physical address
Real mode addressing

- Code and Stack accesses using IP, SP, and BP automatically use CS (code segment) and SS (stack segment) values

- e.g., if IP=0x4000 and CS=0x1100, CS:IP refers to absolute address 0x1100×16 + 0x4000 = 0x15000

- Can be confusing and unwieldy (especially if data straddles segments)
Protected mode addressing

- Full 32-bit addresses stored in registers
- Segment registers (expanded to CS, DS, SS, ES, FS, GS, and still all 16-bit) no longer hold base addresses, but *selectors*
- Selectors are used to load *segment descriptors* from a *descriptor table* which describe location/size/status/etc. of segments
If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor's address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address.

See also: Chapter 4, "Paging."

### 3.4.1 Logical Address Translation in IA-32e Mode

In IA-32e mode, an Intel 64 processor uses the steps described above to translate a logical address to a linear address. In 64-bit mode, the offset and base address of the segment are 64-bits instead of 32 bits. The linear address format is also 64 bits wide and is subject to the canonical form requirement.

Each code segment descriptor provides an L bit. This bit allows a code segment to execute 64-bit code or legacy 32-bit code by code segment.

### 3.4.2 Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

- **Index** (Bits 3 through 15) — Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

- **TI (table indicator) flag** (Bit 2) — Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

![Figure 3-5. Logical Address to Linear Address Translation](image)

**Figure 3-5. Logical Address to Linear Address Translation**
Segmentation

- Recall: segmentation allows virtual addresses to be translated using segment base addresses
- Segment descriptors also allow for access control (e.g., restricted access to certain segments)
- Mapping from segmented to linear address can be simple/flat or arbitrarily complex!
programs to multi-segmented models that employ segmentation to create a robust operating environment in which multiple programs and tasks can be executed reliably.

The following sections give several examples of how segmentation can be employed in a system to improve memory management performance and reliability.

3.2.1 Basic Flat Model

The simplest memory model for a system is the basic "flat model," in which the operating system and application programs have access to a continuous, unsegmented address space. To the greatest extent possible, this basic flat model hides the segmentation mechanism of the architecture from both the system designer and the application programmer.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created, one for referencing a code segment and one for referencing a data segment (see Figure 3-2). Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes. By setting the segment limit to 4 GBytes, the segmentation mechanism is kept from generating exceptions for out of limit memory references, even if no physical memory resides at a particular address. ROM (EPROM) is generally located at the top of the physical address space, because the processor begins execution at FFFFF_FFF0H. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.
More complexity can be added to this protected flat mode to provide more protection. For example, for the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined: code and data segments at privilege level 3 for the user, and code and data segments at privilege level 0 for the supervisor. Usually these segments all overlay each other and start at address 0 in the linear address space. This flat segmentation model along with a simple paging structure can protect the operating system from applications, and by adding a separate paging structure for each task or process, it can also protect applications from each other. Similar designs are used by several popular multitasking operating systems.

3.2.3 Multi-Segment Model

A multi-segment model (such as the one shown in Figure 3-4) uses the full capabilities of the segmentation mechanism to provide hardware enforced protection of code, data structures, and programs and tasks. Here, each program (or task) is given its own table of segment descriptors and its own segments. The segments can be completely private to their assigned programs or shared among programs. Access to all segments and to the execution environments of individual programs running on the system is controlled by hardware.

Figure 3-3. Protected Flat Model
Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.

Note that the processor does not perform segment limit checks at runtime in 64-bit mode.

3.2.5 Paging and Segmentation

Paging can be used with any of the segmentation models described in Figures 3-2, 3-3, and 3-4. The processor’s paging mechanism divides the linear address space (into which segments are mapped) into pages (as shown in Figure 3-1). These linear-address-space pages are then mapped to pages in the physical address space. The paging mechanism offers several page-level protection facilities that can be used with or instead of the segment-
Segment Descriptor Tables

- Kernel is responsible for maintaining descriptor tables on a system wide (via Global Descriptor Table) or task-specific (via Local Descriptor Table) basis

- Part of growing list of kernel data structures!
A segment selector contains the following:

- A 16-bit identifier for a segment (see Section 3.4.2 Segment Selectors).

