IA32/Linux Virtual Memory Architecture
### Application Programming Registers

<table>
<thead>
<tr>
<th>General-purpose registers</th>
<th>Segment registers</th>
<th>Control registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX 31</td>
<td>CS 15</td>
<td>CR0 31</td>
</tr>
<tr>
<td>EBX 15</td>
<td>DS 14</td>
<td>CR1 31</td>
</tr>
<tr>
<td>ECX 13</td>
<td>SS 13</td>
<td>CR2 31</td>
</tr>
<tr>
<td>EDX 12</td>
<td>ES 12</td>
<td>CR3 31</td>
</tr>
<tr>
<td>EBP 11</td>
<td>FS 11</td>
<td>CR4 31</td>
</tr>
<tr>
<td>ESI 10</td>
<td>TR 10</td>
<td></td>
</tr>
<tr>
<td>EDI 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIP 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EFLAGS 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### General-purpose registers
- EAX: 31
- EBX: 15
- ECX: 13
- EDX: 12
- EBP: 11
- ESI: 10
- EDI: 9
- ESP: 8
- EIP: 7
- EFLAGS: 0

#### Segment registers
- CS: 15
- DS: 14
- SS: 13
- ES: 12
- FS: 11
- GS: 10

#### Control registers
- CR0: 31
- CR1: 31
- CR2: 31
- CR3: 31
- CR4: 31

### System Table Registers
- GDTR: linear base address, table limit
- IDTR: linear base address, table limit

### System Segment Registers
- TR: seg. selector
- LDTR: seg. selector
IA32 VM Architecture (1)

- **Segmented memory model**
  - Memory appears to a program as a group of independent address space called segments.
  - A program must issue a logical address, which consists of a segment selector and an offset.
  - Up to 16,383 segments of different sizes and types
    - Each segment can be as large as $2^{32}$ bytes.
  - No way to disable segmentation.
  - The use of paging is optional.
IA32 VM Architecture (2)

- Logical address (far pointer)
  - User’s view, segmented

- Linear address
  - 32-bit, flat

- Physical address
  - 32-bit, flat
  - Pentium Pro and later processors support an extension of the physical address space to $2^{36}$ bytes.
  - Invoked with the physical address extension (PAE) flag located in CR4 register.
IA32 VM Architecture (3)

Logical Address (or Far Pointer)

Segment Selector  Offset

Global Descriptor Table (GDT)

Segment Descriptor

Segment Base Address

Linear Address Space

Segment

Lin. Addr.

Page Directory

Entry

Page Table

Entry

Page

Physical Address Space

Dir  Table  Offset

Segmentation  Paging
Segmentation (1)

- **Basic flat model**
  - The OS and applications have access to a continuous, unsegmented address space.
  - All segment descriptors have the same base address value of 0 and the same segment limit of 4GB.
Segmentation (2)

- **Protected flat model**
  - Segment limits are set to include only the range of addresses for which physical memory actually exists.
  - May have multiple segments, but all overlay each other and start at address 0 in the linear address space.
Multisegment model

- 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.
Segmentation (4)

- **Segment registers**
  - Hold 16-bit segment selectors.
    - A segment selector is a special pointer that identifies a segment in memory
    - To access a particular segment, the segment selector for that segment must be present in the appropriate segment register.
  - **Use of segment registers**
    - CS: for code segment
    - DS, ES, FS, and GS: for data segments (up to 4 segments simultaneously)
    - SS: for stack segment
  - FS and GS registers were introduced with the 80386 family of processors.
Segmentation (5)

- **Logical to linear address**
  - Examine the segment descriptor in GDT or LDT to check the access rights and the offset is within the limits.
  - Adds the segment base address from the segment descriptor to the offset to form a linear address.
Segmentation (6)

- Segment selector

```
<table>
<thead>
<tr>
<th>15</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>I</td>
<td>RPL</td>
<td></td>
</tr>
</tbody>
</table>
```

Table Indicator
- 0 = GDT
- 1 = LDT

Requested Privilege Level (RPL)

- Segment registers

<table>
<thead>
<tr>
<th>Visible Part</th>
<th>Hidden Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Selector</td>
<td>Base Address, Limit, Access Information</td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
</tbody>
</table>
Segmentation (7)

- Segment descriptor tables
  - Each system must have one GDT (Global Descriptor Table), which may be used for all programs and tasks.
  - Optionally, one or more LDTs (Local Descriptor Tables) can be defined in a system segment.
  - GDT is not a segment, but a data structure in the linear address space pointed to by the GDTR register.
  - GDT must contain a segment descriptor for the LDT segment.
  - The first descriptor in GDT is not used.
  - The LDTR register caches the segment descriptor of the current LDT segment.
Segmentation (8)

