x86 & xv6 overview

CS 450: Operating Systems
Michael Lee <lee@iit.edu>
Agenda

- Motivation
- x86 ISA
- PC architecture
- UNIX
- xv6
Motivation

- OS relies on many low-level hardware mechanisms to do its job
- To work on an OS kernel, we must be intimately familiar with the underlying ISA and PC hardware
  - Hardware may dictate what is or isn’t possible, and influence how we represent and manage system-level structures
- We focus on x86, but all modern ISAs support the mechanisms we need
  - e.g., xv6 has been ported to ARM already
§ x86
Documentation

- Intel IA-32 Software Developer’s Manuals are complete references
  - Volume 1: Architectural Overview
  - Volume 2: Instruction Set Reference
  - Volume 3: Systems Programming Guide
- Many diagrams in slides taken from them
x86 coverage

- Timeline
- Syntax
- Registers
- Instruction operands
- Instructions and sample usage
- Processor modes
- Interrupt & Exception handling
Timeline

- **1978**: Intel released 8086, a 16-bit CPU
- **1982**: 80186 and 80286 (still 16-bit)
- **1985**: 80386 was the first 32-bit x86 CPU (aka i386/IA-32)
- **2000**: AMD created x86-64: 64-bit ISA compatible with x86
- **2001**: Intel released IA-64 “Itanium” ISA, incompatible with x86
  - End-of-life announced in 2019 (i.e., official failure)
x86 ISA

- xv6 uses the IA-32 ISA
  - But we can still build/run it on x86-64!
- x86 is a CISC ISA, so we have:
  - Memory operands for non-load/store instructions
  - Complex addressing modes
  - Relatively large number of instructions
Syntax / Formatting

- Two common variants: Intel and AT&T syntax

- *Intel syntax* common in Windows world
  - e.g., `mov DWORD PTR [ebp-4], 10 ; format: OP DST, SRC`

- *AT&T syntax* common in UNIX world (default GCC output)
  - e.g., `movl $10, -4(%ebp) # format: OP SRC, DST`

- We will use this syntax
 Registers

- 8 general-purpose registers
- 6 segment registers for addressing
- Status & Control register
- Program counter / Instruction pointer
- (Many others — including control registers — coming up later)

![Diagram of Registers and Segments](image)
General purpose registers

- Can be directly manipulated, but some have special applications
- Most can be accessed as full 32-bit values, or as 16/8-bit subvalues
- Each register is, by convention, *volatile* or *non-volatile*
  - A *volatile* register may be clobbered by a function call; i.e., its value should be saved — maybe on the stack — if it must be preserved
  - A *non-volatile* register is preserved (by callees) across function calls
The special uses of general-purpose registers by instructions are described in Chapter 5, "Instruction Set Summary," in this volume. See also: Chapter 3, Chapter 4 and Chapter 5 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B & 2C.

The following is a summary of special uses:

- **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)

As shown in Figure 3-5, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SI, DI, and SP. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

### 3.4.1.1 General-Purpose Registers in 64-Bit Mode

In 64-bit mode, there are 16 general-purpose registers and the default operand size is 32 bits. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 are available. R8D-R15D/R8-R15 represent eight new general-purpose registers.

All of these registers can be accessed at the byte, word, dword, and qword level. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>Return value</td>
</tr>
<tr>
<td>%ebx</td>
<td>—</td>
</tr>
<tr>
<td>%ecx</td>
<td>Counter</td>
</tr>
<tr>
<td>%edx</td>
<td>—</td>
</tr>
<tr>
<td>%ebp</td>
<td>Frame/Base pointer</td>
</tr>
<tr>
<td>%esi</td>
<td>Source index (for arrays)</td>
</tr>
<tr>
<td>%edi</td>
<td>Destination index (for arrays)</td>
</tr>
<tr>
<td>%esp</td>
<td>Stack pointer</td>
</tr>
</tbody>
</table>