In 64-bit mode, the offset and base address of a code segment are 64 bits wide and subject to the canonical form requirement.

In IA-32e mode, an Intel 64 processor uses the steps described above to translate a logical address to a linear address.

Address translation is used to translate the linear address into a physical address. If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor's address bus).

If paging is used, a second level of address translation is required. The first descriptor in the GDT is not used by the processor. Each system must have one GDT defined, which may be used for all programs and tasks in the system. Optionally, one or more LDTs can be defined. For example, an LDT can be defined for each separate task being run, or some or all tasks can share the same LDT.

The GDT is not a segment itself; instead, it is a data structure in linear address space. The base linear address and limit values of the GDT must be loaded into the GDTR register (see Section 2.4, "Memory-Management Registers"). The limit value for the GDT is expressed in bytes. As with segments, the limit value for the GDT is the address of the last valid byte. A limit value of 0 results in exactly one valid byte. Because segment descriptors are always 8 bytes long, the GDT limit should always be one less than an integral multiple of eight (that is, 8N – 1).

The GDT limit must be loaded into the GDTR register (using the SGDT instruction) so that the processor can access memory using the descriptor. By initializing the Table Indicator (TI) flag to 0, the processor selects the GDT; setting the TI flag to 1 selects the LDTR register (see Section 2.4, "Memory-Management Registers").

A segment selector to this "null descriptor" does not generate an exception when loaded into a data-segment register, but it always generates a general-protection exception (#GP) when an attempt is made to access memory using the descriptor. By initializing the TI flag to 0, the processor selects the GDT; setting the TI flag to 1 selects the LDTR register (see Section 2.4, "Memory-Management Registers").

The first descriptor in the GDT is not used by the processor. The limit of the GDT must be loaded into the GDTR register (using the SGDT instruction), a 48-bit "pseudo-descriptor" is stored in memory that is added to the base address to get the address of the segment descriptor (see top diagram in Figure 3-11). To avoid alignment check faults in user mode (privilege level 3), the pseudo-descriptor should be located at an odd word address (that is, address MOD 4 is equal to 2). This causes the processor to fault when attempting to access an aligned segment descriptor.

The GDTR register contains information about the GDT, such as its size, limit, and base address. The GDTR register is a 48-bit structure that includes the following fields:

- Base Address
- Limit
- Seg. Sel.

The LDT is located in a system segment of the LDT type. The segment descriptor for an LDT is in the GDT. The segment descriptor for an LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the rights of the LDT are stored in the LDTR register (see Section 2.4, "Memory-Management Registers").

Figure 3-6. Segment Selector

Figure 3-10. Global and Local Descriptor Tables
The flags and fields in a segment descriptor are as follows:

**Segment limit field**
Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
- If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

The processor uses the segment limit in two different ways, depending on whether the segment is an expand-up or an expand-down segment. See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for more information about segment types. For expand-up segments, the offset in a logical address can range from 0 to the segment limit. Offsets greater than the segment limit generate general-protection exceptions (#GP, for all segments other than SS) or stack-fault exceptions (#SS for the SS segment). For expand-down segments, the segment limit has the reverse function; the offset can range from the segment limit plus 1 to FFFFFFFFH or FFFFH, depending on the setting of the B flag. Offsets less than or equal to the segment limit generate general-protection exceptions or stack-fault exceptions. Decreasing the value in the segment limit field for an expand-down segment allocates new memory at the bottom of the segment's address space, rather than at the top. IA-32 architecture stacks always grow downwards, making this mechanism convenient for expandable stacks.

**Base address fields**
Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

**Type field**
Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors (see Figure 5-1). See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for a description of how this field is used to specify code and data-segment types.