- Global and local descriptor tables

![Diagram showing global and local descriptor tables with GDTR and LDTR registers.]
Segmentation (9)

- Segment descriptor

| 31  | 24 23 22 21 20 19 | 16 15 14 13 12 11 | 8 7 0 |
|-----|-------------------|-------------------|------|---|
| Base 31:24 | G | D / B | L | AVL | Seg. Limit 19:16 | P | DPL | S | Type | Base 23:16 |
|            | 4 |      |   |     |               | 0 |     |   |      |           |

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

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

- **Paging support in IA-32**
  - Optional: enabled by PG flag of CR0 register
  - Default page size: 4KB
    - PSE (page size extension) flag of CR4 enables 4MB page size (From Pentium)

- **36-bit physical addressing**
  - Pentium Pro and later processors support an extension of the physical address space to $2^{36}$ bytes.
    - Enabled by PAE (physical address extension) flag of CR4
    - With PAE enabled, 2MB page size is supported
  - Pentium III introduced PSE-36 mechanism
    - Available when PSE-36 CPUID feature flag is set
    - Map up to 1024 4MB pages into 64GB physical address space
Paging (2)

- **Linear to physical address (4KB)**
  - The *physical* address of the current page directory is stored in the CR3 register (a.k.a. page directory base register or PDBR).

![Diagram of paging structure](image-url)

*32 bits aligned onto a 4-KByte boundary.
Paging (3)

- Page tables and directories
  - Page directory
    - An array of 32-bit page-directory entries (PDEs) contained in a 4KB page (1024 PDEs/page).
  - Page table
    - An array of 32-bit page-table entries (PTEs) contained in a 4KB page (1024 PTEs/page).
    - Page tables are not used for 2MB or 4MB pages.
  - Page
    - Supports page sizes of 4KB, 2MB, and 4MB.
  - Page-directory-pointer table
    - An array of four 64-bit entries pointing to a page directory.
    - Only used when the physical address extension is enabled.
Paging (4)

- Linear to physical address (4MB, PSE enabled)
  - Both 4MB pages and page tables for 4KB pages can be accessed from the same page directory
  - Place OS kernel in 4MB pages to reduce TLB misses
### Paging (5)

- Linear to physical address (4KB, PAE enabled)

![Diagram of paging](image)

- **Linear Address**
  - 31 30 29 21 20 12 11 0
  - Directory Pointer
  - Directory
  - Table
  - Offset

- **Page Directory**
  - 9
  - Directory Entry

- **Page-Directory-Pointer Table**
  - 32*
  - CR3 (PDPTR)

- **Page Table**
  - 9
  - Page-Table Entry

- **Physical Address**
  - 4-KByte Page

- **4 PDPTE * 512 PDE * 512 PTE = 2^{20} Pages**

*32 bits aligned onto a 32-byte boundary*
## Paging (6)

### Page directory entry (PDE)

<table>
<thead>
<tr>
<th>Page-Table Base Address</th>
<th>Avail</th>
<th>P</th>
<th>S</th>
<th>A</th>
<th>V</th>
<th>L</th>
<th>A</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>W</th>
<th>T</th>
<th>U</th>
<th>R</th>
<th>S</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
</table>

Available for system programmer’s use

Global page (Ignored)

Page size (0 indicates 4 KBytes)

Available

Accessed

Cache disabled

Write-through

User/Supervisor

Read/Write

Present
### Paging (7)

#### Page table entry (PTE)

<table>
<thead>
<tr>
<th>PTE Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Base Address</td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td>Global Page</td>
</tr>
<tr>
<td>P</td>
<td>Access Control Bits</td>
</tr>
<tr>
<td>D</td>
<td>Present Bit</td>
</tr>
<tr>
<td>C</td>
<td>Accessible Bit</td>
</tr>
<tr>
<td>W</td>
<td>User/Supervisor Bit</td>
</tr>
<tr>
<td>S</td>
<td>Write-Through Bit</td>
</tr>
<tr>
<td>P</td>
<td>Read/Write Bit</td>
</tr>
</tbody>
</table>

