COM SCI M151B Week 7

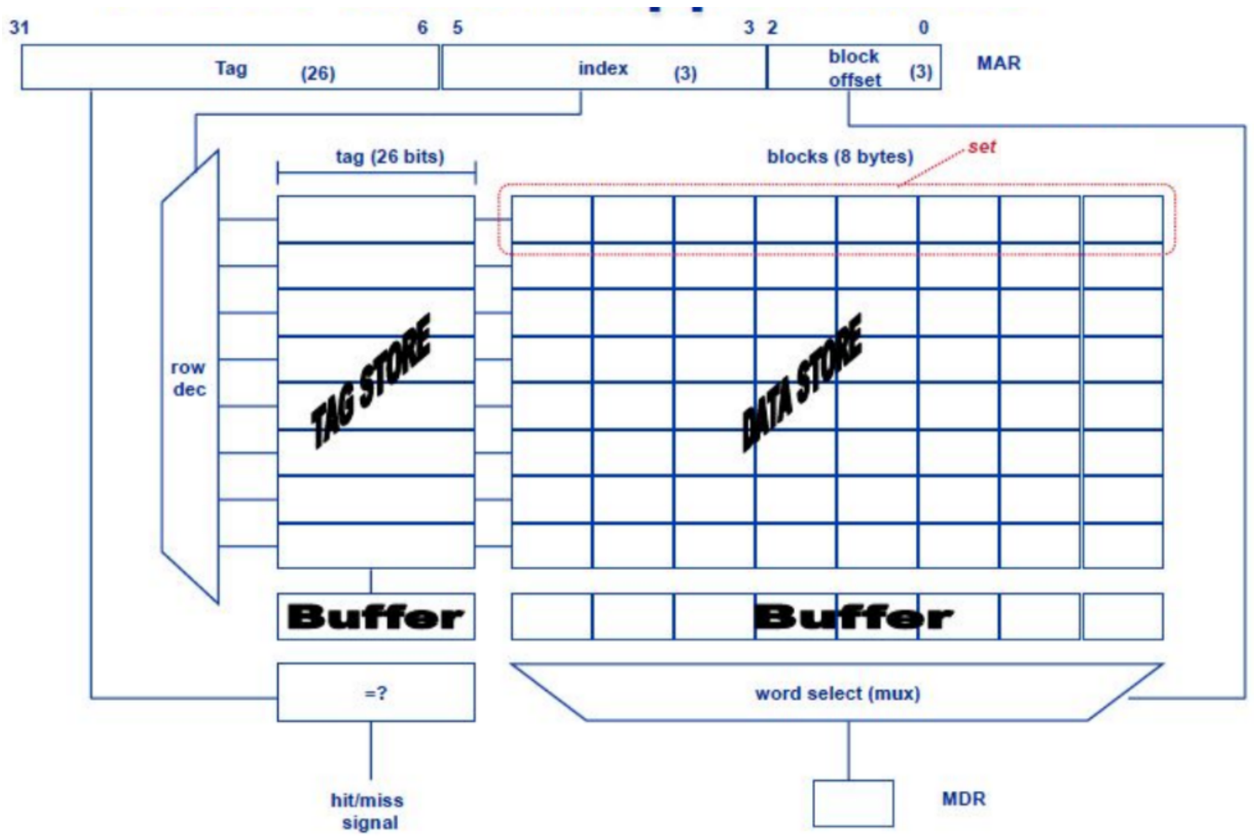
Aidan Jan

November 14, 2024

Cache Block

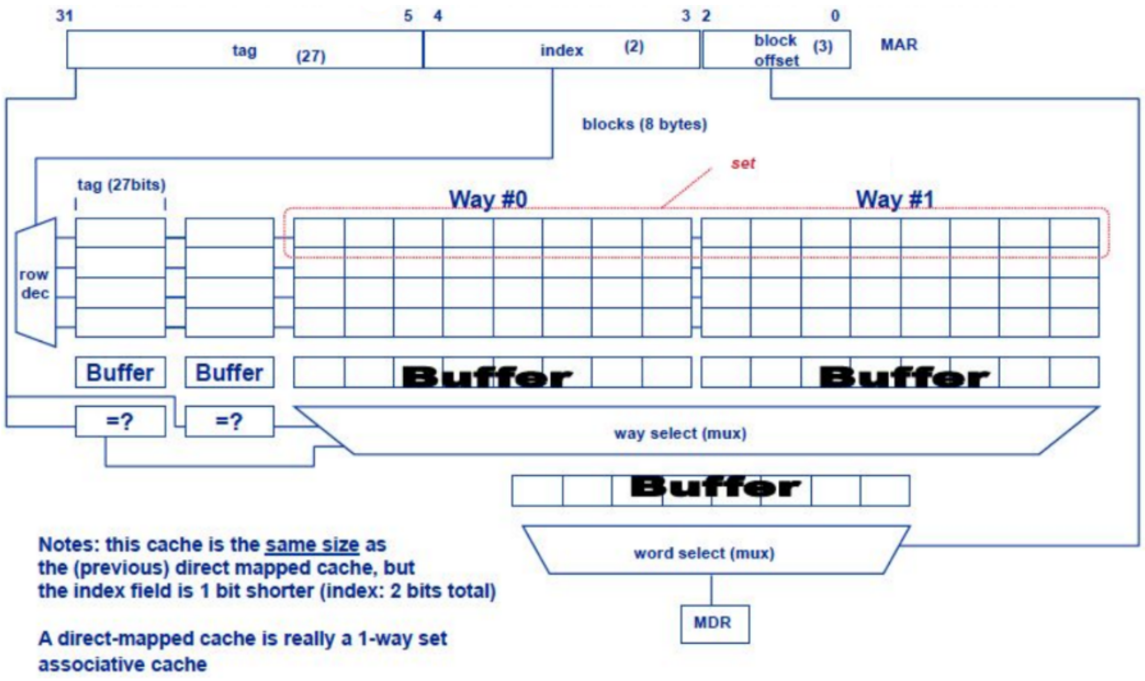
- Instead of storing 1 byte per row, we can store a block with multiple bytes.
  - Every time we need to load something to the cache, we load it at block level. (Spatial Locality)
  - We still send things to the CPU at byte level
    - How to do that? $\rightarrow$ We need *block offset* to decide which byte within the block should be selected!
  - How big a block should be? It depends! Typically somewhere between 8B-64B.
- Therefore, **Cache Address = {tag, index, block offset}**

A 64B Direct Mapped Cache



- Address is used to select index, of tag, which selects "rows", and block offset selects the word "columns".

