Operating Systems

Address translation
(Segmentation and Paging)

Anton Burtsev
October, 2021
Two programs one memory

main() {
    ...
    yield()
}
maint() {
    ...
    yield()
}

Save/restore
Two programs one memory

- How can we do this?

main() {
  ...
  yield()
}

main() {
  ...
  yield()
}

Save/restore
Relocation

• One way to achieve this is to relocate program at different addresses
  • Remember relocation (from linking and loading)
Relocate binaries to work at different addresses
• One way to achieve this is to relocate program at different addresses
  • Remember relocation (from linking and loading)
  • This works! But not ideal.
• What is the problem?
• One way to achieve this is to relocate program at different addresses
  • Remember relocation (from linking and loading)
  • This works! But not ideal.
• What is the problem?
  • Isolation can be enforced in software
  • Software Fault Isolation (SFI)
    – Google NaCl (Chrome Sandbox)
    – WASM (Web Assembly, another sandbox standard)
• Another way is to ask for hardware support
Segmentation
What are we aiming for?

- Illusion of a private address space
  - Identical copy of an address space in multiple programs
- Simplifies software architecture
  - One program is not restricted by the memory layout of the others
Two processes, one memory?

Process 1 (ls)

Process 2 (ls)

Memory
Two processes, one memory?

- We want hardware to add base value to every address used in the program
Seems easy

- One problem
  - Where does this base address come from?
Seems easy

- One problem
  - Where does this base address come from?
  - Hardware can maintain a table of base addresses
    - One base for each process
  - Dedicate a special register to keep an index into that table
• One problem
  • Where does this base address come from?
  • Hardware can maintain a table of base addresses
    – One base for each process
  • Dedicate a special register to keep an index into that table
Segmentation: example

```
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0x0
EBX = 0x300010
```

Process 1 (ls)

Physical Memory

Process 2 (ls)
Segmentation: address consists of two parts

- Segment register contains segment selector
- General registers contain offsets
- Intel calls this address: “logical address”
• GDT is an array of segment descriptors
  • Each descriptor contains base and limit for the segment
  • Plus access control flags
Segmentation: Global Descriptor Table

- Location of GDT in physical memory is pointed by the GDT register

Segment register (CS, SS, DS, ES, FS, GS)

Segment register (CS, SS, DS, ES, FS, GS)

- Location of GDT in physical memory is pointed by the GDT register

Global Descriptor Table (table of segment sizes and bases)

Global Descriptor Table (table of segment sizes and bases)

- Location of GDT in physical memory is pointed by the GDT register

Physical Memory

Physical Memory

- Location of GDT in physical memory is pointed by the GDT register

mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0x0
EBX = 0x300010, DS = 0x1

GDT register (pointer to the GDT, physical address)

GDT register (pointer to the GDT, physical address)
Segmentation: base + offset

Segment register (0x1) chooses an entry in GDT

- This entry contains base of the segment (0x110000) and limit (size) of the segment (not shown).

GDT register (pointer to the GDT, physical address)

- Segment register (0x1) chooses an entry in GDT
Segmentation: base + offset

**Physical address:**
- $0x410010 = 0x300010$ (offset) + $0x110000$ (base)
- Intel calls this address “linear”
Each process has a private GDT

- OS will switch between GDTs
New addressing mode: “logical addresses”
All addresses are logical address

• They consist of two parts
  • Segment selector (16 bit) + offset (32 bit)
- Segment selector (16 bit)
  - Is simply an index into an array (Descriptor Table)
  - That holds segment descriptors
    - Base and limit (size) for each segment
Elements of the descriptor table are segment descriptors

- Base address
  - 0 – 4 GB
- Limit (size)
  - 0 – 4 GB
- Access rights
  - Executable, readable, writable
  - Privilege level (0 - 3)
- Offsets into segments (x in our example) or "Effective addresses" are in registers.
• Logical addresses are translated into physical

• *Effective address + DescriptorTable[selector].Base*
• Logical addresses are translated into physical
  • Effective address + DescriptorTable[selector].Base
Logical addresses are translated into physical addresses by adding the effective address and the base address from the descriptor table. This process involves using the selector to index into the descriptor table, which contains segment descriptors, and then adding the base address to the offset to obtain the linear address.
- Logical addresses are translated into physical
- Effective address + DescriptorTable[selector].Base
• **Physical address** =
  
  \[ \text{Effective address} + \text{DescriptorTable}[\text{selector}].\text{Base} \]

• Effective addresses (or offsets) are in registers

• **Selector is in a special register**
Segment registers

- Hold 16 bit segment selectors
  - Indexes into GDT

- Segments are associated with one of three types of storage
  - Code
  - Data
  - Stack
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

- 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)
Programming model, cont.

