**Virtual Memory**

**Overview and Motivation**

- Programs refer to virtual memory addresses

- Conceptually, memory is just a large array of bytes; system provides private address space to each process

- Issue is we probably don’t have 2w bytes of physical memory; certainly don’t have 2w bytes of physical memory for every process

- But processes should not interfere with one another, will get segmentation fault otherwise

- Problem 1: How does everything fit

- 64-bit virtual addresses can address several exabytes, while physical main memory offers a few gigabytes

- Problem 2: Memory management

- Multiple processes, each with stack/heap/.text/.data

- How is this mapped to physical memory

- Problem 3: How to protect

- Multiple processes accessing same chunk of physical memory

- Problem 4: How to share

- Share data from chunk of physical memory to multiple processes

- Solve these problems by adding another level of indirection

- Instead of every process having direct access to one chunk of memory, each process has direct access to a pointer that tells where the memory is

- If you were to move something, without indirection each process would have to be changed to point to the new thing

- Virtual memory maps to physical memory, allowing each process to get its own private virtual address space

- Every byte in memory has one physical address and zero, one, or more virtual addresses (VAs)

- A virtual address can be mapped to either physical memory or disk

- Unused VAs may not have a mapping, and VAs from different processes may map to same location in memory/disk

**VM as a tool for caching**

- Virtual memory uses limited main memory (RAM) efficiently

- Use RAM as a cache for the parts of a virtual address space

- Keep only active areas of virtual address space in memory

- Virtual memory simplifies memory management for programmers while isolating address spaces

- *Pages* of virtual memory are usually stored in physical memory, but sometimes spill to disk

- Pages are another unit of aligned memory

- Each virtual page can be stored in any physical page (no fragmentation)

- VM avoids disk accesses because of locality

- Working set is the set of virtual pages that a program is actively accessing at any point in time

- Working set < physical memory: good performance for one process after compulsory misses

- Working set > physical memory: thrashing – performance meltdown where pages are swapped between memory and disk continuously

- Large page size, fully associative, write-back cache

**Address Translation**

- How do we perform the virtual 🡪 physical address translation

- CPU-generated address can be split into:



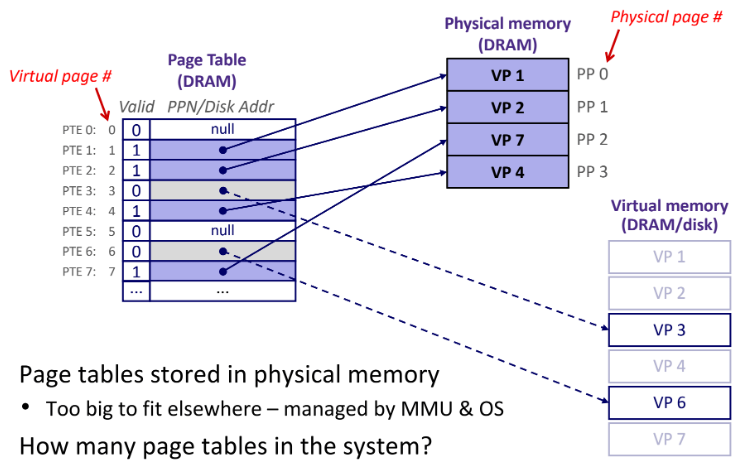
- Request is virtual address, want physical address

- Physical offset = virtual offset (page-aligned)

- Use lookup table called page table (PT)

- Replace virtual page number for physical page number to generate physical address

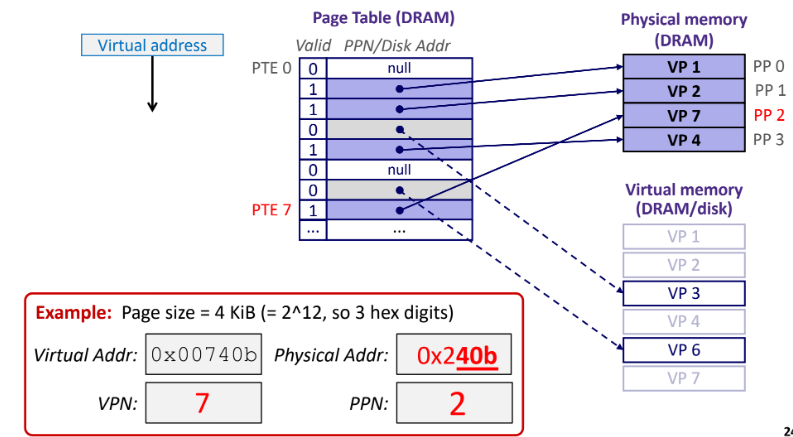
- Page tables stored in physical memory; each process has its own page table



