Virtual memory

- Virtual memory is just another level in the memory hierarchy
  - It allows main memory to cache pages (blocks) normally stored on disk
  - As with caches, the operations performed by virtual memory are transparent to properly-running user programs
- Virtual memory's similarity to caching
  - Block ≡ page
    - Blocks in caches are equivalent to pages in virtual memory
    - Pages are anywhere from 1 KB to 64 KB (though today's page sizes are usually 4+ KB)
  - Miss ≡ page fault
    - A miss in a cache is analogous to a page fault
    - The only difference is the penalty...
      - Millions of clock cycles for VM
      - Tens of clock cycles for caches

Virtual memory

- Miss rate
  - The miss rate for VM is very low -- less than 0.001%
  - Fewer than one in a million accesses causes a VM miss (often lower)
- Size
  - Memory caches are 16 KB - 1 MB or more
  - VM "cache" is 16 MB to 1024 MB or more — a factor of 1000 larger
- Differences include:
  - Replacement mechanism.
    - In caches, it is primarily controlled by the hardware
    - In VM, replacement is primarily controlled by the OS
  - The number of bits in the address determines the size of VM where cache size is independent of the address size
- Two kinds of VM: paging systems and segmentation systems

Basic virtual memory caching questions

- Where can a block be placed?
  - Since miss penalties are very high, OS designers always choose lower miss rates over simple placement algorithms
  - VM is almost always fully-associative (blocks can be placed anywhere in main memory)
- Which block is replaced?
  - Most operating systems use LRU or an approximation to it
  - The page table often includes a reference bit to help do LRU replacement

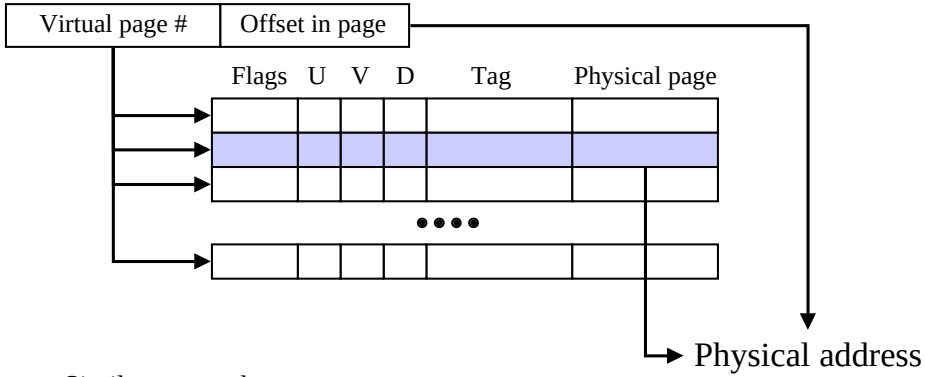
Basic virtual memory caching questions

- How is a block found?
  - Paging systems use a page table to translate virtual page numbers into physical page numbers
  - The physical address is constructed by concatenating the physical page number (found in the table) to the offset
  - Segmented systems use a similar structure except that the segment's physical address is ADDED to the offset
  - The page table needs enough entries to map the entire virtual address space since it is accessed using virtual page numbers
    - This results lots of space dedicated just to the page table
    - One optimization is to use hashing to restrict the number of page table entries to the number of physical pages (inverted page table)
  - Translation lookaside buffers (TLBs) are used to cache these translations, and reduce address translation time

Basic virtual memory caching questions

- What happens on a write?
  - VM is always writeback (capture as many writes as possible before writing the page to disk)
  - Write-through doesn't make sense => very large access penalty
  - The page table uses a dirty bit to keep track which pages have been modified and must be written to disk before they are replaced
  - Don't write pages back to disk unless they've been modified
- Page tables imply that a memory reference requires at least two memory accesses
  - One (or more) for the page table
  - One to get the data
- A TLB, which caches previous translations, can be effective in reducing memory references to the page table (uses locality)

Translation lookaside buffer (TLB)



- Similar to a cache
  - Tag holds the virtual address
  - Data portion holds the physical page frame number, protection field, valid bit, use bit and a dirty bit

Translation lookaside buffer (TLB)