**Figure 3-8. Segment Descriptor**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Base 31:24</td>
<td>Base address</td>
</tr>
<tr>
<td>24</td>
<td>G</td>
<td>Granularity flag</td>
</tr>
<tr>
<td>23</td>
<td>D/L</td>
<td>Default operation size (0 = 16-bit segment; 1 = 32-bit segment)</td>
</tr>
<tr>
<td>22</td>
<td>A/L</td>
<td>Available for use by system software</td>
</tr>
<tr>
<td>21</td>
<td>Type</td>
<td>Segment type</td>
</tr>
<tr>
<td>20</td>
<td>DPL</td>
<td>Descriptor privilege level</td>
</tr>
<tr>
<td>19</td>
<td>S</td>
<td>Descriptor type (0 = system; 1 = code or data)</td>
</tr>
<tr>
<td>15</td>
<td>L</td>
<td>64-bit code segment (IA-32e mode only)</td>
</tr>
<tr>
<td>14</td>
<td>BASE</td>
<td>Segment base address</td>
</tr>
<tr>
<td>13</td>
<td>D/B</td>
<td>Default operation size</td>
</tr>
<tr>
<td>12</td>
<td>AVL</td>
<td>Available for use by system software</td>
</tr>
<tr>
<td>11</td>
<td>TYPE</td>
<td>Segment type</td>
</tr>
<tr>
<td>8</td>
<td>Type</td>
<td>Segment type</td>
</tr>
<tr>
<td>7</td>
<td>P</td>
<td>Segment present</td>
</tr>
<tr>
<td>4</td>
<td>LIMIT</td>
<td>Segment limit</td>
</tr>
<tr>
<td>3</td>
<td>S</td>
<td>Descriptor type (0 = system; 1 = code or data)</td>
</tr>
<tr>
<td>2</td>
<td>BASE</td>
<td>Segment base address</td>
</tr>
<tr>
<td>1</td>
<td>DESCRIPTOR</td>
<td>Descriptor privilege level</td>
</tr>
<tr>
<td>0</td>
<td>TYPE</td>
<td>Segment type</td>
</tr>
</tbody>
</table>
CS and CPL

- Bottom 2 bits of CS indicate the CPL (current privilege level)
  - Recall: 0 = highest, 3 = lowest — used to guard access to privileged/restricted instructions and memory
- CS segment selector cannot be manipulated directly (why?)
  - Loaded from descriptor DPL (when switching segments)
xv6 and Segmentation

- xv6 mostly uses a flat model, so segmentation setup is simple

- But segmentation descriptors are very similar to those used in interrupt descriptor tables (IDTs)

- Used for carrying out interrupts and enforcing privilege level (CPL) policies — coming later
Paging

- Recall: paging allows for more granular mapping of linear address spaces onto physical memory
  - *Fixed sized pages* mapped from linear to physical address space
- We will use support for 2-level paging for 32-bit addresses:
  - 1K page directory entries $\rightarrow$ 1K page tables (each)
  - 4KB pages
Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the form of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.
Segmentation & Paging

- Segmentation is set up to provide a (mostly) flat model in xv6
- Paging is used to do heavy lifting of virtual → physical memory mapping
- xv6 sets up paging structures on a per-process basis
If paging is not used, the linear address space of the processor is mapped directly into the physical address space of the processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk storage. When using paging, each segment is divided into pages (typically 4 KBytes each in size), which are stored either in physical memory or on the disk. The operating system or executive maintains a page directory and a set of page tables to keep track of the pages. When a program (or task) attempts to access an address location in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location.

If the page being accessed is not currently in physical memory, the processor interrupts execution of the program (by generating a page-fault exception). The operating system or executive then reads the page into physical memory from the disk and continues executing the program. When paging is implemented properly in the operating system or executive, the swapping of pages between physical memory and the disk is transparent to the correct execution of a program. Even programs written for 16-bit IA-32 processors can be paged (transparently) when they are run in virtual-8086 mode.