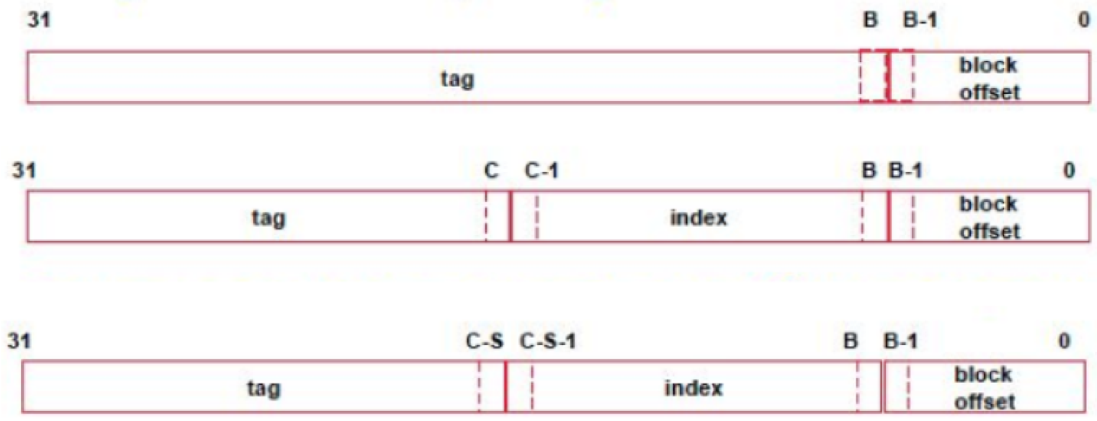
A 64B 2-way Set Associative Cache


31
5 4
3 2
0
tag (27)
index (2)
block offset (3)
MAR
blocks (8 bytes)
set
tag (27bits)
Way #0
Way #1
row dec
Buffer
Buffer
Buffer
Buffer
=?
=?
way select (mux)
Buffer
word select (mux)
MDR
Notes: this cache is the same size as the (previous) direct mapped cache, but the index field is 1 bit shorter (index: 2 bits total)
A direct-mapped cache is really a 1-way set associative cache

- Compared to the direct mapped cache, the tag is two bits shorter to make way for an index.
- Here, the tag is used to select which "Way" is used, and the two bits of the index selects the row. The block offset then gets the column.