- This is cumbersome,
- Instead the idea is: infer code, data and stack segments from the 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
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
Segmentation: what did we achieve

- Illusion of a private address space
  - Identical copy of an address space in multiple programs
    - We can implement `fork()`

- Isolation
  - Processes cannot access memory outside of their segments
Segmentation works for isolation, i.e., it does provide programs with illusion of private memory.
Segmentation is ok... but
What if process needs more memory?
What if process needs more memory?
You can move P2 in memory
Or even swap it out to disk
Problems with segments

• Segments are somewhat inconvenient
  • Relocating or swapping the entire process takes time

• Memory gets fragmented
  • There might be no space (gap) for the swapped out process to come in
  • Will have to swap out other processes
Paging
Pages

![Diagram showing memory allocation for two processes](image-url)
Pages

Process 1 (ls)

Process 2 (ls)

Memory

Page table

Level 1

Level 2
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
How can we build this translation mechanism?
Paging: naive approach: translation array

- Linear address 0x410010
  - Remember it’s result of logical to linear translation (aka segmentation)
    - 0x410010 = 0x300010 (offset) + 0x110000 (base)
Paging: naive approach: translation array

- Linear address 0x410010
  - Remember it’s result of logical to linear translation (aka segmentation)
    - $0x410010 = 0x300010$ (offset) + $0x110000$ (base)
What is wrong?
What is wrong?

• We need 4 bytes to relocate each page
  • 20 bits for physical page number
  • 12 bits of access flags

• Therefore, we need array of 4 bytes x 1M entries
  - 4MBs
Paging: naive approach: translation array

- Each entry is 4 bytes
  -- 20 bits to represent page number + access control bits
- 1 page can contain 1024 entries
- We need 1024 pages to represent all possible 1M translations
Paging: array with size

- The size controls how many entries are required
But still what may go wrong?
Paging: array with size
Paging: array with size

V2P translation array

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

500,000
Paging: array with chunks

0x410010 = 00 0000 0001|00 0001 0000|0000 0001 0000

Table of array regions

V2P translation array

500,000
Paging: array with chunks

$0x410010 = \begin{array}{c}
000000001 | 00001 0000 | 0000 0000 0001 0000
\end{array}$

Table of array regions

- $0x8$
- $0x9$
- ...
- $0xF$

V2P translation array

Physical memory
Paging: page table

$0x410010 = \begin{array}{c}
00000000100000000000000000010000
\end{array}$

- **Page Table Directory** (aka Level 1)
  - 0x8
  - 0x9
  - ...
  - 0xF

- **V2P translation array**
  - Page Table Entry (aka Level 2)
    - Page Table Entry (aka Level 2)
    - Page Table Entry (aka Level 2)

- **Physical memory**

- **CR3 = 0x2**
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 0101|00 0000 0011|0000 0000 0001

page number

Virtual Address Space (or Memory) of the Process

0 1 2

... page number = 5123 or (0b1 0100 0000 0011)

Physical Memory

0 1 2 3 4 5 6 7 8 9 10 11 12
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 0101|00 0000 0011|0000 0000 0001

Virtual Address Space (or Memory) of the Process

CR3 = 0

page number = 5123
or (0b1 0100 0000 0011)
`mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809`
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 010 00 0000 0011 0000 0000 0001

Virtual Address Space (or Memory) of the Process

CR3 = 0

Physical Memory

32 bits (4 bytes)

Page number = 5123 or (0b1 0100 0000 0011)
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 0101 | 00 0000 0011 | 0000 0000 0001

