CS5460/6460: Operating Systems

Lecture 5: Paging

Several slides in this lecture use slides developed by Don Porter

Anton Burtsev
January, 2014
Address translation
(recap)
Descriptor table

15  0
Seg. Selector

31(63)  0
Offset (Effective Address)

Logical Address

Descriptor Table

Segment Descriptor

Base Address

31(63)  0
Linear Address
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)

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

```c
ds:x = 1; // data
ds: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
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 translation

Linear Address

31  22  21  12  11  0
Directory  Table  Offset

Page Directory

PDE with PS=0

CR3

Page Table

PTE

Physical Address

4-KByte Page

10

10

12

20
Page directory entry (PDE)

| 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 table | Ignored | 0 | Ign | A | P | C | D | P | W | U | S | R | W | 1 | PDE: page table |

- 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 translation

Linear Address

31 22 21 12 11 0

Directory Table Offset

Page Directory

10

PDE with PS=0

32

CR3

Page Table

10

PTE

12

4-KByte Page

Physical Address
Page table entry (PTE)

- 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

Page Directory

PDE with PS=0

Page Table

PTE

CR3

4-KByte Page

Physical Address
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!
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
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</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>

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

PDE with PS=1

CR3

4-MByte Page

Physical Address