%eax, %ecx, %edx are volatile registers
# Instruction operands

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example(s)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$0x42, $0xd00d</td>
<td>Literal value</td>
</tr>
<tr>
<td>Register</td>
<td>%eax, %esp</td>
<td>Value found in register</td>
</tr>
<tr>
<td>Direct</td>
<td>0x4001000</td>
<td>Value found in address</td>
</tr>
<tr>
<td>Indirect</td>
<td>(%esp)</td>
<td>Value found at address in register</td>
</tr>
<tr>
<td>Base-Displacement</td>
<td>$8(%esp), $8(%esp), $8(%esp), $8(%esp)</td>
<td>Given $D(B)$, value found at address $D+B$ (i.e., address in base register $B + numeric offset D$)</td>
</tr>
<tr>
<td>Scaled Index</td>
<td>$8(%esp, %esi, 4)</td>
<td>Given $D(B, I, S)$, value found at address $D+B+I\times S$ $S \in {1, 2, 4, 8}$; $D$ and $I$ default to 0 if left out, $S$ defaults to 1</td>
</tr>
</tbody>
</table>

## Memory references
Instructions

- Instructions have 0-3 operands

- For many 2 operand instructions, one operand is both read and written
  
  - e.g., `addl $1, %eax`  
    # `%eax = %eax + 1`

- Instruction suffix indicates width of operands (1/w/b → 32/16/8 bits)

- Arithmetic operations populate EFLAGS register bits, including ZF (zero result), SF (signed/neg result), CF (carry-out of MSB occurred), OF (overflow occurred)

- Used by subsequent conditional instructions (e.g., jump if result = zero)
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**

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | D  | I  | D  | X  | X  | X  | X  | X  | X  | X  | X  | X  | S  | C  | C  | S  | S  | S  | S  |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | D  | I  | D  | X  | X  | X  | X  | X  | X  | X  | X  | X  | S  | C  | C  | S  | S  | S  | S  |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | D  | I  | D  | X  | X  | X  | X  | X  | X  | X  | X  | X  | S  | C  | C  | S  | S  | S  | S  |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | D  | I  | D  | X  | X  | X  | X  | X  | X  | X  | X  | X  | S  | C  | C  | S  | S  | S  | S  |
| X  | ID Flag (ID) | X  Virtual Interrupt Pending (VIP) | X Virtual Interrupt Flag (VIF) | X Alignment Check / Access Control (AC) | X Virtual-8086 Mode (VM) | X Resume Flag (RF) | X Nested Task (NT) | X I/O Privilege Level (IOPL) | S Overflow Flag (OF) | C Direction Flag (DF) | X Interrupt Enable Flag (IF) | X Trap Flag (TF) | S Sign Flag (SF) | S Zero Flag (ZF) | S Auxiliary Carry Flag (AF) | S Parity Flag (PF) | S Carry Flag (CF) |

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE. Always set to values previously read.
## Arithmetic

<table>
<thead>
<tr>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{add,sub,imul} src, dst</td>
<td>dst = dst {+,−,×} src</td>
</tr>
<tr>
<td>neg dst</td>
<td>dst = −dst</td>
</tr>
<tr>
<td>{inc,dec} dst</td>
<td>dst = dst {+,−} 1</td>
</tr>
<tr>
<td>{sal,sar,shr} src, dst</td>
<td>dst = dst {&lt;&lt;,&gt;&gt;,&gt;&gt;&gt;} src (arithmetic &amp; logical shifts)</td>
</tr>
<tr>
<td>{and,or,xor} src, dst</td>
<td>dst = dst {&amp;,</td>
</tr>
<tr>
<td>not dst</td>
<td>dst = ~dst (bitwise)</td>
</tr>
</tbody>
</table>

src can be an immediate, register, or memory operand; dst can be a register or memory operand. But at most one memory operand!
## Conditions and Branches