Virtual Address Space (or Memory) of the Process

CR3 = 0

page number = 5123 or (0b1 0100 0000 0011)

1M (1,048,575)

Physical Memory

Level 1 (Page Table Directory)

Level 2 (Page Table)

Page
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 0101 00 0000 0011 0000 0000 0001

Virtual Address Space (or Memory) of the Process

CR3 = 0

page number = 5123 or (0b1 0100 0000 0011)

Result:
- EAX = 55

1M (1,048,575)

Level 1 (Page Table Directory)

Level 2 (Page Table)

Page
Page translation

- Linear Address
  - 31 22 21 12 11 0
    - Directory | Table | Offset

- Page Directory
- PDE with PS=0
- CR3

- Page Table
- PTE

- 4-KByte Page
- Physical Address
Page directory entry (PDE)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>Address of page table</td>
</tr>
<tr>
<td>0</td>
<td>Ignored</td>
</tr>
<tr>
<td>1</td>
<td>PDE: page table</td>
</tr>
</tbody>
</table>

- 20 bit address of the page table
Page directory entry (PDE)

- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits
Page directory entry (PDE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of page table</td>
<td>Ignored</td>
<td>0</td>
<td>I g n</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>P W</td>
<td>U</td>
<td>S</td>
<td>R</td>
<td>W</td>
<td>P D E: page table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits

- Pages 4KB each, we need 1M to cover 4GB
- Pages start at 4KB (page aligned boundary)
Page translation
Page table entry (PTE)

- 20 bit address of the 4KB page
- Pages 4KB each, we need 1M to cover 4GB
Page translation

Linear Address

31 22 21 12 11 0
Directory Table Offset

Page Directory

Page Table

PTE

PDE with PS=0

CR3

4-KByte Page

Physical Address
Benefit of page tables

... Compared to arrays?

- Page tables represent sparse address space more efficiently
  - An entire array has to be allocated upfront
  - But if the address space uses a handful of pages
  - Only page tables (Level 1 and 2 need to be allocated to describe translation)

- On a dense address space this benefit goes away
  - I'll assign a homework!
What about isolation?

- Two programs, one memory?
- Each process has its own page table
  - OS switches between them

```cpp
main() {
  ...
  yield()
}
```

```cpp
main() {
  ...
  yield()
}
```
P1 and P2 can't access each other memory
Compared to segments pages allow ...

- Emulate large virtual address space on a smaller physical memory
  - In our example we had only 12 physical pages
  - But the program can access all 1M pages in its 4GB address space
  - The OS will move other pages to disk
Compared to segments pages allow ...

- Share a region of memory across multiple programs
- Well… segments allow this too
- Communication (shared buffer of messages)
- Shared libraries
Recap: complete address translation
32bit x86 supports two page sizes

- 4KB pages
- 4MB pages
Page translation for 4MB pages

0x410010 = 00 0000 0001 00 0001 0000 0000 0001 0000

Page Table Directory (aka Level 1)

Physical memory

CR3 = 0x8192
Page translation for 4MB pages

Linear Address

31 22 21 0

Directory Offset

Page Directory

PDE with PS=1

4-MByte Page

Physical Address

CR3
Virtual addresses are 48 bits

Physical addresses are 52 bits
Questions?
What pages are used for

- Protect parts of the program
  - E.g., map code as read-only
    - Disable code modification attacks
    - Remember R/W bit in PTD/PTE entries!
  - E.g., map stack as non-executable
    - Protects from stack smashing attacks
    - Non-executable bit
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
TLB

- Walking page table is slow
  - Each memory access is 240 (local) - 360 (one QPI hop away) cycles on modern hardware
  - L3 cache access is 50 cycles
TLB

- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)

<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 invalidation

- 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!!!
TLB invalidation

- After every page table update, OS needs to manually invalidate cached values
  - Flush TLB
    - Either one specific entry
    - Or entire TLB, e.g., when CR3 register is loaded
    - This happens when OS switches from one process to another
  - This is expensive
    - Refilling the TLB with new values takes time
Tagged TLBs

- 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)

