Virtual Memory

Today’s Menu:

- Virtual Memory
- Virtual vs. Physical Address Spaces
- Page Table
- Address Translation
- Page Fault
- Replacement Policy
- Translation Lookaside Buffer (TLB)
- Protection

Memory Hierarchy of a Computer System

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

![Memory Hierarchy Diagram]

How Does a Program Start Running?

- The program is copied from permanent storage into memory - on PCs and workstations, the "operating system" copies the program (bits) from disk.
- CPU’s Program Counter is then set to starting address of program and program begins execution.

What if the Program is Too Big

- Some machines won’t let you run the program
- Original DOS

Memory Allocation for Multiprogramming

- Multiple jobs (multiprogramming) each require memory
- 4 jobs and their memory requirements
Memory Allocation for Multiprogramming (cont.)

- Job 5 doesn’t fit
  - There is enough empty memory to hold the program
  - But job 5 won’t contiguously fit in either hole
  - We call this external fragmentation

![Memory Allocation Diagram]

Virtual Memory

- What is virtual memory?
  - Technique that allows execution of a program that can reside in non-contiguous memory locations
  - Does not have to completely reside in memory
  - Allows the computer to "fake" a program into believing that its memory is contiguous
  - Memory space is larger than physical memory

- Why is VM important?
  - Cheap - no longer have to buy lots of RAM
  - Removes burden of memory resource management from the programmer
  - Enables multiprogramming, time-sharing, protection

Virtual Memory

- Main memory (physical memory) can act as a cache for the secondary storage (disk)

- Advantages:
  - Illusion of having more and contiguous physical memory
  - Program relocation by "pages"
  - Protection in multiprogramming

How Does VM Work

- Two memory "spaces"
  - Virtual memory space - what the program "sees"
  - Physical memory space - what the program runs in (size of RAM)

- On program startup
  - OS copies program into RAM
  - If there is not enough RAM, OS stops copying program & starts running the program with some portion of the program loaded in RAM
  - When the program touches a part of the program not in physical memory (RAM), OS copies that part of the program from disk into RAM (page fault exception)
  - In order to copy some of the program from disk to RAM, the OS must evict parts of the program already in RAM
  - OS copies the evicted parts of the program back to disk if the pages are dirty (i.e., if they have been written into, and changed - dirty bit just like caches)

Basic VM Algorithm

- Program uses virtual addresses (load, store, instruction fetch)
- Computer translates virtual address (VA) to physical address (PA)
- Computer reads RAM using PA, returning the data to program
Page Tables

- A table which holds VA -> PA translations is called the page table.
- In our current scheme, each word is translated from a virtual address to a physical address.
  - How big is the page table?

\[ \text{Processor (running program)} \]
\[ \text{Virtual Address} \rightarrow \text{Physical Address} \]
\[ \text{Instructions (or data)} \]
\[ \text{Page Tables (cont.)} \]

- Instead of a "fine-grain" VM where any word in VM can map to any RAM word location, we partition memory into bigger chunks called pages.
  - Typical page size today is 4KBytes.
  - But we're moving towards "large" 2MB pages.
- Reduces the number of VA->PA translation entries:
  - Only one translation per page.
  - For 4KByte page, that's one VA->PA translation for every 1,024 words.

\[ \text{Virtual Address} \rightarrow \text{Physical Address} \]

Aside: Caches and VM

- Idea is just like lines in caches

\[ \text{Address} \rightarrow \text{Tag} \rightarrow \text{Index} \rightarrow \text{Word offset} \]

Virtual Pages & Physical Page Frames

- For 4KBytes, the bottom 12 bits are not translated: only the upper bits choose the page, the lower bits choose where in the page.

\[ \text{Virtual Address Space} \rightarrow \text{Physical Address Space} \]

Page Table

- Each entry in the page table is called a Page Table Entry (PTE).
- # of bits in page offset = \( \log_2 \text{Page Size} \).
- # of bits in physical address = \( \log_2 (\text{# RAM Bytes}) \).
- Machine has 24 bits of physical address space.
- How much memory is that?
**EVICTING PAGES**

What Happens if Page is not in RAM?

- How do we know it’s not in RAM?
  - Page table entry’s valid bit is set to INVALID (DISK)
- What do we do?
  - Hardware asks OS to fetch the page from disk - we call this a page fault
  - Before page is read from disk, OS must evict a page from RAM (if RAM is full)
    - The page to be evicted is called the victim page
    - If the page to be evicted is dirty, write the page back to disk
    - Only data pages can be dirty
  - OS then reads the requested page from disk
  - OS changes the page table to reflect the new mapping
  - Hardware restarts at the faulting virtual address
Which Page Should We Evict?