<table>
<thead>
<tr>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmp src, dst</code></td>
<td>dst – src (discard result but set flags)</td>
</tr>
<tr>
<td><code>test src, dst</code></td>
<td>dst &amp; src (discard result but set flags)</td>
</tr>
<tr>
<td><code>jmp target</code></td>
<td>Unconditionally jump to target (change %eip)</td>
</tr>
<tr>
<td><code>{je, jne} target</code></td>
<td>Jump to target if dst equal/not equal src (ZF=1 / ZF=0)</td>
</tr>
<tr>
<td><code>{jl, jle} target</code></td>
<td>Jump to target if dst $\leq$ src (SF≠OF / ZF=1 or SF≠OF)</td>
</tr>
<tr>
<td><code>{jg, jge} target</code></td>
<td>Jump to target if dst $\geq$ src (ZF=0 and SF=OF / SF=OF)</td>
</tr>
<tr>
<td><code>{ja, jb} target</code></td>
<td>Jump to target if dst above/below src (CF=0 and ZF=0 / CF=1)</td>
</tr>
</tbody>
</table>

*target* is usually an address encoded as an immediate operand (e.g., `jmp $0x4001000`), but addresses may be stored in a register or memory, in which case *indirect addressing* is required, which uses the * symbol. E.g., `jmp *%eax` (jump to address in %eax), `jmp *0x4001000` (jump to address found at address 0x4001000)
E.g., basic control structures

```c
if (cond) {
    // if-clause
} else {
    // else-clause
}
...
```

```assembly
testl %eax, %eax  # %eax = cond
je ELSE
    # if-clause
jmp ENDIF
ELSE:
    # else-clause
ENDIF:
    # ...
```

```c
while (cond) {
    // loop-body
}
...
```

```assembly
testl %eax, %eax  # %eax = cond
je ENDLOOP
je ENDLOOP
    # loop-body
LOOP:
    # loop-body
testl %eax, %eax
jne LOOP
ENDLOOP:
    # ...
```
## Data movement

<table>
<thead>
<tr>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov src, dst</td>
<td>Copy data from src to dst (memory→memory moves not possible)</td>
</tr>
<tr>
<td>movzbl src, dst</td>
<td>Copy 8-bit value to 32-bit target (&amp; other variants), using zero-fill</td>
</tr>
<tr>
<td>movsbl src, dst</td>
<td>Copy 8-bit value to 32-bit target (&amp; other variants), using sign-extension</td>
</tr>
<tr>
<td>{cmove/ne} src, dst</td>
<td>Move data from src to dst if ZF=1 / ZF=0</td>
</tr>
<tr>
<td>{cmovg/ge/l/le/a/b/…}</td>
<td>Conditionally move data from src to dst (per jump naming conventions)</td>
</tr>
</tbody>
</table>

## Address computation

| lea address, dst | dst = address (no memory access! just computes value of address) |
## Functions and Call stack

<table>
<thead>
<tr>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push src</td>
<td>Push src onto stack</td>
</tr>
<tr>
<td>pop dst</td>
<td>Pop top of stack into dst</td>
</tr>
<tr>
<td>call target</td>
<td>Push current %eip (address of instruction after call) onto stack, jump to target</td>
</tr>
<tr>
<td>leave</td>
<td>Restore frame pointer (%ebp) and clears stack frame</td>
</tr>
<tr>
<td>ret</td>
<td>Pop top of stack into %eip</td>
</tr>
</tbody>
</table>

All instructions above implicitly adjust %esp and access the stack.

*target* may use *indirect addressing* as well, e.g., `call *%eax` (call function whose address is in %eax)
Function calls

- Functions make extensive use of the call stack — leads to convention-driven prologue and epilogue blocks in assembly code

- Typical function prologue:
  - Save old frame pointer and establish new frame pointer
  - Save non-volatile register values we might clobber (“callee-saved”)
  - Load needed parameters from prior stack frame
  - Allocate stack space for any local data
Function calls

- Typical function epilogue:
  - Place return value in %eax
  - Deallocate any space used for local data
  - Restore/Pop any clobbered non-volatile register values
  - Restore/Pop old frame pointer
  - Return
Function calls (Optimization)

- Many of these steps may be optimized (simplified or neglected altogether) by the compiler!

