RECAP: VIRTUAL MEMORY AND CACHE

Previous Next

Why virtual memory?

- With a 32-bit address you can access 4 GB of physical memory (you will never get the full memory though)
  - Seems enough for most day-to-day applications
  - But there are important applications that have much bigger memory footprint: databases, scientific apps operating on large matrices etc.
  - Even if your application fits entirely in physical memory it seems unfair to load the full image at startup
  - Just takes away memory from other processes, but probably doesn't need the full image at any point of time during execution: hurts multiprogramming
- Need to provide an illusion of bigger memory: Virtual Memory (VM)

Virtual memory

- Need an address to access virtual memory
  - Virtual Address (VA)
- Assume a 32-bit VA
  - Every process sees a 4 GB of virtual memory
  - This is much better than a 4 GB physical memory shared between multiprogrammed processes
  - The size of VA is really fixed by the processor data path width
  - 64-bit processors (Alpha 21264, 21364; Sun UltraSPARC; AMD Athlon64, Opteron; IBM POWER4, POWER5; MIPS R10000 onwards; Intel Itanium etc., and recently Intel Pentium4) provide bigger virtual memory to each process
  - Large virtual and physical memory is very important in commercial server market: need to run large databases

Addressing VM

- There are primarily three ways to address VM
  - Paging, Segmentation, Segmented paging
  - We will focus on flat paging only
- Paged
  - The entire VM is divided into small units called pages
  - Virtual pages are loaded into physical page frames as and when needed (demand paging)
  - Thus the physical memory is also divided into equal sized page frames
  - The processor generates virtual addresses
  - But memory is physically addressed: need a VA to PA translation

Previous Next

VA to PA translation

- The VA generated by the processor is divided into two parts:
    - Page offset and Virtual page number (VPN)
    - Assume a 4 KB page: within a 32-bit VA, lower 12 bits will be page offset (offset within a page) and the remaining 20 bits are VPN (hence 1 M virtual pages total)
    - The page offset remains unchanged in the translation
    - Need to translate VPN to a physical page frame number (PPFN)
    - This translation is held in a page table resident in memory: so first we need to access this page table
    - How to get the address of the page table?
- Accessing the page table
    - The Page table base register (PTBR) contains the starting physical address of the page table
    - PTBR is normally accessible in the kernel mode only
    - Assume each entry in page table is 32 bits (4 bytes)
    - Thus the required page table address is

      PTBR + (VPN << 2)

    - Access memory at this address to get 32 bits of data from the page table entry (PTE)
    - These 32 bits contain many things: a valid bit, the much needed PPFN (may be 20 bits for a 4 GB physical memory), access permissions (read, write, execute), a dirty/modified bit etc.

Page fault

- The valid bit within the 32 bits tells you if the translation is valid
- If this bit is reset that means the page is not resident in memory: results in a page fault
- In case of a page fault the kernel needs to bring in the page to memory from disk
- The disk address is normally provided by the page table entry (different interpretation of 31 bits)
- Also kernel needs to allocate a new physical page frame for this virtual page
- If all frames are occupied it invokes a page replacement policy

VA to PA translation

- Page faults take a long time: order of ms
    - Need a good page replacement policy
- Once the page fault finishes, the page table entry is updated with the new VPN to PPFN mapping
- Of course, if the valid bit was set, you get the PPFN right away without taking a page fault
- Finally, PPFN is concatenated with the page offset to get the final PA

PPFN	Offset

- Processor now can issue a memory request with this PA to get the necessary data
- Really two memory accesses are needed
- Can we improve on this?

TLB

- Why can't we cache the most recently used translations?
  - Translation Look-aside Buffers (TLB)
  - Small set of registers (normally fully associative)
  - Each entry has two parts: the tag which is simply VPN and the corresponding PTE
  - The tag may also contain a process id
  - On a TLB hit you just get the translation in one cycle (may take slightly longer depending on the design)
  - On a TLB miss you may need to access memory to load the PTE in TLB (more later)
  - Normally there are two TLBs: instruction and data

Caches

- Once you have completed the VA to PA translation you have the physical address. What's next?
- You need to access memory with that PA
- Instruction and data caches hold most recently used (temporally close) and nearby (spatially close) data
- Use the PA to access the cache first
- Caches are organized as arrays of cache lines
- Each cache line holds several contiguous bytes (32, 64 or 128 bytes)

Previous Next

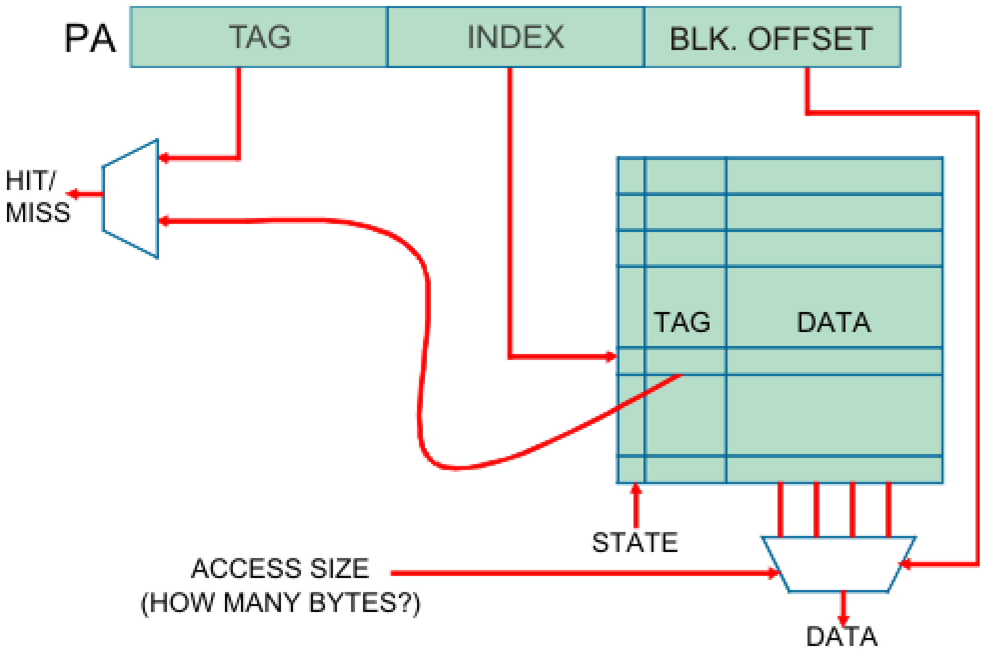
Module 4: "Recap: Virtual Memory and Caches"

Lecture 7: "Virtual Memory, TLB, and Caches"

Addressing a cache

- The PA is divided into several parts

- The block offset determines the starting byte address within a cache line
- The index tells you which cache line to access
- In that cache line you compare the tag to determine hit/miss



- An example
    - PA is 32 bits
    - Cache line is 64 bytes: block offset is 6 bits
    - Number of cache lines is 512: index is 9 bits
    - So tag is the remaining bits: 17 bits
    - Total size of the cache is 512*64 bytes i.e. 32 KB
    - Each cache line contains the 64 byte data, 17-bit tag, one valid/invalid bit, and several state bits (such as shared, dirty etc.)
    - Since both the tag and the index are derived from the PA this is called a physically indexed physically tagged cache

Previous Next