- Optimal solution: evict a page that won’t be referenced (used) again.
- If all pages will be used again, then evict the page that will not be used for the longest period of time.
  - Guarantees the lowest possible page fault rate (if of faults per second).
  - Can’t be done unless we can tell the future.
- Other page replacement algorithms:
  - First-in, First-out (FIFO).
  - Least Recently Used (LRU).
- So LRU is really expensive to keep track of. Can we do this?
  - How long does it take to load the page from disk?

First-in, First-out (FIFO)

- Oldest page is evicted.
- How do we know which page is oldest?
  - Keep a list of pages ordered by reference.
  - When a page must be evicted, pick the page at the end (bottom) of the list.
- Example w/ 4 pages:

Least Recently Used (LRU)

- Evict the page that has not been used for the longest period of time.
- How do we know which page is oldest?
  - Keep a list of pages ordered by reference.
  - When a page must be evicted, pick the page at the end (bottom) of the list.
- Example w/ 4 pages:

Performance of Virtual Memory

- If every program in a multiprogramming environment fits into RAM, then virtual memory never "pages" (goes to disk).
- If any program doesn’t fit into RAM, then the VM system must page between RAM and disk.
  - Paging is very costly.
  - A disk access (4KBytes) can take ~10 ms... in 10 ms, a processor can execute ~40 Million instructions.
  - Basically, you really don’t want to page very often, if you don’t have to.

When Bad Things Happen to Good Memories...

- So, can anything go wrong here?
  - i.e., this sounds like a great idea. Have reasonable size RAMs, big disks.
  - What bad stuff can happen?
- Answer: thrashing.
  - Your program is so large, or, your machine has so many separate jobs (or users), that very little of the program(s) fit in RAM.
  - Your physical RAM is not providing you even minimal spatial locality for your address refs., so very very often, an address faults, and you page constantly.

Pathological Thrashing

- Consider the following reference pattern (in pages):
  - 1, 2, 3, 4, 5, 1, 2, 3, 4, 5.
  - What happens if there are only 4 page frames in RAM?
  - Not all of the program will fit into RAM.
  - Assume LRU.
Pathological Thrashing (cont.)

Consider the following reference pattern (in pages)
- 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- What happens if there are only 4 page frames in RAM?
- Not all of the program will fit into RAM
- Assume LRU

Pathological Thrashing (cont.)

Consider the following reference pattern (in pages)
- 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- What happens if there are only 4 page frames in RAM?
- Not all of the program will fit into RAM
- Assume LRU

Pathological Thrashing (cont.)

Consider the following reference pattern (in pages)
- 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- What happens if there are only 4 page frames in RAM?
- Not all of the program will fit into RAM
- Assume LRU

Pathological Thrashing (cont.)

Consider the following reference pattern (in pages)
- 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- What happens if there are only 4 page frames in RAM?
- Not all of the program will fit into RAM
- Assume LRU
- LRU can result in thrashing ... Random replacement or some other policy could avoid this problem

Memory Protection

Multiprogramming and Protection

Most machines run multiple processes (also called jobs, tasks)
- The processes share RAM
- Example: simultaneously running emacs, verilog and X-windows system
- Example: simultaneously running browser, email, chat client

Need mechanism to provide the memory to each process

Need mechanism to protect the memory of each process
- Avoid one process reading or writing the memory of another
- Avoid one process crashing another, especially the OS
- Usually referred to as "protection"
Protection with Address Bounds

- Simplest approach
  - Provide pair of registers that checks every address reference to make sure the address falls between the two limits
  - Example, when running Job 3, the two registers would point to the bottom and top of Job 3's memory range
- Base ≤ Address ≤ Bound
- How owns base & bound?
  - The OS
  - User process should not be allowed to alter these registers
- Only OS can alter these
  - How?
    - Processor provides at least two modes
    - User and system (OS)
    - System mode is a called a privileged mode
    - Code running in system mode can access parts of the machine (registers) that user mode cannot access (modify)

Problems with Address Bounding

- Loader has to re-locate code (jumps, etc.), memory loads, etc.
  - Or ISA can be all PC relative addressing
- What if program can’t fit into given memory slots
  - Only provides for contiguous data segments
- What if every process thought it had its very own virtual address space?
  - 4GB of space, all your own
  - Start/end where ever you want
  - No relocation necessary
  - But...need to translate between your virtual address and the physical memory you own

VM and Memory Management

- This is how EACH process in Linux sees memory:
  - Kernel Virtual Memory
    - User Stack
    - Shared Libraries
    - Run-time heap
    - Read/Write Segment
    - Read-only Segment
  - Some reserved for the kernel...
  - How to ensure that each process gets its own physical memory allocated?
  - How to ensure that one process cannot overwrite another’s memory?
  - 32-bit virtual address space
  - Some reserved for the kernel...