- Prefer registers to stack-based args or local vars (regs vs. memory)

- `%esp` doesn’t always reflect the top of the stack (only need to do this if calling another function)

- `lea` often used in surprising ways (addressing modes as arithmetic)
Call Stack

- Maintains dynamic state and context of executing program
- Saved frame pointers (previous values of %ebp) create a chain of stack frames
- Useful to navigate for debugging and tracing! (e.g., gdb “backtrace”)
E.g., function calls

```c
int main() {
    int x=10, y=20;
    sum(x, y);
    return 0;
}

int sum(int a, int b) {
    int ret = a + b;
    return ret;
}
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl $10, -4(%ebp)
    movl $20, -8(%ebp)
    movl -4(%ebp), %eax
    movl -8(%ebp), %edx
    call sum
    movl $0, %eax
    addl $16, %esp
    popl %ebp
    ret
```

```
sum: # unoptimized
pushl %ebp
movl %esp, %ebp
movl %edi, -4(%ebp)
movl %esi, -8(%ebp)
movl -4(%ebp), %eax
addl -8(%ebp), %eax
movl %eax, -12(%ebp)
movl -12(%ebp), %eax
popl %ebp
ret
```

```
sum: # optimized
leal (%edi,%esi), %eax
ret
```
Processor modes

- When an x86 system first boots up, it runs in 16-bit real mode (8086 compatible) — all addresses reference “real” memory locations.

- 16/32-bit protected modes add privilege levels, virtual memory, and other mechanisms useful to the OS (e.g., for multitasking).

- 64-bit long mode removes some instructions and adds 64-bit registers and addressing.
Real mode addressing

- Only 16-bit registers, but support for **20-bit** addresses (1MB address space) through the use of segment registers: CS, DS, ES, SS

- Left-shift segment number by 4 (i.e., $\times 16$) to obtain base address, and add to offset to compute 20-bit physical address

- Code (via IP) and Stack (via SP and BP) accesses automatically use CS (code segment) and SS (stack segment) to compute addresses

- e.g., if IP=0x4000 and CS=0x1100, CS:IP refers to physical address $0x1100\times 16 + 0x4000 = 0x15000$
Protected mode

- Segment registers (expanded to CS, DS, SS, ES, FS, GS) 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

- CS selector contains a 2-bit CPL in addition to selector value

- Recall: privileged instructions are only available when CPL=0
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.

Segments allow complex memory mapping and access control (e.g., restricted access), among other things.
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-
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.

In practice, a flat model is used by most OSes, and more granular memory mapping & protection is carried out via paging (coming up)

- But segment descriptors are still used for privilege level based restrictions
DPL is loaded as CPL (in CS register) if (successful) jump occurs to this segment

---

The flags and fields in a segment descriptor are as follows:

- **P** — Segment present
- **G** — Granularity
- **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
- **S** — Descriptor type (0 = system; 1 = code or data)
- **TYPE** — Segment type
- **LIMIT** — Segment limit
  - **BASE_31_24**
  - **G**
  - **D**
  - **L**
  - **AVL**
  - **Seg. Limit_19_16**
  - **P**
  - **DPL**
  - **S**
  - **Type**
  - **Base_23_16**

**Example segment descriptor**

```c
struct segdesc {
    uint lim_15_0 : 16; // Low bits of segment limit
    uint base_15_0 : 16; // Low bits of segment base address
    uint base_23_16 : 8; // Middle bits of segment base address
    uint type : 4; // Segment type (see STS_ constants)
    uint s : 4; // 0 = system, 1 = application
    uint dpl : 2; // Descriptor Privilege Level
    uint p : 1; // Present
    uint lim_19_16 : 4; // High bits of segment limit
    uint avl : 1; // Unused (available for software use)
    uint rsv1 : 1; // Reserved
    uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
    uint g : 1; // Granularity: limit scaled by 4K when set
    uint base_31_24 : 8; // High bits of segment base address
};
```

**Normal segment**

```c
#define SEG(type, base, lim, dpl) (struct segdesc)  
   { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,  
     (uint)(base) >> 16) & 0xff, type, 1, dpl, 1,  
     (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 } 
```

```c
#define SEG16(type, base, lim, dpl) (struct segdesc)  
   { (lim) & 0xffff, (uint)(base) & 0xffff,  
     (uint)(base) >> 16) & 0xff, type, 1, dpl, 1,  
     (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 } 
```