3.2 USING SEGMENTS
The segmentation mechanism supported by the IA-32 architecture can be used to implement a wide variety of system designs. These designs range from flat models that make only minimal use of segmentation to protect...
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Address of page directory\(^1\) | Ignored | P | C | D | Ignored | CR3 |
| Address of page table | Ignored | 0 | P | I | g | n | A | P | C | D | P | W | T | U / S | R / W | 1 | PDE: page table |
| Ignored | 0 | PDE: not present |
| Address of 4KB page frame | Ignored | G | P | A | T | D | A | P | C | D | P | W | T | U / S | R / W | 1 | PTE: 4KB page |
| Ignored | 0 | PTE: not present |

**Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging**

---

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.

2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change. Table 4-3. Use of CR3 with 32-Bit Paging

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>3</td>
<td>PWT: Page-level write-through; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)</td>
</tr>
<tr>
<td>4</td>
<td>PCD: Page-level cache disable; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)</td>
</tr>
<tr>
<td>11:5</td>
<td>Ignored</td>
</tr>
<tr>
<td>31:12</td>
<td>Physical address of the 4-KByte aligned page directory used for linear-address translation</td>
</tr>
<tr>
<td>63:32</td>
<td>Ignored (these bits exist only on processors supporting the Intel-64 architecture)</td>
</tr>
</tbody>
</table>
Control & System Registers

- Transitioning between real & protected mode, activating paging, switching descriptor tables, etc., are all governed by control & system register settings

- Modifying these settings is restricted by CPL
CR0 - Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.

When loading a control register, reserved bits should always be set to the values previously read. The flags in control registers are:

- **CR0.PG** (Paging, bit 31 of CR0) - Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses. The PG flag has no effect if the PE flag (bit 0 of CR0) is not also set; setting the PG flag when the PE flag is clear causes a general-protection exception (#GP). See also: Chapter 4, "Paging."

- **CR0.CD** (Cache Disable, bit 30 of CR0) - When the CD and NW flags are clear, caching of memory locations for the whole of physical memory in the processor's internal (and external) caches is enabled. When the CD flag is set, caching is restricted as described in Table 11-5. To prevent the processor from accessing and updating its caches, the CD flag must be set and the caches must be invalidated so that no cache hits can occur. See also: Section 11.5.3, "Preventing Caching," and Section 11.5, "Cache Control."

- **CR0.NW** (Not Write-through, bit 29 of CR0) - When the NW and CD flags are clear, write-back (for Pentium 4, Intel Xeon, P6 family, and Pentium processors) or write-through (for Intel486 processors) is enabled for writes that hit the cache and invalidation cycles are enabled. See Table 11-5 for detailed information about the effect of the NW flag on caching for other settings of the CD and NW flags.

**Figure 2-7. Control Registers**
When loading a control register, reserved bits should always be set to the values previously read. The flags in control registers are:

- **CD** flag: Enables write-through (for Intel 486 processors) or write-back (for Pentium 4, Intel Xeon, P6 family, and Pentium processors) or write-combine (bit 29 of CR0)
- **NW** flag: Enables/disable write-through (bit 16 of CR0); when clear, write-back is enabled
- **PE** flag: Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses
- **P** flag: Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses
- **PG** flag: Enables or disables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses
- **M** flag: Enables or disables writing to the page directory and page tables when set; disables writing when clear
- **E** flag: Enables or disables execution of code when set; disables execution when clear
- **V** flag: Enables or disables execution of code when set; disables execution when clear
- **D** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **F** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **A** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **P** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **K** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **S** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **X** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **I** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **U** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **M** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **C** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **N** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses
- **R** flag: Enables or disable caching of memory locations for writes that hit the cache and invalidation cycles are disabled, all linear addresses are treated as physical addresses

