Virtual Memory II
CSE 351 Summer 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

https://xkcd.com/1495/
Administrivia

❖ Questions doc: https://tinyurl.com/CSE351-8-10

❖ hw19 is optional
  ▪ Can complete it at any point before the quarter ends
  ▪ Practice with virtual memory concepts

❖ hw20 due Friday (8/14) – 10:30am

❖ Lab 4 due Wednesday (8/12) – 11:59pm
  ▪ All about caches!
Virtual Memory (VM)

- Overview and motivation
- VM as a tool for caching
- **Address translation**
- VM as a tool for memory management
- VM as a tool for memory protection
Address Translation

How do we perform the virtual → physical address translation?

CPU Chip

Virtual address (VA) 0x4100

MMU

Physical address (PA) 0x4

Main memory

0: 
1: 
2: 
3: 
4: 
5: 
6: 
7: 
8: 

Memory Management Unit

Data (int/float)
Address Translation: Page Tables

❖ CPU-generated address can be split into:

\[ n \text{-bit address: } \begin{array}{c|c}
\text{Virtual Page Number} & \text{Page Offset} \\
\end{array} \]

▪ Request is Virtual Address (VA), want Physical Address (PA)
▪ Note that Physical Offset = Virtual Offset (page-aligned)

❖ Use lookup table that we call the page table (PT)

▪ Replace Virtual Page Number (VPN) for Physical Page Number (PPN) to generate Physical Address
▪ Index PT using VPN: page table entry (PTE) stores the PPN plus management bits (e.g. Valid, Dirty, access rights)
▪ Has an entry for every virtual page
Page Table Diagram

- Page tables stored in physical memory
  - Too big to fit elsewhere – managed by MMU & OS
- How many page tables in the system?
  - One per process
Page Table Address Translation

Virtual address (VA)

Virtual page number (VPN) Virtual page offset (VPO)

Page table

Valid PPN

Physical page number (PPN) Physical page offset (PPO)

CPU

Page table base register (PTBR)

Page table address for process

Valid bit = 0: page not in memory (page fault)

In most cases, the MMU can perform this translation without software assistance
Polling Question [VM II]

❖ How many bits wide are the following fields?
   - 16 KiB pages
   - 48-bit virtual addresses
   - 16 GiB physical memory
   - Vote at: http://pollev.com/pbjones

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>34</td>
</tr>
<tr>
<td>(B)</td>
<td>32</td>
</tr>
<tr>
<td>(C)</td>
<td>30</td>
</tr>
<tr>
<td>(D)</td>
<td>34</td>
</tr>
</tbody>
</table>
Page Hit

- **Page hit:** VM reference is in physical memory

Example: Page size = 4 KiB

Virtual Addr: 0x00740b

VPN: 

Physical Addr:

PPN:
Page Fault

- **Page fault:** VM reference is NOT in physical memory

Example: Page size = 4 KiB
Provide a virtual address request (in hex) that results in this particular page fault:

Virtual Addr: 

```
Example: Page size = 4 KiB
Provide a virtual address request (in hex) that results in this particular page fault:

```

```
Reminder: Page Fault Exception

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```c
int a[1000];
int main () {
    a[500] = 13;
}
```

80483b7: c7 05 10 9d 04 08 0d movl $0xd, 0x8049d10

- Page fault handler must load page into physical memory
- Returns to faulting instruction: mov is executed again!
  - Successful on second try
Handling a Page Fault

- Page miss causes page fault (an exception)

---

### Page Table (DRAM)

<table>
<thead>
<tr>
<th>Valid</th>
<th>PPN/Disk Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Physical memory (DRAM)

- VP 1
- VP 2
- VP 7
- VP 4

### Virtual memory (DRAM/disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7
Handling a Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
Handling a Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
Handling a Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
- Offending instruction is restarted: page hit!

![Diagram of page table and memory structure](image-url)
Virtual Memory (VM)

- Overview and motivation
- VM as a tool for caching
- Address translation
- VM as a tool for memory management
- VM as a tool for memory protection
VM for Managing Multiple Processes

- Key abstraction: each process has its own virtual address space
  - It can view memory as a simple linear array
- With virtual memory, this simple linear virtual address space need not be contiguous in physical memory
  - Process needs to store data in another VP? Just map it to any PP!

![Address translation diagram]

Virtual Address Space for Process 1:

- VP 1
- VP 2
- ...
- N-1

Virtual Address Space for Process 2:

- VP 1
- VP 2
- ...
- N-1

Physical Address Space (DRAM):

- PP 2
- PP 6
- PP 8
- ...
- M-1

(e.g., read-only library code)
Simplifying Linking and Loading

- **Linking**
  - Each program has similar virtual address space
  - Code, Data, and Heap always start at the same addresses

- **Loading**
  - `execve` allocates virtual pages for `.text` and `.data` sections & creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system
VM for Protection and Sharing

- The mapping of VPs to PPs provides a simple mechanism to *protect* memory and to *share* memory between processes
  - **Sharing**: map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - **Protection**: process can’t access physical pages to which none of its virtual pages are mapped (here: Process 2 can’t access PP 2)
Memory Protection Within Process

- VM implements read/write/execute permissions
  - Extend page table entries with permission bits
  - MMU checks these permission bits on every memory access
    - If violated, raises exception and OS sends SIGSEGV signal to process (segmentation fault)

<table>
<thead>
<tr>
<th>Process (i):</th>
<th>Valid</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process (j):</th>
<th>Valid</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 11</td>
</tr>
</tbody>
</table>
Review Question

What should the permission bits be for pages from the following sections of virtual memory?