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf0231000</td>
<td>0x1000</td>
<td>P1</td>
</tr>
<tr>
<td>0x00b31000</td>
<td>0x1f000</td>
<td>P2</td>
</tr>
<tr>
<td>0xb0002000</td>
<td>0xc1000</td>
<td>P1</td>
</tr>
</tbody>
</table>
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, 4-level page tables (or 5-levels now)
- Page tables do not fit in L3 cache
  - Modern servers come with 32MB cache
- Every cache miss results in up to 4 cache misses due to page walk (remember 4-level page tables)
  - Each cache miss is 250 cycles

- Solution: 1GB pages
Back of the envelope

• If a page is 4K and an entry is 4 bytes, how many entries per page?
Back of the envelope

- If a page is 4K and an entry is 4 bytes, how many entries per page?
  - 1k
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?
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 * 1page/entry * 4K/page = 4MB
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 * 1page/entry * 4K/page = 4MB
• How large can we get with a second level of translation?
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 * 1page/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!
### Segment descriptors

<table>
<thead>
<tr>
<th>Base 31:24</th>
<th></th>
<th>D</th>
<th>L</th>
<th>AVL</th>
<th>Seg. Limit 19:16</th>
<th>P</th>
<th>DPL</th>
<th>S</th>
<th>Type</th>
<th>Base 23:16</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>

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

Linear Address:
- 31: Directory
- 22-21: Table
- 12-11: Offset

4-KByte Page

Physical Address

Page Table

PTE

Page Directory

PDE with PS=0

CR3

12

10

10

20

32
## Page directory entry (PDE)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>Address of page table</td>
</tr>
<tr>
<td>27</td>
<td>Ignored</td>
</tr>
<tr>
<td>26-19</td>
<td>Ignored</td>
</tr>
<tr>
<td>18-12</td>
<td>Ignored</td>
</tr>
<tr>
<td>11</td>
<td>Ignored</td>
</tr>
<tr>
<td>10</td>
<td>Ignored</td>
</tr>
<tr>
<td>9-8</td>
<td>Ignored</td>
</tr>
<tr>
<td>7-6</td>
<td>Ignored</td>
</tr>
<tr>
<td>5-4</td>
<td>Ignored</td>
</tr>
<tr>
<td>3-2</td>
<td>Ignored</td>
</tr>
<tr>
<td>1</td>
<td>PDE: page table</td>
</tr>
</tbody>
</table>

- 20 bit address of the page table
Page directory entry (PDE)

- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits
Page directory entry (PDE)

- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits
- Pages 4KB each, we need 1M to cover 4GB
- Pages start at 4KB (page aligned boundary)
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 | PCD | PW | UT | RS | RW |

- Bit #1: R/W – writes allowed?
- But allowed where?
Page directory entry (PDE)

- Bit #1: R/W – writes allowed?
  - But allowed where?
  - One page directory entry controls 1024 Level 2 page tables
    - Each Level 2 maps 4KB page
  - So it's a region of 4KB x 1024 = 4MB
**Page directory entry (PDE)**

<table>
<thead>
<tr>
<th>Bit #2: U/S – user/supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>If 0 – user-mode access is not allowed</td>
</tr>
<tr>
<td>Allows protecting kernel memory from user-level applications</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address of page table</th>
<th>Ignored</th>
<th>0</th>
<th>Ign</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>
</tr>
</thead>
<tbody>
<tr>
<td>PDE: page table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Bit #2: U/S – user/supervisor
  - If 0 – user-mode access is not allowed
  - Allows protecting kernel memory from user-level applications
Page translation

Linear Address

Directory | Table | Offset

31 22 21 12 11 0

4-KByte Page

12

Page Directory

10

PDE with PS=0

10

Page Table

20

PTE

12

Physical Address

20

CR3

32
**Page table entry (PTE)**

| 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 4KB page frame | Ignored | G | P | A | T | D | A | P | C | D | P | W | U | S | R | U | S | R | W | 1 | PTE: 4KB page |

- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - To a 4KB page
- Bit #2: U/S – user/supervisor
  - If 0 user-mode access is not allowed
- Bit #5: A – accessed
- Bit #6: D – dirty – software has written to this page
Questions?