The processor switches to SMM whenever it receives an SMI. The processor enters into IA-32e mode from protected mode if IA32_EFER.LME (bit 8) is set. See also: Chapter 9, "Processor Management and Initialization."
.globl start
start:
  cli               # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw %ax,%ax      # Set %ax to zero
  movw %ax,%ds      # -> Data Segment

  lgdt gdtdesc
  movl %cr0, %eax
  orl $CR0_PE, %eax
  movl %eax, %cr0

  # Complete transition to 32-bit protected mode by using long jmp
  # to reload %cs and %eip. The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp $(SEG_KCODE<<3), $start32

.code32            # Tell assembler to generate 32-bit code now.
.start32:
  # Set up the protected-mode data segment registers
  movw $(SEG_KDATA<<3), %ax      # Our data segment selector
  movw %ax, %ds                   # -> DS: Data Segment

  # Set up the stack pointer and call into C.
  movl $start, %esp
  call bootmain
I/O

- Two basic models for accessing I/O devices:
  - Separate *I/O port* address space from data address space
  - Memory-mapped I/O
I/O Ports

- Separated, limited address space

- Special, privileged instructions (\texttt{in/out}) to explicitly read from / write to I/O ports

  - e.g., \texttt{in }$0\times60$, \%al \# get last typed character (PS/2 kbd)
Memory-mapped I/O

- I/O controllers responsible for mapping ranges of physical addresses to I/O devices
- Look like regular memory accesses
- No special instructions!
- Accessing these special addresses read/write I/O devices
I/O Protocols

- Two basic protocols for interacting with I/O devices:
  - Polling, aka Programmed I/O (PIO)
  - Interrupt-driven I/O
Polling / Programmed I/O

- Software-driven, e.g.,

```
L0:
  inb STATUS_PORT, %al
  andb %al, BUSY_FLAG  # check if busy (e.g., still writing)
  jne L0

movb (%esi,%ebx), %al  # load data byte
incb %ebx              # increment index

outb DATA_PORT, %al   # write data byte to data port
outb CONTROL_PORT, $0x01  # let device know to inspect data
outb CONTROL_PORT, 0

jmp L0
```
Polling / Programmed I/O

- CPU must periodically probe hardware for status
- to check if device is ready to be written to
- to check if data is available to be read
- to check for exceptional conditions
Interrupt-driven I/O

- Device is responsible for *notifying* CPU of event
- CPU might still initial request, but doesn’t need to poll
- Notification happens via hardware *interrupt* mechanism
- Invokes interrupt handler (kernel routine)
Pros/Cons of Polled vs. Interrupt-driven I/O?
Pros/Cons of Polling

- no additional hardware requirements
- “easy” to program
- if CPU is idle, increase utilization

- places burden on CPU — problematic if heavy load
- possibly unresponsive
- even if no I/O is needed!
Pros/Cons of Interrupt-driven

- good for CPU utilization
- outsource responsibility
- if no I/O needed, no CPU activity at all!

- requires hardware support (limited interrupts)
- if frequent I/O, requires lots of context switches!
- concurrency issues?
Interrupts & Exceptions

- NB: terminology differs a bit from what we used in CS 351!
- Encompass all events that request special CPU attention, typically by transferring control from the active task to a special interrupt/exception handler
Interrupts

- Hardware-sourced events requesting CPU attention
- Typically unrelated to executing instruction
- Can also be generated by software with `int N` instruction
Exceptions

- Errors/Events arising due to the *currently executing instruction*

- Classes:
  - Faults
  - Traps
  - Aborts
Exception classes (review!)