<table>
<thead>
<tr>
<th>Section</th>
<th>Read</th>
<th>Write</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Literals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Address Translation: Page Hit

1) Processor sends *virtual* address to MMU (*memory management unit*)

2-3) MMU fetches PTE from page table in cache/memory
    (Uses PTBR to find beginning of page table for current process)

4) MMU sends *physical* address to cache/memory requesting data

5) Cache/memory sends data to processor

**Notes:**
- VA = Virtual Address
- PTEA = Page Table Entry Address
- PTE = Page Table Entry
- PA = Physical Address
- Data = Contents of memory stored at VA originally requested by CPU
Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in cache/memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction
Hmm... Translation Sounds Slow

- The MMU accesses memory twice: once to get the PTE for translation, and then again for the actual memory request
  - The PTEs may be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

- What can we do to make this faster?
  - Solution: add another cache! 🎉
Speeding up Translation with a TLB

- **Translation Lookaside Buffer (TLB):**
  - Small hardware cache in MMU
    - Split VPN into **TLB Tag** and **TLB Index** based on # of sets in TLB
  - Maps virtual page numbers to physical page numbers
  - Stores *page table entries* for a small number of pages
    - Modern Intel processors have 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory
A TLB hit eliminates a memory access!
TLB Miss

- A TLB miss incurs an additional memory access (the PTE)
  - Fortunately, TLB misses are rare
Fetching Data on a Memory Read

1) Check TLB
   - **Input**: VPN, **Output**: PPN
   - **TLB Hit**: Fetch translation, return PPN
   - **TLB Miss**: Check page table (in memory)
     - **Page Table Hit**: Load page table entry into TLB
     - **Page Fault**: Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB

2) Check cache
   - **Input**: physical address, **Output**: data
   - **Cache Hit**: Return data value to processor
   - **Cache Miss**: Fetch data value from memory, store it in cache, return it to processor
Address Translation

Virtual Address

TLB Lookup

TLB Miss
Check the Page Table

Page not in Mem
Page Fault (OS loads page)
Find in Disk

Page in Mem
Update TLB
Find in Mem

TLB Hit
Protection Check

Access Denied
Protection Fault
SIGSEGV

Access Permitted
Physical Address
Check cache

Miss
Hit
Address Manipulation

request from CPU: \( n \)-bit virtual address

split to access TLB: TLB Tag \( \rightarrow \) TLB Index \( \rightarrow \) Page Offset

(on TLB miss) access PT: Virtual Page Number \( \rightarrow \) Page offset

\( m \)-bit physical address:

split to access cache: Physical Page Number \( \rightarrow \) Cache Tag \( \rightarrow \) Cache Index \( \rightarrow \) Offset

TRANSLATION
Context Switching Revisited

❖ What needs to happen when the CPU switches processes?

▪ Registers:
  • Save state of old process, load state of new process
  • Including the Page Table Base Register (PTBR)

▪ Memory:
  • Nothing to do! Pages for processes already exist in memory/disk and protected from each other

▪ TLB:
  • *Invalidate* all entries in TLB – mapping is for old process’ VAs

▪ Cache:
  • Can leave alone because storing based on PAs – good for shared data
Memory Overview

\[ \text{movl 0x8043ab, \%rdi} \]
Summary of Address Translation Symbols

❖ Basic Parameters
  ▪ $N = 2^n$ Number of addresses in virtual address space
  ▪ $M = 2^m$ Number of addresses in physical address space
  ▪ $P = 2^p$ Page size (bytes)

❖ Components of the virtual address (VA)
  ▪ $VPO$ Virtual page offset
  ▪ $VPN$ Virtual page number
  ▪ $TLBI$ TLB index
  ▪ $TLBT$ TLB tag

❖ Components of the physical address (PA)
  ▪ $PPO$ Physical page offset (same as VPO)
  ▪ $PPN$ Physical page number
Virtual Memory Summary

❖ Programmer’s view of virtual memory
  ▪ Each process has its own private linear address space
  ▪ Cannot be corrupted by other processes

❖ System view of virtual memory
  ▪ Uses memory efficiently by caching virtual memory pages
    • Efficient only because of locality
  ▪ Simplifies memory management and sharing
  ▪ Simplifies protection by providing permissions checking
Memory System Summary

❖ Memory Caches (L1/L2/L3)
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware

❖ Virtual Memory
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Operating System (software)
    - Allocates/shares physical memory among processes
    - Maintains high-level tables tracking memory type, source, sharing
    - Handles exceptions, fills in hardware-defined mapping tables
  - Hardware
    - Translates virtual addresses via mapping tables, enforcing permissions
    - Accelerates mapping via translation cache (TLB)
Simple Memory System Example (small)

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes
Simple Memory System: Page Table

- Only showing first 16 entries (out of _____)
  - **Note:** showing 2 hex digits for PPN even though only 6 bits
  - **Note:** other management bits not shown, but part of PTE

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Simple Memory System: TLB

- 16 entries total
- 4-way set associative

Why does the TLB ignore the page offset?
Simple Memory System: Cache

- Direct-mapped with $K = 4$ B, $C/K = 16$
- Physically addressed

Note: It is just coincidence that the PPN is the same width as the cache Tag
# Current State of Memory System

## TLB:

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>V</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1B</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

## Page table (partial):

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Tag</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>28</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>02</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Tag</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>17</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>09</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>2D</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0D</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

## Cache:

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>V</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>A</td>
<td>0B</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>B</td>
<td>12</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>V</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1B</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>
Polling Question [VM III]
Memory Request Example #1

- Virtual Address: \(0x03D4\)

- Physical Address:

Give your answer for Data(byte) at: [http://pollev.com/pbjones](http://pollev.com/pbjones)
Memory Request Example #2

- **Virtual Address:** \(0x038F\)

  
  \[
  \begin{array}{cccccccccccc}
  \text{TLBT} & & & & & & & & & & & \text{TLBI} \\
  13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
  0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
  \end{array}
  \]

- **Physical Address:**

  \[
  \begin{array}{cccccccccccc}
  \text{CT} & & & & & & & & & & & \text{Cl} \\
  11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
  \text{PPN} & & & & & & & & & & & \text{PPO}
  \end{array}
  \]

**Note:** It is just coincidence that the PPN is the same width as the cache Tag.
Memory Request Example #3

❖ Virtual Address: \(0x0020\)

![Virtual Address Diagram]

VPN _____  TLBT _____  TLBI _____  TLB Hit? ___  Page Fault? ___  PPN _____

❖ Physical Address:

![Physical Address Diagram]

CT ______  CI _____  CO _____  Cache Hit? ___  Data (byte) _______

Note: It is just coincidence that the PPN is the same width as the cache Tag.
Memory Request Example #4

❖ Virtual Address: \(0x036B\)

Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Physical Address:
Practice VM Question

❖ Our system has the following properties
  ▪ 1 MiB of physical address space
  ▪ 4 GiB of virtual address space
  ▪ 32 KiB page size
  ▪ 4-entry fully associative TLB with LRU replacement

a) Fill in the following blanks:

_________ Entries in a page table  __________ Minimum bit-width of PTBR

_________ TLBT bits  __________ Max # of valid entries in a page table
Practice VM Question

- One process uses a page-aligned square matrix `mat[]` of 32-bit integers in the code shown below:
  ```c
  #define MAT_SIZE = 2048
  for(int i = 0; i < MAT_SIZE; i++)
      mat[i*(MAT_SIZE+1)] = i;
  ```

b) What is the largest stride (in bytes) between successive memory accesses (in the VA space)?
Practice VM Question

❖ One process uses a page-aligned square matrix $\text{mat}[\ ]$ of 32-bit integers in the code shown below:

```c
#define MAT_SIZE = 2048
for (int i = 0; i < MAT_SIZE; i++)
    mat[i*(MAT_SIZE+1)] = i;
```

c) Assuming all of $\text{mat}[\ ]$ starts on disk, what are the following hit rates for the execution of the for-loop?

<table>
<thead>
<tr>
<th>TLB Hit Rate</th>
<th>Page Table Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>__________</td>
<td>__________</td>
</tr>
</tbody>
</table>
Page Table Reality

❖ Just one issue... the numbers don’t work out for the story so far!

❖ The problem is the page table for each process:
  - Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
  - How many page table entries is that?
    1 PTE for every virtual page: $2^{n-p} = 2^{51}$ PTEs
  - About how long is each PTE?
    $PPN_{\text{width}} + \text{management bits} = 20 + 5 = 25 \text{ bits} \approx 3 \text{ bytes}$

❖ Moral: Cannot use this naïve implementation of the virtual→physical page mapping – it’s way too big
A Solution: Multi-level Page Tables

This is called a *page walk*

Page table base register (PTBR)

Virtual Address

Physical Address

This is extra (non-testable) material
Multi-level Page Tables

- A tree of depth $k$ where each node at depth $i$ has up to $2^j$ children if part $i$ of the VPN has $j$ bits
- Hardware for multi-level page tables inherently more complicated
  - But it’s a necessary complexity – 1-level does not fit
- Why it works: Most subtrees are not used at all, so they are never created and definitely aren’t in physical memory
  - Parts created can be evicted from cache/memory when not being used
  - Each node can have a size of $\sim$1-100KB
- But now for a $k$-level page table, a TLB miss requires $k + 1$ cache/memory accesses
  - Fine so long as TLB misses are rare – motivates larger TLBs