Mapping to Physical Memory

- Virtual memory can be used to share memory and protect memory from another process!
  - (And all sorts of cool things with virtual machines!)

Protection with Private Page Table

- Finer-grained protection is accomplished by protecting each page
  - Give each user process its own page table
  - Because every address necessarily goes thru VA→PA translation, the process can only access physical pages listed in its own page table
  - Example: process can only access physical pages (1, 3, 4, 5, 8)

Protection (cont.)

- Separate page tables (per process) provide page-level protection
  - OS creates and manages page tables so that no user-level process can alter any process’ page table
  - Page tables are mapped into kernel memory where only the OS can read or write
Protection (cont.)

- Notice that both processes use the same virtual addresses.
- Notice the physical addresses are never the same.
- Can process sharing of virtual addresses cause problems?
  - Yes (we’ll see when we talk about TLBs).

<table>
<thead>
<tr>
<th>Process 1's Page Table</th>
<th>Process 2's Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical memory</td>
<td>Physical memory</td>
</tr>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
</tr>
<tr>
<td>0x03</td>
<td>0x03</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
</tr>
<tr>
<td>0x05</td>
<td>0x05</td>
</tr>
<tr>
<td>0x06</td>
<td>0x06</td>
</tr>
<tr>
<td>0x07</td>
<td>0x07</td>
</tr>
<tr>
<td>0x08</td>
<td>0x08</td>
</tr>
<tr>
<td>0x09</td>
<td>0x09</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x0A</td>
</tr>
<tr>
<td>0x0B</td>
<td>0x0B</td>
</tr>
</tbody>
</table>

| Physical memory        | Physical memory        |
| 0x00                   | 0x00                   |
| 0x01                   | 0x01                   |
| 0x02                   | 0x02                   |
| 0x03                   | 0x03                   |
| 0x04                   | 0x04                   |
| 0x05                   | 0x05                   |
| 0x06                   | 0x06                   |
| 0x07                   | 0x07                   |
| 0x08                   | 0x08                   |
| 0x09                   | 0x09                   |
| 0x0A                   | 0x0A                   |
| 0x0B                   | 0x0B                   |

Protection and the TLB

- A process presents a Virtual Address to the TLB for translation.
- Each process could present virtual address 0x0000.

VM Support

- Each process has its own page table.
- Extra bits to allow/deny:
  - Read access
  - Write access
  - Access when in supervisor mode only (kernel code).
- Support shared pages (libraries or memory).
  - Read only for multiple processes.
- If a process accesses a memory location that it does not have proper access to:
  - Segmentation Fault.

Protection and the TLB (cont.)

- Many machines append a PID (process ID) to each TLB entry.
- OS maintains a Process ID Register (updated during the switch between processes, called a context switch).
- Another solution is to flush the TLB on a context switch; flush means "empty it".

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00002 000</td>
<td>0x105 000</td>
</tr>
</tbody>
</table>

Why is this better/worse?
Who Does the VA->PA Translation?

- Software would be too slow for every memory reference
  - If not in software, then must be done in hardware
  - Hardware needs to hold the translations (PTEs)
- Can hardware hold all of the PTEs (i.e., the entire page table)?
- New question: How big is the page table?
  - Assumes a 4K page and 32 bit virtual address space
  - 32 bits can address 4 Gigabytes of RAM (2^32 = 4G)
  - 12 bits are for the page offset
  - 20 bits for the page number \( \rightarrow 2^{20} \) virtual pages
  - \( 2^{20} = 1,048,576 \) PTEs
  - Each PTE is at least 4 Bytes
  - 1 Million PTEs * 4 Bytes/PTE \( \approx \) 4 Megabytes
  - Too large to hold entire page table inside CPU chip itself
  - Most processors "cache" the most recently used PTEs

Where? Translation Lookaside Buffer (TLB)

- TLB Caches most recently referenced PTEs
  - Very similar to instruction and data caches
  - Can be accessed in <= single cycle
  - Remember 1.33 accesses/instruction

Making Address Translation Fast

- A cache for address translations: translation lookaside buffer

Where is My Page? My Page Table Entry?