- Faults: can be corrected — after handler, return to state prior to faulting instruction (e.g., page fault)

- Traps: reported immediately after execution of instruction (e.g., debugging breakpoint, system call), regular return

- Abort: severe errors; cannot return to task
“Tasks”

- Must be some sort of context to be “interrupted”
- TSS segment is used to define the currently executing task
  - General purpose registers
  - Control registers (including EFLAGS, EIP, LDTR, etc.)
  - Stack pointers for different privilege levels
7.1.2 Task State

The following items define the state of the currently executing task:

- The task's current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS).
- The state of the general-purpose registers.
- The state of the EFLAGS register.
- The state of the EIP register.
- The state of control register CR3.
- The state of the task register.
- The state of the LDTR register.
- The I/O map base address and I/O map (contained in the TSS).
- Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS).
- Stack pointer to the previously executed task (contained in the TSS).

Prior to dispatching a task, all of these items are contained in the task’s TSS, except the state of the task register. Also, the complete contents of the LDTR register are not contained in the TSS, only the segment selector for the LDT.

7.1.3 Executing a Task

Software or the processor can dispatch a task for execution in one of the following ways:

- An explicit call to a task with the CALL instruction.
- An explicit jump to a task with the JMP instruction.
- An implicit call (by the processor) to an interrupt-handler task.
- An implicit call to an exception-handler task.
- A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

All of these methods for dispatching a task identify the task to be dispatched with a segment selector that points to a task gate or the TSS for the task. When dispatching a task with a CALL or JMP instruction, the selector in the instruction may select the TSS directly or a task gate that holds the selector for the TSS. When dispatching a task, the processor fetches the instruction from the segment selector and stores the instruction in the instruction register, initializes the general-purpose registers, and jumps to the instruction.
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.

**Figure 7-2. 32-Bit Task-State Segment (TSS)**

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

Note: Reserved bits. Set to 0.
Handling Interrupts/Exceptions

- Interrupt Descriptor Table (IDT) contains “gate” descriptors associating service routines with interrupt/exception numbers
- 255 total indices (aka vector numbers):
  - 0-31: architecture-defined
  - 32-255: user-defined; can be assigned to I/O devices
NOTE
Because interrupts are delivered to the processor core only once, an incorrectly configured IDT could result in incomplete interrupt handling and/or the blocking of interrupt delivery.

IA-32 architecture rules need to be followed for setting up IDTR base/limit/access fields and each field in the gate descriptors. The same applies for the Intel 64 architecture. This includes implicit referencing of the destination code segment through the GDT or LDT and accessing the stack.

6.11 IDT DESCRIPTORS
The IDT may contain any of three kinds of gate descriptors:

- Task-gate descriptor
- Interrupt-gate descriptor
- Trap-gate descriptor

Figure 6-2 shows the formats for the task-gate, interrupt-gate, and trap-gate descriptors. The format of a task gate used in an IDT is the same as that of a task gate used in the GDT or an LDT (see Section 7.2.5, "Task-Gate Descriptor"). The task gate contains the segment selector for a TSS for an exception and/or interrupt handler task.

Interrupt and trap gates are very similar to call gates (see Section 5.8.3, "Call Gates"). They contain a far pointer (segment selector and offset) that the processor uses to transfer program execution to a handler procedure in an exception- or interrupt-handler code segment. These gates differ in the way the processor handles the IF flag in the EFLAGS register (see Section 6.12.1.2, "Flag Usage By Exception- or Interrupt-Handler Procedure").

Figure 6-1. Relationship of the IDTR and IDT

[Diagram showing the relationship between the IDTR register and the IDT, including how gate entries are indexed.

ILLOIS INSTITUTE OF TECHNOLOGY
College of Science]
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.