- Page table address translation done by MMU

- Map VPN to page table and replace it with physical page number

- Page hit occurs because VM reference is in physical memory



- A page fault occurs when the VM reference is not in physical memory

- Page fault exception is when the user writes to a page of memory that is currently on disk

- Page fault handler must load page into physical memory; instruction successful on second try

- A page miss causes a page fault

- Page fault handler selects a victim to be evicted

- Offending instruction is restarted and should result in a page hit

**Virtual Memory as a tool for memory management and protection**

- With virtual memory, the linear virtual address space need not be contiguous in physical memory

- Simplifies linking and loading

- Linking: each program has similar virtual address space; code, data, and heap always start at same addresses

- Loading: execve allocates virtual pages for .text and .data sections & creates PTEs marked as invalid

- Mapping of VPs to PPs provides a simple mechanism to protect memory and share memory between processes

- Sharing: map virtual pages in separate address spaces to same physical page

- Protection: process can’t access physical pages to which none of its virtual pages are mapped

- VM implements read/write/execute permissions; extend page table entries with permission bits

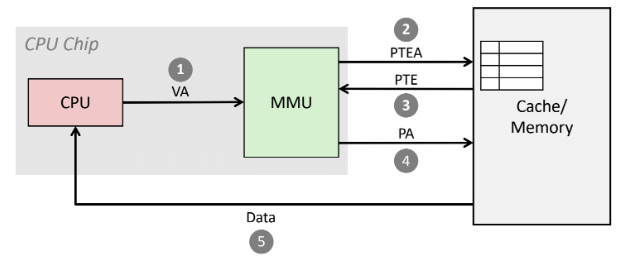
- Address translation: page hit

1. Processor sends virtual address to MMU

2-3. MMU fetches PTE from page table in cache/memory. MMU 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



- 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 0, so MMU triggers page fault exception

5. Handler identifies victim

6. Handler pages in new page and updates PTE in memory

7. Handler returns to original process, restarting faulting instruction

- Translation is slow because MMU accesses memory twice, once to get PTE for translation and again for actual memory request

- Use another cache

- Translation Lookaside Buffer (TLB)

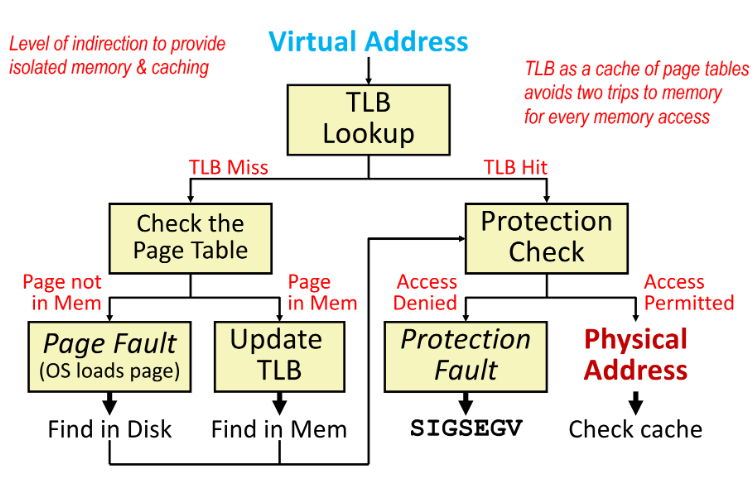
- Small hardware cache in MMU that maps virtual page numbers to physical page numbers

- Contains complete page table entries for small number of pages

- Much faster than page table lookup in cache/memory

- TLB hit eliminates a memory access

- TLB miss incurs an additional memory access (PTE) but is rare



- When the CPU switches processes:

- Registers save state of old process, load state of new process; including page table base register

- Memory has nothing to do; pages for processes already exist in memory/disk and are protected from each other

- TLB invalidates all entries in it because mapping is for old processes

- Cache can be left alone because it is storing based on PAs, which is good for sharing data