Solution 2: TLBs

- We have a large pile of data (i.e., the page table) and we want to access it very quickly (i.e., in one clock cycle)
- So, build a cache for the page mapping, but call it a “translation lookaside buffer” or “TLB”
TLBs

- TLBs are small (maybe 128 entries), highly-associative (often fully-associative) caches for page table entries.
- This raises the possibility of a TLB miss, which can be expensive.
  - To make them cheaper, there are “hardware page table walkers” -- specialized state machines that can load page table entries into the TLB without OS intervention.
  - This means that the page table format is now part of the big-A architecture.
  - Typically, the OS can disable the walker and implement its own format.
Solution 3: Defer translating Accesses

- If we translate before we go to the cache, we have a “physical cache”, since cache works on physical addresses.
  - Critical path = TLB access time + Cache access time

- Alternately, we could translate after the cache
  - Translation is only required on a miss.
  - This is a “virtual cache”
The Danger Of Virtual Caches (I)

- Process A is running. It issues a memory request to address 0x10000
  - It is a miss, and 0x10000 is brought into the virtual cache
- A context switch occurs
- Process B starts running. It issues a request to 0x10000
- Will B get the right data?

- No! We must flush virtual caches on a context switch.
The Danger Of Virtual Caches (2)

- There is no rule that says that each virtual address maps to a different physical address.
- When this occurs, it is called “aliasing”
- Example: An alias exists in the cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0xfff0000</td>
</tr>
<tr>
<td>0x2000</td>
<td>0xfff0000</td>
</tr>
</tbody>
</table>

- Store B to 0x1000

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>B</td>
</tr>
<tr>
<td>0x2000</td>
<td>A</td>
</tr>
</tbody>
</table>

- Now, a load from 0x2000 will return the wrong value
The Danger Of Virtual Caches (2)

- Why are aliases useful?
- Example: Copy on write
  - `memcpy(A, B, 100000)`
  - Adjusting the page table is much faster for large copies
  - The initial copy is free, and the OS will catch attempts to write to the copy, and do the actual copy lazily.
  - There are also system calls that let you do this arbitrarily.

Two virtual addresses pointing the same physical address
Avoiding Aliases

- If the system has virtual caches, the operating system must prevent alias from occurring in the cache.
- This means that any addresses that may alias must map to the same cache index.
  - If VA1 and VA2 are aliases,
  - VA1 mod (cache size) == VA2 mod (cache size)
- Since the OS controls the page map, and it creates any aliases that exist (e.g., via copy on write), it can ensure this property.
Solution (4): Virtually indexed physically tagged

key idea: page offset bits are not translated and thus can be presented to the cache immediately

Index L is available without consulting the TLB
⇒ cache and TLB accesses can begin simultaneously
   Critical path = max(cache time, TLB time)!!!
Tag comparison is made after both accesses are completed

Work if Cache Size ≤ Page Size (⇒ C ≤ P)
because then none of the cache inputs need to be translated
(i.e., the index bits in physical and virtual addresses are the same)
Virtualizing Memory

- We need to make it appear that there is more memory than there is in a system
  - Allow many programs to be “running” or at least “ready to run” at once (mostly)
  - Absorb memory leaks (sometimes... if you are programming in C or C++)
Page table with pages on disk

Virtual Address
31 22 21 12 11 0
p1  p2  offset

10-bit  10-bit
L1 index  L2 index

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

Level 2 Page Tables

Data Pages

Page in primary memory
Page on disk
PTE of a nonexistent page

Adapted from Arvind and Krste’s MIT Course 6.823 Fall 05
The TLB With Disk

- TLB entries always point to memory, not disks
The Value of Paging

- Disk are really really slow.
- Paging is not very useful for expanding the active memory capacity of a system
  - It’s good for “coarse grain context switching” between apps
  - And for dealing with memory leaks ;-)
- As a result, fast systems don’t page.
The Future of Paging

- Non-volatile, solid-state memories significantly alter the trade-offs for paging.
  - NAND-based SSDs can be between 10-100x faster than disk
- Is paging viable now? In what circumstances?
Other uses for VM

- VM provides us a mechanism for adding “meta data” to different regions of memory.
  - The primary piece of meta data is the location of the data in physical ram.
  - But we can support other bits of information as well

- 

- 

Other uses for VM

- VM provides us a mechanism for adding “meta data” to different regions of memory.
  - The primary piece of meta data is the location of the data in physical ram.
  - But we can support other bits of information as well

- Backing memory to disk
  - next slide

- Protection
  - Pages can be readable, writable, or executable
  - Pages can be cachable or un-cachable
  - Pages can be write-through or write back.

- Other tricks
  - Arrays bounds checking
  - Copy on write, etc.