@endef
Privilege check

- When loading segments, hardware ensures that CPL ≤ DPL (actually a bit more complicated, but this is very close to the truth!)
  - I.e., privilege level can only stay the same or be lowered
- Prevents user code from transitioning directly to kernel code
  - To elevate privilege, must do so by way of interrupts/traps!
Segment descriptor tables

- Kernel is responsible for maintaining descriptor tables
  - System wide (Global)
  - Task-specific (Local)
- Must be set up before transitioning to protected mode

![Diagram of segment descriptor tables]

- **Global Descriptor Table (GDT)**
  - First Descriptor in GDT is Not Used
  - TI = 0
  - GDTR Register
    - Base Address
    - Limit

- **Local Descriptor Table (LDT)**
  - TI = 1
  - LDTR Register
    - Base Address
    - Seg. Sel.
    - Limit
Control & System registers

- Transitioning between real & protected mode, and activating/controlling other hardware features are governed by control & system register flags
The control registers are:

- Compatibility mode.
- Processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in value that operating systems use to control the priority class of external interrupts allowed to interrupt the

When the CD and NW flags are clear, caching of memory locations for

Not Write-through (bit 29 of CR0) is not also set; setting the PG flag

Figure 2-7. Control Registers

Figure 2-3. Transitions Among the Processor's Operating Modes
.code16  # Assemble for 16-bit mode
.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

    ...  # Switch from real to protected mode. Use a bootstrap GDT that makes
    # virtual addresses map directly to physical addresses so that the
    # effective memory map doesn't change during the transition.
    lgdt    gdtdesc
    movl    %cr0, %eax
    orl     $CR0_PE, %eax
    movl    %eax, %cr0
    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
    ...

    # Bootstrap GDT
    gdt:
        SEG_NULLASM   # null seg
        SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)  # code seg
        SEG_ASM(STA_W, 0x0, 0xffffffff)  # data seg
Paging

- Protected mode also enables virtual memory via *paging*
- A much more granular (but potentially expensive) form of virtual memory
  - Will discuss this in detail later!
- Kernel must set up and maintain per-process structures for paging, too
Interrupts & Exceptions

- Events that require special CPU attention, typically by transferring control from the active task (kernel/user) to a kernel handler

- **Interrupts** are 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*

- Subclasses:

  - **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
Handling Interrupts/Exceptions

- **Interrupt Descriptor Table** (IDT) contains descriptors (aka “gates”) 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 and IDT](image-url)
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>INT3 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>#TS</td>
<td>Coprocessor Segment Overrun (reserved)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point instruction.</td>
</tr>
<tr>
<td>10</td>
<td>#NP</td>
<td>Invalid TSS</td>
<td>Fault</td>
<td>Yes</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#SS</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>#GP</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>#PF</td>
<td>General Protection</td>
<td>Fault</td>
<td>Yes</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>15</td>
<td>—</td>
<td>(Intel reserved. Do not use.)</td>
<td>—</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 (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>#XM</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>User Defined (Non-reserved) Interrupts</td>
<td>Fault</td>
<td>No</td>
<td>External interrupt or INT n instruction.</td>
</tr>
<tr>
<td>32-255</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

### Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)

<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>15</td>
<td>—</td>
<td>(Intel reserved. Do not use.)</td>
<td>—</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 (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>#XM</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>User Defined (Non-reserved) Interrupts</td>
<td>Fault</td>
<td>No</td>
<td>External interrupt or INT n instruction.</td>
</tr>
<tr>
<td>32-255</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
6.12 EXCEPTION AND INTERRUPT 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").