**Figure 6-3. Interrupt Procedure Call**
Interrupt/Exception Vectors

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

<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>INTO 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>UD 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 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.¹</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>
<tr>
<td>15</td>
<td>—</td>
<td>(Intel reserved. Do not use.)</td>
<td>Fault</td>
<td>No</td>
<td>x87 FPU floating-point or WAIT/FwAIT instruction.</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>Any data reference in memory.²</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Fault</td>
<td>Yes</td>
<td>Error codes (if any) and source are model dependent.³</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Abort</td>
<td>No</td>
<td>SSE/SSE2/SSE3 floating-point instructions⁴</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>Fault</td>
<td>No</td>
<td>EPT violations⁵</td>
</tr>
<tr>
<td>20</td>
<td>#VE</td>
<td>Virtualization Exception</td>
<td>Fault</td>
<td>No</td>
<td>External interrupt or INT n instruction.</td>
</tr>
</tbody>
</table>

¹ This exception can occur only on processors that support floating-point emulation. The #DF exception is generated only on processors that do not support floating-point emulation.² This exception can occur only on processors that support floating-point emulation.³ This exception can occur only on processors that support floating-point emulation.⁴ This exception can occur only on processors that support floating-point emulation.⁵ This exception can occur only on processors that support floating-point emulation.
Gate Descriptors

- Interrupts are associated with *Interrupt Gates*
- Exceptions are associated with *Trap Gates*
### 6.12 Exception and Interrupt Handling

The processor handles calls to exception- and interrupt-handling procedures in a manner 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 the 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").

#### 6.12.1 Exception- or Interrupt-Handler Procedures

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](image-url)

**Task Gate**

- **Offset 31..16**
- **DPL**
- **Offset 15..0**
- **Segment Selector**
- **Offset 15..0**

**Interrupt Gate**

- **Offset 31..16**
- **DPL**
- **Offset 15..0**

**Trap Gate**

- **Offset 31..16**
- **DPL**
- **Offset 15..0**

- **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**
Privilege level checks

- Hardware ensures that interrupts cannot transfer control from more-privileged to less-privileged code
  - i.e., enforce CPL $\geq$ destination segment DPL
- For interrupts generated with `int` instruction, gate DPL is checked to restrict interrupts
  - i.e., enforce CPL $\leq$ gate DPL
Masking Interrupts

- Most external interrupts can be *masked* (i.e., ignored), by setting the IF (interrupt flag) in EFLAGS register

- `cli/sti` instructions: clear/set interrupt flag

- IF is automatically cleared when an interrupt (but not a trap) gate is taken

- How is this useful?
Interrupt Procedure

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.
To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The IF flag is changed only if the CPL is less than or equal to the IOPL. See Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for a description of the complete operation performed by the IRET instruction.

If a stack switch occurred when calling the handler procedure, the IRET instruction switches back to the interrupted procedure's stack on the return.

### 6.12.1.1 Protection of Exception- and Interrupt-Handler Procedures

The privilege-level protection for exception- and interrupt-handler procedures is similar to that used for ordinary procedure calls when called through a call gate (see Section 5.8.4, "Accessing a Code Segment Through a Call Gate"). The processor does not permit transfer of execution to an exception- or interrupt-handler procedure in a less privileged code segment (numerically greater privilege level) than the CPL.

An attempt to violate this rule results in a general-protection exception (#GP). The protection mechanism for exception- and interrupt-handler procedures is different in the following ways:

- Because interrupt and exception vectors have no RPL, the RPL is not checked on implicit calls to exception and interrupt handlers.
- The processor checks the DPL of the interrupt or trap gate only if an exception or interrupt is generated with an INTn, INT3, or INTO instruction. Here, the CPL must be less than or equal to the DPL of the gate. This restriction prevents application programs or procedures running at privilege level 3 from using a software interrupt to access critical exception handlers, such as the page-fault handler, providing that those handlers are