- Page in physical mem, PTE in TLB
  - Life is good. Best answer: TLB lookup hits, VA->PA fastest, address resolved, fast access to data.
- Page in physical mem, PTE not in TLB, PTE only in phys mem
  - Life is so-so. TLB lookup misses. TLB reloads from page table out in memory. (10s-1000s cycles)
  - VA-PA takes longer, but at least your address is already in physical memory.
    (People actually optimize code to avoid this!)
- Page not in physical mem, PTE in TLB
  - Life stinks. TLB hits, quickly tells you your address is not in physical memory. Page fault, you want to go load the page.
  - (Disk latency \( \approx \) 10ms – you wait that long.)
- Page not in phys mem, PTE not in TLB, PTE only in phys mem
  - Life sucks. TLB lookup misses. TLB reloads from page table out in memory. During that reload, you discover your page itself is not in physical memory. Page fault, you want to go load the page.
  - (Disk latency \( \approx \) 10ms – you wait that long. PLUS 10s-1000s of cycles to load TLB)

TLB Example: Address Ref 1

Assume a 4K-Byte pages and a 32 bit address space

Virtual Address

<table>
<thead>
<tr>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0002</td>
</tr>
<tr>
<td>0x01</td>
</tr>
<tr>
<td>0x02</td>
</tr>
<tr>
<td>0x03</td>
</tr>
<tr>
<td>0x04</td>
</tr>
<tr>
<td>0x0000</td>
</tr>
<tr>
<td>0x05</td>
</tr>
<tr>
<td>0x06</td>
</tr>
<tr>
<td>0x07</td>
</tr>
<tr>
<td>434</td>
</tr>
</tbody>
</table>

Fully Assoc. TLB

Physical Page Frame

<table>
<thead>
<tr>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>434</td>
</tr>
</tbody>
</table>

TLB Example: Address Ref 1 Misses in TLB

Virtual Address

<table>
<thead>
<tr>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0002 434</td>
</tr>
</tbody>
</table>

Fully Assoc. TLB

Physical Page Frame

<table>
<thead>
<tr>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>434</td>
</tr>
</tbody>
</table>

CPU

TLB

Memory
TLB Example: Address Ref 1 Misses in TLB

Virtual Address
0x00002 434

Fully Assoc. TLB

Page Table:

Physical Page Frame

Physical Address
0x105 434

TLB Example: Address Ref 2 Hits in TLB

Virtual Address
0x00002 438

Fully Assoc. TLB

Page Table:

Physical Page Frame

Physical Address
0x105 438

TLB Example: Address Ref 3 Misses in TLB

Virtual Address
0x00007 1CA

Fully Assoc. TLB

Page Table:

Physical Page Frame

Physical Address
0x094 1CA

TLB Example: Address Ref 3 Misses in TLB

Virtual Address
0x00007 1CA

Fully Assoc. TLB

Page Table:

Physical Page Frame

Physical Address
0x094 1CA

TLB Example: Ref 4 Misses in TLB, TLB Conflict

Virtual Address
0x00005 000

Fully Assoc. TLB

Page Table:

Physical Page Frame

Physical Address
000

TLB Example: Ref 4 Misses in TLB, TLB Conflict

Virtual Address
0x00005 000

Fully Assoc. TLB

Page Table:

Physical Page Frame

Physical Address
000
### TLB Example: Ref 4 Misses in TLB, TLB Conflict

#### Virtual Address

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>TAG</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00005</td>
<td>0x81</td>
<td>0x02</td>
</tr>
</tbody>
</table>

#### Physical Address

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>0x081</th>
</tr>
</thead>
</table>

### Real Life Examples: Pentium Pro

- **Data TLB:**
  - 44 entries, 4 way SA
- **Data Cache:**
  - 8kB, 2 way SA
- **Instruction TLB:**
  - 22 entries, 4 way SA
- **Instruction Cache:**
  - 8kB, 4 way SA

### Summary

- **Virtual memory**
  - Gives illusion of a LARGE physical RAM, even if you have LESS real RAM
  - RAM divided into chunks called pages.
  - “Live” pages are in the physical RAM.
  - Pages that don’t fit aren’t on the disk.
  - Hardware translates the virtual address (big address space) into the physical address (real RAM address).
  - Every application sees the same virtual address space. (Simplifies multiprogramming.)
  - Provides fine-grained protection (per-page)
  - Provides efficient use of real memory (non-contiguous mappings)

- **But...**
  - Adds another level of indirection that you need to go through on every memory access
  - Another cache to maintain, and you have to go to the OS if the cache misses
  - Oh, and it complicates caches...