**Page-Table Entry (4-KByte Page)**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Page Base Address</td>
</tr>
<tr>
<td>12 11</td>
<td>Available</td>
</tr>
<tr>
<td>9</td>
<td>Global Page</td>
</tr>
<tr>
<td>8</td>
<td>Access Control Bits</td>
</tr>
<tr>
<td>7</td>
<td>Present Bit</td>
</tr>
<tr>
<td>6 5 4 3 2 1 0</td>
<td>Accessible Bit, User/Supervisor Bit, Write-Through Bit, Read/Write Bit, Present Bit</td>
</tr>
</tbody>
</table>
Paging (8)

**TLBs**

- The P6 family and Pentium processors have separate TLBs for the data and instruction. (DTLB & ITLB)
- Separate TLBs for 4KB and 4MB page sizes
- All TLBs are automatically invalidated if the PDBR register is loaded.
  - by explicit MOV instruction
  - implicitly by executing a task switch
- A specific page-table entry in the TLB can be invalidated using INVLPG instruction.
- The page global enable (PGE) flag in CR4 and the global (G) flag of a PDE or PTE can be used to prevent frequently used pages from being automatically invalidated.
IA32 References

- For more information, see
    - Volume 1: Basic Architecture
    - Volume 2: Instruction Set Reference
    - Volume 3: System Programming Guide
  - Available at Intel’s web site:
Linux VM Architecture (1)

Virtual memory

PAGE_OFFSET = 0xC0000000

Physical memory

Available Page Frames

Kernel code
Kernel data
Page tables
Freelists, etc.

3GB
1GB
1GB
0x00000000
0xC0000000
0xFFFFFFFF

0x00000000
0x00000000
0x3FFFFFFF
0x3FFFFFFF
Linux VM Architecture (2)

- Segmentation: Minimal approach
  - For better portability across machines

<table>
<thead>
<tr>
<th>Address</th>
<th>Segment Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>NULL</td>
<td>(not used)</td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td>(not used)</td>
</tr>
<tr>
<td>0x10</td>
<td>Kernel text from 0 (4GB)</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>Kernel data from 0 (4GB)</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>User text from 0 (4GB)</td>
<td></td>
</tr>
<tr>
<td>0x28</td>
<td>User data from 0 (4GB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(not used)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(not used)</td>
</tr>
<tr>
<td></td>
<td>Used for APM</td>
<td>(4 entries)</td>
</tr>
<tr>
<td>0xa0</td>
<td></td>
<td>Used for PNPBIOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(8 entries)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 entries per CPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For TSS’s &amp; LDT’s</td>
</tr>
</tbody>
</table>

Segment selectors

- __KERNEL_CS: 0x10
- __KERNEL_DS: 0x18
- __USER_CS: 0x23
- __USER_DS: 0x2b
Paging: Three-level address translation

- In i386, the size of Page Middle Directory (PMD) is 1, if the physical address extension (PAE) flag is disabled.
Linux VM Architecture (4)

- **Virtual memory areas (VMA)**
  - Nonoverlapping regions, each region representing a continuous, page-aligned subset of the virtual address space.
  - Described by a single `vm_area_struct`
  - VMAs are linked into a balanced binary tree to allow fast lookup of the region corresponding to any virtual address.
    - VMAs form a red-black tree.
Linux VM Architecture (5)

- **task_struct**
  - mm
  - map_count
  - pgd
  - mmap
  - mm_rb

- **vm_area_struct**
  - vm_start
  - vm_end
  - vm_mm
  - vm_rb
  - vm_ops
  - vm_next

- **mm_struct**
  - mm
  - pgd
  - mmap
  - mm_rb

- **page directory**
  - PFN

- **Virtual address space**
  - VM Area 1
  - VM Area 2
Linux VM Architecture (6)

- VMA example

<table>
<thead>
<tr>
<th>VMA</th>
<th>permission</th>
<th>offset</th>
<th>device</th>
<th>i-node</th>
<th>mapped file</th>
</tr>
</thead>
<tbody>
<tr>
<td>08048000-0804e000</td>
<td>r-xp</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>0804f000-08050000</td>
<td>rw-p</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>40000000-40013000</td>
<td>r-xp</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>40013000-40014000</td>
<td>rw-p</td>
<td>00013000</td>
<td>00013000</td>
<td>00013000</td>
<td>00013000</td>
</tr>
<tr>
<td>40031000-40032000</td>
<td>rw-p</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>4212c000-4212d000</td>
<td>r-xp</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>4212d000-42131000</td>
<td>rw-p</td>
<td>0012c000</td>
<td>0012c000</td>
<td>0012c000</td>
<td>0012c000</td>
</tr>
<tr>
<td>42131000-42135000</td>
<td>rw-p</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>bffff000-c0000000</td>
<td>rwxp</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>