6.12.1 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.

```
// Gate descriptors for interrupts and traps
struct gatedesc {
    uint off_15_0 : 16; // low 16 bits of offset in segment
    uint cs : 16; // code segment selector
    uint args : 5; // # args, 0 for interrupt/trap gates
    uint rsv1 : 3; // reserved (should be zero I guess)
    uint type : 4; // type(STS_{IG32,TG32})
    uint s : 1; // must be 0 (system)
    uint dpl : 2; // descriptor (meaning new) privilege level
    uint p : 1; // Present
    uint off_31_16 : 16; // high bits of offset in segment
};

#define SETGATE(gate, istrap, sel, off, d)                
    {                                                         
        (gate).off_15_0 = (uint)(off) & 0xffff;                  
        (gate).cs = (sel);                                      
        (gate).args = 0;                                        
        (gate).rsv1 = 0;                                        
        (gate).type = (istrap) ? STS_TG32 : STS_IG32;           
        (gate).s = 0;                                           
        (gate).dpl = (d);                                       
        (gate).p = 1;                                           
        (gate).off_31_16 = (uint)(off) >> 16;                   
    }

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

- Three variables: CPL, gate DPL, and destination segment DPL

- Destination segment DPL is always 0 (handler is in kernel)

- CPU guarantees that:
  - for hardware interrupts, CPL ≥ destination segment DPL
    - i.e., interrupt cannot lower privilege!
  - for software generated interrupts (via int), CPL ≤ gate DPL
    - i.e., can use this to allow user mode to invoke only certain interrupts
  - if assertions fail, general protection fault (#13)
Masking Interrupts

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

- If interrupts occur in user mode, running handler with current stack is unsafe (unpredictable state)
- TSS segment defines 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).
- Link to 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 sets the CR3 register to the selector of the TSS for the task.

Figure 7-1. Structure of a Task
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.

![32-Bit Task-State Segment (TSS)](image_url)

*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>0</td>
<td>I/O Map Base Address</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td>LDT Segment Selector</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
<td>GS</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
<td>FS</td>
</tr>
<tr>
<td>16</td>
<td>Reserved</td>
<td>DS</td>
</tr>
<tr>
<td>20</td>
<td>Reserved</td>
<td>SS</td>
</tr>
<tr>
<td>24</td>
<td>Reserved</td>
<td>CS</td>
</tr>
<tr>
<td>28</td>
<td>Reserved</td>
<td>ES</td>
</tr>
<tr>
<td>32</td>
<td>EDI</td>
<td>68</td>
</tr>
<tr>
<td>34</td>
<td>ESI</td>
<td>64</td>
</tr>
<tr>
<td>36</td>
<td>EBX</td>
<td>60</td>
</tr>
<tr>
<td>38</td>
<td>ESP</td>
<td>56</td>
</tr>
<tr>
<td>40</td>
<td>EBX</td>
<td>52</td>
</tr>
<tr>
<td>42</td>
<td>EDX</td>
<td>48</td>
</tr>
<tr>
<td>44</td>
<td>ECX</td>
<td>44</td>
</tr>
<tr>
<td>46</td>
<td>EAX</td>
<td>40</td>
</tr>
<tr>
<td>48</td>
<td>EFLAGS</td>
<td>36</td>
</tr>
<tr>
<td>50</td>
<td>EIP</td>
<td>32</td>
</tr>
<tr>
<td>52</td>
<td>CR3 (PDBR)</td>
<td>28</td>
</tr>
<tr>
<td>56</td>
<td>Reserved</td>
<td>SS2</td>
</tr>
<tr>
<td>60</td>
<td>Reserved</td>
<td>ESP2</td>
</tr>
<tr>
<td>64</td>
<td>Reserved</td>
<td>SS1</td>
</tr>
<tr>
<td>68</td>
<td>Reserved</td>
<td>ESP1</td>
</tr>
<tr>
<td>72</td>
<td>Reserved</td>
<td>SS0</td>
</tr>
<tr>
<td>76</td>
<td>Reserved</td>
<td>ESP0</td>
</tr>
<tr>
<td>80</td>
<td>Reserved</td>
<td>Previous Task Link</td>
</tr>
</tbody>
</table>

*Reserved bits. Set to 0.*
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
PC Architecture
What else?

- Memory + memory layout
- Persistent store (disk)
- Text/graphics display
- Keyboard/Mouse + other I/O devices and controllers
- BIOS, Clock
Protected mode memory-mapped devices

Unused

Extended memory

BIOS ROM

Real-mode devices

VGA display

Low memory

Physical RAM limit

0x00000000

0x00000000 (0KB)

0x000A0000 (640KB)

0x000C0000 (768KB)

0x000F0000 (960KB)

0x00100000 (1MB)

0xFFFF0000 (4GB)
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