### TLB Structure & Performance

- **TLB structure**
  - 32 to 1024 entries (slots)
  - Can be direct mapped, set associative, fully associative
  - Easier to be fully associative here, since TLBs are often pretty small

- **TLB miss cost**
  - 20 to 1000 cycles (much longer if the page is on disk)
  - Hardware and software based

- **TLB miss rates**
  - 4% to 8% for typical Unix workloads
  - Can be much higher for large applications
  - Operating systems can significantly influence the TLB miss rate

- **Page size**
  - 4K or 8K Bytes
  - Often support multiple page sizes
    - up to 2GB(!), 2MB is becoming more widely used (“large pages”)

### Next Issue: Page Table Size

- **Previously determined: Linux page table size**
  - 4K page
  - 32-bit virtual address space
  - 20 bits for the page number \( \rightarrow \) 2^20 virtual pages
  - \( 2^{20} = 1,048,576 \) PTEs
  - Each PTE is at least 4 Bytes
  - 1 Million PTEs \( \times \) 4 Bytes/PTE = 4 Megabytes to map 32 bit virtual address space

- **Hey: you said each process has its own page table!**
  - 100 processes \( \times \) 400 MB of memory for page tables!!!
  - If the page table is not in memory, you can’t find it
  - The page tables cannot be swapped out

- **What’s the deal?**
Solution: Multi-level page tables

Example: from Linux

- Have two levels of page tables

- PTE 0
- PTE 1

- PTE 2 (null)
- PTE 3 (null)

- PTE 1023
- PTE 0

- PTE 1

- PTE 1023

Each of these is actually one page (4K) in size!

Only this table needs to stay in memory at all times.

Multi-level Page Tables

- VPN 1
- VPN 2
- Offset

- Level 1 Table (Page Directory)
- Level 2 Table (Page Table)

- PPN
- Offset

Multi-level Advantages

- Minimizes page: 4KB+4KB table
- Allows for maximum: 4KB + 4MB
- Page table size scales with memory usage

- Only level 1 MUST be in real memory
  - Level 2 tables can be swapped just like any other page

- Multi-level and TLB integration
  - TLB only stores Level 2 PTEs
  - Don’t have to do two TLB accesses to do VA -> PA translation

CACHES AND TLBS

Next Issue: Cache & TLBs, How They Interact

- We do memory hierarchies to hide unpleasant facts
  - We don’t have as much fast memory as we would ideally like. Solution?
  - Cache hierarchy gives illusion of speed—most of the time. Occasionally it’s slow.
  - We don’t have as much memory as we would like. Solution?
  - VM hierarchy gives illusion of size—most of the time. Occasionally it’s slow.

- Roughly put: We do cache for speed. We do VM for size.

- So, we have “regular” cache for fast access, and we have a TLB for fast translation VA->PA.
  - How do they interact? They must interact somehow...
  - Do we wait for VA->PA translation before looking in the cache?
  - Is the cache full of virtual or physical addresses?

Simplest Scheme is Sequential: TLB then Cache

- Slowest, but simplest
  1. CPU sends out virtual address
  2. TLB translates to physical, or page faults and we wait for page to load
  3. On TLB hit, translated physical address sent to cache
  4. Cache lookup gives data access fast, or...
  5. Cache miss goes to main mem

Slow because you have to wait for the translation before you can check the cache.
Address Translation/Cache Lookup

Virtual Address

Physical Address

VPN

PPN

TAG

IDX

PO

Cache

Data

Hit/Miss

Physical Address & Physical Tag to the cache – have to do address translation first

Real Example: DECstation 3100

TLBs and Caches: Basic Flow for Access

Resource Usage

Speed & Timing Impacts

- If we do these accesses sequentially, big impact on speed
  - You have to do lookup in the TLB
  - Then you have to do lookup in the cache
  - Involves a lot of memory access time

- One Solution: pipelining
  - Spread the accesses across stages of the pipeline
  - TLB and cache are just like any other resources in the pipeline
  - You gotta be careful to know how long they take (impacts pipe cycle time)
  - You gotta know who is trying to use the resource in what pipe stage
  - You can have hazards, need state, need forwarding paths, etc

Ex: MIPS R3000 Instruction Pipeline
Speeding it Up

TLB and then Cache... Why? What else could we do?

