Address translation
Segmentation
Descriptor table

Logical Address

15 0
Seg. Selector

Descriptor Table

Segment Descriptor

31(63) 0
Offset (Effective Address)

Base Address

31(63) 0
Linear Address

+
Segment descriptors

- Base address
  - 0 – 4 GB
- Limit (size)
  - 0 – 4 GB
- Access rights
  - Executable, readable, writable
  - Privilege level (0 - 3)
Segment descriptors

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

- Hold 16 bit segment selectors
  - Pointers into a special table
  - Global or local descriptor table
- Segments are associated with one of three types of storage
  - Code
  - Data
  - Stack
Code segment

- Code
  - CS register
  - EIP is an offset inside the segment stored in CS
- Can only be changed with
  - procedure calls,
  - interrupt handling, or
  - task switching
Data segment

- Data
  - DS, ES, FS, GS
  - 4 possible data segments can be used at the same time
Stack segment

- Stack
  - SS
- Can be loaded explicitly
  - OS can set up multiple stacks
  - Of course, only one is accessible at a time
Programming model

- Segments for: code, data, stack, “extra”
  - A program can have up to 6 total segments
  - Segments identified by registers: cs, ds, ss, es, fs, gs

- Prefix all memory accesses with desired segment:
  - mov eax, ds:0x80  (load offset 0x80 from data into eax)
  - jmp cs:0xab8    (jump execution to code offset 0xab8)
  - mov ss:0x40, ecx (move ecx to stack offset 0x40)
Segmented programming (not real)

static int x = 1;
int y; // stack
if (x) {
    y = 1;
    printf ("Boo");
} else {
    y = 0;
}

ds:x = 1; // data
ss:y; // stack
if (ds:x) {
    ss:y = 1;
    cs:printf(ds:"Boo");
} else {
    ss:y = 0;
}
Programming model, cont.

• This is cumbersome, so infer code, data and stack segments by instruction type:
  • Control-flow instructions use code segment (jump, call)
  • Stack management (push/pop) uses stack
  • Most loads/stores use data segment
• Extra segments (es, fs, gs) must be used explicitly
Logical address

- Segment selector (16 bit) + offset (32 bit)
Paging
Paging idea

- Break up memory into 4096-byte chunks called pages
  - Modern hardware supports 2MB, 4MB, and 1GB pages
- Independently control mapping for each page of linear address space

- Compare with segmentation (single base + limit)
  - many more degrees of freedom
Why do we need paging?

- Illusion of a private address space
  - Identical copy of an address space in multiple programs
    - Remember `fork()`?
- Simplifies software architecture
  - One program is not restricted by the memory layout of the others
Why do we need paging?

- Illusion of a private address space
  - Identical copy of an address space in multiple programs
    - Remember `fork()`?
- Simplifies software architecture
  - One program is not restricted by the memory layout of the others
- Emulate large virtual address space on a smaller physical memory
  - Swap rarely accessed pages to disk
Why do we need paging?

- Share a region of memory across multiple programs
  - Communication (shared buffer of messages)
  - Shared libraries
- Isolate parts of the program
- Isolate programs from OS
Page directory entry (PDE)

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4MB region controlled by this entry
- U/S – user/supervisor
  - If 0 – user-mode access is not allowed
- A – accessed
Page table entry (PTE)

<table>
<thead>
<tr>
<th>Address of 4KB page frame</th>
<th>Ignored</th>
<th>G</th>
<th>P</th>
<th>A</th>
<th>T</th>
<th>D</th>
<th>A</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>W</th>
<th>U</th>
<th>S</th>
<th>R</th>
<th>W</th>
<th>1</th>
<th>PTE: 4KB page</th>
</tr>
</thead>
</table>

- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4KB page
- U/S – user/supervisor
  - If 0 user-mode access is not allowed
- A – accessed
- D – dirty – software has written to this page
Page translation

Linear Address
31  22  21  12  11  0
Directory  Table  Offset

10
Page Directory

10
Page Table

12
4-KByte Page

12
Physical Address

32
CR3

PDE with PS=0

PTE
Back of the envelope

- If a page is 4K and an entry is 4 bytes, how many entries per page?
  - 1k
- How large of an address space can 1 page represent?
  - 1k entries * 1 page/entry * 4K/page = 4MB
- How large can we get with a second level of translation?
  - 1k tables/dir * 1k entries/table * 4k/page = 4 GB
  - Nice that it works out that way!
Questions?
TLB

- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)
- Walking page table is slow
  - Each memory access is 200-300 cycles on modern hardware
  - L3 cache access is 70 cycles

```
<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf0231000</td>
<td>0x1000</td>
</tr>
<tr>
<td>0x00b31000</td>
<td>0x1f000</td>
</tr>
<tr>
<td>0xb0002000</td>
<td>0xc1000</td>
</tr>
</tbody>
</table>
```
TLB

- TLB is a cache (in CPU)
  - It is not coherent with memory
  - If page table entry is changes, TLB remains the same and is out of sync

<table>
<thead>
<tr>
<th>Virt Addr.</th>
<th>Phys Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf0231000</td>
<td>0x1000</td>
</tr>
<tr>
<td>0x00b31000</td>
<td>0x1f000</td>
</tr>
<tr>
<td>0xb0002000</td>
<td>0xc1000</td>
</tr>
</tbody>
</table>

Same Virt Addr. No Change!!!
Invalidating TLB

• After every page table update, OS needs to manually invalidate cached values
• Modern CPUs have “tagged TLBs”,
  • Each TLB entry has a “tag” – identifier of a process
  • No need to flush TLBs on context switch
• On Intel this mechanism is called
  • Process-Context Identifiers (PCIDs)
More paging tricks

- Determine a working set of a program?
More paging tricks

• Determine a working set of a program?
  • Use “accessed” bit
More paging tricks

- Determine a working set of a program?
  - Use “accessed” bit
- Iterative copy of a working set?
  - Used for virtual machine migration
More paging tricks

- Determine a working set of a program?
  - Use “accessed” bit
- Iterative copy of a working set?
  - Used for virtual machine migration
  - Use “dirty” bit
More paging tricks

• Determine a working set of a program?
  • Use “accessed” bit

• Iterative copy of a working set?
  • Used for virtual machine migration
  • Use “dirty” bit

• Copy-on-write memory, e.g. lightweight fork()?
More paging tricks

- Determine a working set of a program?
  - Use “accessed” bit
- Iterative copy of a working set?
  - Used for virtual machine migration
  - Use “dirty” bit
- Copy-on-write memory, e.g. lightweight `fork()`?
  - Map page as read/only
When would you disable paging?
When would you disable paging?

- Imagine you're running a memcached
  - Key/value cache
- You serve 1024 byte values (typical) on 10Gbps connection
  - 1024 byte packets can leave every 835ns, or 1670 cycles (2GHz machine)
  - This is your target budget per packet
When would you disable paging?

• Now, to cover 32GB RAM with 4K pages
  • You need 64MB space
  • 64bit architecture, 3-level page tables
• Page tables do not fit in L3 cache
  • Modern servers come with 32MB cache
• Every cache miss results in up to 3 cache misses due to page walk (remember 3-level page tables)
  • Each cache miss is 200 cycles

• Solution: 1GB pages
Page translation for 4MB pages

Linear Address

31  22  21  0
Directory Offset

Page Directory

10
PDE with PS=1

CR3

18

4-MByte Page

Physical Address