- As with normal caches, the TLB may be fully-associative, direct-mapped, or set-associative
- Replacement may be done in hardware or may be assisted by software
  - For example, a miss in the TLB causes an exception which is handled by the OS, which places the appropriate page information into the TLB
  - Hardware handling is (usually) faster, but software is more flexible
- Small, fast TLBs are crucial because they are on the **critical path** to accessing data from the cache
  - ⇒ This is particularly true if the cache is physically addressed

Selecting page size

- Large page sizes are generally better because
  - They reduce the size of the page table
  - They are more efficient to transfer between memory and disk
  - They allow a TLB to cache translations for more of memory
- The biggest drawback to large pages is that they may waste memory: *internal fragmentation*
  - Assuming a process has three primary segments (text, heap and stack)
  - The average wasted storage per process will be 1.5 times the page size
  - When page size is 4 KB or 8 KB, this is negligible for machines with megabytes of memory
  - For larger pages, e.g., 64+ KB, lots of storage may be wasted
- Variable size pages can be used to get advantages of large pages without internal fragmentation

Memory protection

- VM is often used to protect a program from other programs
  - ⇒ Protection mechanisms must have hardware support
- Base & bounds
  - Each reference must fall between two addresses, given by the base & bound registers
  - This method also allows some relocation
  - User processes cannot be allowed to change these registers, but the OS must be able to do so on a process switch
- Therefore, the hardware must provide:
  - At least two modes of operations, user and kernel mode and a mechanism to switch between them
  - A protection mechanism for other portions of the CPU state to prevent user processes from being malicious

Using virtual memory for protection

- To ensure protection, CPU provides:
  - User/supervisor mode bit(s): separation of user & OS functions
  - Interrupt enable/disable bit(s): atomic operations
- Virtual memory offers a more fine-grained alternative
  - Each process has its own page table, which it cannot modify itself
  - Permission flags are provided with each segment or page
    - Read/write
    - Execute
  - *Concentric rings of security* and *capability lists* are more fine-grained alternatives, allowing more than two levels of protection

⇒ The OS course discusses VM in more detail

Effect of CPU design on memory hierarchy

- Superscalar & vector execution
  - A superscalar or vector machine may fetch several words per cycle
  - Clearly, the memory system must deliver the bandwidth to handle this; otherwise the benefit is lost
  - The brunt of the load falls upon the L1 cache
    - Bandwidth can be increased by widening the path to the cache or by providing extra ports to the cache
    - However, cache access is often the bottleneck in modern CPUs
- Speculative execution
  - Speculative execution and conditional instructions may generate invalid addresses that would not occur otherwise
  - The memory system must recognize and suppress these exceptions
  - Similarly, it must not stall the cache on a miss caused by a speculative instruction

I/O and cache consistency

- I/O devices move data from peripherals to memory
- This has two pitfalls:
  - Data written into memory is not automatically updated in the cache
  - Data in a writeback cache is not written to memory immediately so memory has stale data
- One solution is to flush blocks from the cache that are used in the I/O operation
  - Before the I/O for a write (so the write operation uses up-to-date information)
  - After the I/O for the read (before the I/O should work as well. The CPU should not access the data as it is being read into memory)
- An alternate method is simply to mark the blocks from I/O buffers as *uncacheable*

I/O and cache consistency: other solutions

- Watch the I/O buses for addresses in the tag
  - This eliminates the consistency problem
  - The drawback is that the checking slows down the cache
- Do I/O directly into the cache
  - This method guarantees consistency but it slows down the cache since both the CPU and I/O access it
  - Moreover, it displaces data in the cache with new data that is unlikely to be accessed soon by the CPU,

Stuff to beware of...

- Don't predict cache performance of code A from code B
  - Programs vary widely in how they use cache
  - A scientific program may have a small tight code loop but access large quantities of data
  - On the other hand, a word processing program might operate on relatively little data but use lots of code
- Simulate plenty of memory references
  - A CPU executes 500 million or more instructions per second
  - Simulating cache behavior using traces of only a few million traces can be misleading
  - Program locality behavior isn't constant over the entire program run
- Don't ignore the OS
  - Context switches can have a devastating effect on performance
  - The OS can miss or interfere with application programs, causing misses