Two options
1. Overlapped cache & TLB access (in parallel)
   - What are the limitations?
2. Why does our cache use physical addresses?
   - Could it store virtual addresses?
   - What are the problems/considerations?

Overlapped Cache & TLB Access

How Overlapping Reduces Translation Time

Basic plumbing
- High order-bits of the VA are used to look in the TLB...
  - Remember: low order bits are the page offset—which byte address on the page
  - High order bits are what really changes, from virtual to physical address

...while low-order bits are used as index into cache
- Remember: lowest bits are the cache line offset—which byte on the cache line
- The "intermediate" bits are the cache index. Which line in the cache should we check to see if the address we want is actually cached in the cache memory?
- The highest order bits are the cache tag. When we look in the cache at a time of cache, we must compare these bits with the cache tag. To see if a cacheline in the cache is really the address we want, or just another line of memory that happens to map to the same place in this cache

The "action" here is on the cache tag high order bits...

Cache vs. TLB Access

What happens for large caches?
Should we use VA instead?
Should we only use VA bits to index?

Two Cache Architectures

Remember:
- Cache index used to look up data in the cache
- Cache tag used to verify what data is in the cache

1. Virtually-indexed Virtually-tagged Caches
   - Also known as Virtually-Addressed or Virtual Address Caches
   - The VPN bits are used for both tags and index bits
2. Virtually-indexed Physically-tagged Caches
   - The VPN bits only contribute to the index
   - The tag is physical and requires a TLB lookup, but it can be done in parallel
**Virtual Address Cache**

- Lookup using VA
- TLB access on miss
- Use PA to access next level (L2)

**Multiple Virtual Address Spaces (Multiprogramming)**

- Load/store to VA (page 0x02)
- Is it VA from Process 1 or Process 2?
- Is it a hit/miss?

**Multiple Address Space Solution**

1. Keep "process id" with cache block tags
   - Upon cache lookup check both address tag and process id
2. Flush cache on context switch
   - Expensive (lose contents of the cache every switch)

**Or, Only Use Virtual Bits to Index Cache**

- Don’t need to wait for TLB
- Parallel TLB access (e.g., for larger caches)
- Physically-tagged but virtually-indexed Cache
- Can distinguish addresses from different processes
- But, what if multiple processes share memory?

**Virtual Address Synonyms**

- P1 makes a ref to data on page 0x04
- P1 block is a miss
- But, P2 block is in
- Can’t look it up!

**Virtual Address Synonyms (Cont.)**

- Virtual addresses on page 0x04 of P1 are synonyms of those on page 0x00 of P2
- Synonyms are also referred to as aliases
- The page is shared among the processes
  - Example shared pages are kernel data structures
- Must avoid allowing multiple synonyms to co-exist in the cache
  - Only memory read/written must be resolved
  - Read-only memory (e.g., instructions) can exist in multiple locations
Synonym Solutions

- Avoid: Limit cache size to page size times associativity
  - get index from page offset

- Avoid: Eliminate by OS convention
  - single virtual space
  - restrictive sharing model

- Detect: Search all sets in parallel
  - 64K, 4-way cache, 4K pages, search 4 sets (16 entries)

- Reduce search space: Restrict page placement in OS
  - make sure index(VA) = index(PA)

Summary

- Memory access is hard and complicated!
  - Speed of CPU core demands very fast memory access. We do cache hierarchy to solve this one. 
    - Gives illusion of speed -- most of the time. Occasionally slow.
  - Size of programs demands large RAM. We do VM hierarchy to solve this one.
    - Gives illusion of size -- most of the time. Occasionally slow.

- VM Hierarchy
  - Another form of cache, but now between RAM and disk.
    - Atomic units of memory are pages, typically 4KB to 2MB.
    - Page table serves as translation mechanism from virtual to physical address
    - Page table lives in physical memory, managed by OS
      - For 64-bit addresses, multi-level tables used, some of the table is in VM
    - TLB is yet another cache -- caches translated addresses, page table entries.
      - Saves from having to go to physical memory to do lookup on each access
      - Usually very small, managed by OS
    - VM, TLB, cache have “interesting” interactions.
      - Big impacts on speed, pipelineing. Big impacts on exactly where the virtual to physical mapping takes place.