---

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

```plaintext
Stack Usage with No Privilege-Level Change

Interrupted Procedure's and Handler's Stack

<table>
<thead>
<tr>
<th>ESP Before Transfer to Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS</td>
</tr>
<tr>
<td>CS</td>
</tr>
<tr>
<td>EIP</td>
</tr>
<tr>
<td>Error Code</td>
</tr>
</tbody>
</table>

ESP After Transfer to Handler

Stack Usage with Privilege-Level Change

Interrupted Procedure's Stack

<table>
<thead>
<tr>
<th>ESP Before Transfer to Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS</td>
</tr>
<tr>
<td>ESP</td>
</tr>
<tr>
<td>EFLAGS</td>
</tr>
<tr>
<td>CS</td>
</tr>
<tr>
<td>EIP</td>
</tr>
</tbody>
</table>

ESP After Transfer to Handler

Handler's Stack

<table>
<thead>
<tr>
<th>Error Code</th>
</tr>
</thead>
</table>
```
Returning from interrupt

- Need to restore previous stack/register states, and potentially return to previous privileged level (switch stacks)

- `iret` instruction automatically restores state using values saved on stack (including EIP, CS, EFLAGS) by interrupt procedure
Figure 2-1. IA-32 System-Level Registers and Data Structures
§ PC Architecture
What else?

- Memory + memory layout
- Persistent store (disk)
- Text/graphics display
- Keyboard/Mouse + other I/O devices and controllers
- BIOS, Clock
<table>
<thead>
<tr>
<th>Physical memory map</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Protected mode memory-mapped devices</td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td>0xFFFFFFFF (4GB)</td>
</tr>
<tr>
<td>Extended memory</td>
<td></td>
</tr>
<tr>
<td>BIOS ROM</td>
<td>0x00100000 (1MB)</td>
</tr>
<tr>
<td>Real-mode devices</td>
<td>0x000F0000 (960KB)</td>
</tr>
<tr>
<td>VGA display</td>
<td>0x000C0000 (768KB)</td>
</tr>
<tr>
<td>Low memory</td>
<td>0x000A0000 (640KB)</td>
</tr>
<tr>
<td></td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
Startup & BIOS

- On startup, transfer control to address FFFF:0000 (real mode)
- BIOS executes power on self test, initializes video card, disk controller, and sets up basic interrupt routines for simple I/O
- If boot drive is found, load boot sector (512 bytes, tagged with ending 0x55AA marker) from drive at address 0000:7C00
Bootloader Responsibilities

- Set up minimal execution environment (stack, protected mode)
- Scans disk for kernel image (may load second-stage bootloader to navigate partitions, file system, executable formats, etc.)
- Load kernel image at predetermined location in memory
- Transfer control to kernel
On Bootloaders

- Bootloaders can get very complicated!
- E.g., multistage boot loaders like Linux Loader (LILO) and Grand Unified Bootloader (GRUB) understand file systems and executable file formats
- Also have scripting support and built-in shells
§ QEMU
Full System Emulator

- Emulates the behavior of a real x86 PC in software
- Simulates physical memory map and I/O devices
- Supports up to 255 CPUs (speed dependent on host machine)
- Simple to debug, and won’t break your actual OS!
- Can connect to GDB to “step” through instructions
The QEMU PC System emulator simulates the following peripherals:
- i440FX host PCI bridge and PIIX3 PCI to ISA bridge
- Cirrus CLGD 5446 PCI VGA card or dummy VGA card with Bochs VESA extensions (hardware level, including all non standard modes).
- PS/2 mouse and keyboard
- 2 PCI IDE interfaces with hard disk and CD-ROM support
- Floppy disk
- PCI and ISA network adapters
- Serial ports
- IPMI BMC, either and internal or external one
- Creative SoundBlaster 16 sound card
- ENSONIQ AudioPCI ES1370 sound card
- Intel 82801AA AC97 Audio compatible sound card
- Intel HD Audio Controller and HDA codec
- Adlib (OPL2) - Yamaha YM3812 compatible chip
- Gravis Ultrasound GF1 sound card
- CS4231A compatible sound card
- PCI UHCI, OHCI, EHCI or XHCI USB controller and a virtual USB-1.1 hub.
SMP is supported with up to 255 CPUs.
QEMU uses the PC BIOS from the Seabios project and the Plex86/Bochs LGPL VGA BIOS.
§ Demo