Virtual Memory II
CSE 351 Spring 2018

https://xkcd.com/1495/
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
Review: Terminology

- Context switch
  - Switch between processes on the same CPU

- Page in
  - Move pages of virtual memory from disk to physical memory

- Page out
  - Move pages of virtual memory from physical memory to disk

- Thrashing
  - Total working set size of processes is larger than physical memory and causes excessive paging in and out instead of doing useful computation
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!

![Diagram showing virtual address spaces and address translation](image)
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>
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

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
  - Maps virtual page numbers to physical page numbers
  - Contains complete *page table entries* for small number of pages
    - Modern Intel processors have 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory

![TLB Diagram](image)
A TLB hit eliminates a memory access!
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 [unlike we said!], **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 Page Table

Page not in Mem

Page Fault
(OS loads page)

Find in Disk

TLB Hit

Protection Check

Page in Mem

Update TLB

Find in Mem

Protection Fault

Access Denied

SIGSEGV

Access Permitted

Physical Address

Check cache
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
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
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?

  - About how long is each PTE?

- **Moral:** Cannot use this naïve implementation of the virtual→physical page mapping – it’s way too big
  - Wouldn’t work even if each PTE was one *bit*!
A Solution: Multi-level Page Tables

This is called a page walk

TLB

<table>
<thead>
<tr>
<th>VPN</th>
<th>PTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>PTE</td>
</tr>
<tr>
<td>VPN</td>
<td>PTE</td>
</tr>
</tbody>
</table>

Virtual Address

<table>
<thead>
<tr>
<th>VPN 1</th>
<th>VPN 2</th>
<th>...</th>
<th>VPN k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 page table</td>
<td>Level 2 page table</td>
<td>...</td>
<td>Level k page table</td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-1</td>
<td>p-1</td>
</tr>
</tbody>
</table>

Page table base register (PTBR)
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 (built for a specific tree shape!)
- 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 ~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 and larger page size
Wrap-Up

- Without VM, our prior view of program execution wouldn’t work!
  - No room in RAM for the full address space of a process
  - No way to isolate processes from each other

- Implementing VM efficiently requires substantial hardware complexity
  - Multi-level page tables, page faults, PTBR, TLB, ...
  - Provided by all processors that support OSes that protect separate processes

- OS has to service page faults by updating page tables

- Application programmers can largely ignore VM (a valuable abstraction!)
  - Which is why you probably had never heard of it 😊
  - Keep working set small with good locality to avoid thrashing