CBS

- C = log(bytes per cache)
- B = log(bytes per block)
- S = log(blocks per set)



How Big A Block Should Be?

- Bringing more data is nice because you have spatial locality
- However, it is not always the best idea because it increases overhead

- You are essentially making a trade off between miss rate and miss penalty.

Reducing Miss Rate

- Miss rate can be reduced by making blocks bigger, but that comes at the trade-off of miss penalty.
- What if we increase associativity? (e.g., add more ways)
  - More ways leads to higher hit time. (As log(cache size) increases, miss rate drops, but it drops following an exponential decay function. "diminishing returns")
  - An 8-way set associative cache is as good as fully-associative. After that, the limit is capacity miss.
- We can also increase cache size. But this leads to slower hit time. Making cache larger also has diminishing returns!
- Prefetching: Idea: if we can guess the access pattern we can bring data before it is needed!

Prefetching - Four Questions!

- **What** addresses to prefetch (i.e., address prediction algorithm)
- **When** to initiate a prefetch request (early, late, on time)
- **Where** to place the prefetched data (different layers of caches, separate buffer)
- **How** does the prefetcher operate and who operates it (software, hardware, hybrid)

Prefetchers look at the history of addresses accessed to predict the next address access. Similar to how a branch predictor looks at the history of branches, the prefetcher looks at the history of addresses.

- This reduces compulsory misses and therefore miss rate
- However, this leads to cache pollution
  - Need to monitor prefetching accuracy to change its *aggressiveness*
  - Other than this, no other negative impacts! No correctness issues!

Software vs. Hardware Prefetch

- Software prefetching
  - ISA provides prefetch instructions
  - Programmer or compiler inserts prefetch instructions (effort)
  - Usually works well only for "regular access patterns"
- Hardware prefetching
  - Hardware monitors processor accesses
  - Memorizes or finds patterns/strides
  - Generates prefetch addresses automatically

Example: Hardware Prefetcher

Next line prefetcher:

- Always prefetch next N cache lines after a demand access
- Pros:
  - Simple to implement
  - No need for sophisticated pattern detection
  - Works well for sequential/streaming access patterns (instrctions?)
- Cons:
  - Can waste bandwidth with irregular patterns
  - Low prefetch accuracy if access stride = 2 or when the program is traversing memory from higher to lower addresses.
- Better options? Stride prefetcher, stream buffers, etc.

Victim Cache

- Idea: for heavily conflicting addresses, a few "extra" temporary sets could remove conflicts!
  - Use a very small buffer (called victim cache) to save the recently discarded blocks. Search through them as well.
  - Reduce conflict misses
    - Research shows a 4-entry victim cache can remote up to 90% of conflicts.
  - Extra overhead
  - More complex design.

Compiler and Software

- Reorder accesses/arrays to increase locality.
- Combine loops with similar behavior
- Use "tiling" to access arrays region by region instead of whole
  - If column-major, `x[i+1, j]` follows `x[i, j]` in memory.
  - Meanwhile, `x[i, j + 1]` is far away from `x[i, j]`.
  - Poor code:

    ```
    for i = 1:
        for j = 1:
            sum = sum + x[i, j]
    ```

  - Better code:

    ```
    for j = 1:
        for i = 1:
            sum = sum + x[i, j]
    ```

- Use compiler profiling to improve prefetching

Reducing Miss Rate

- Replacement policy
  - LRU vs. PLRU vs. Random
  - Storage vs. Accuracy tradeoff!

Reducing Miss Penalty

- Write buffer: use a load store queue
- Pros:
  - No wait for stores needed.
  - Lower miss penalty for loads.
- Cons:
  - More overhead.

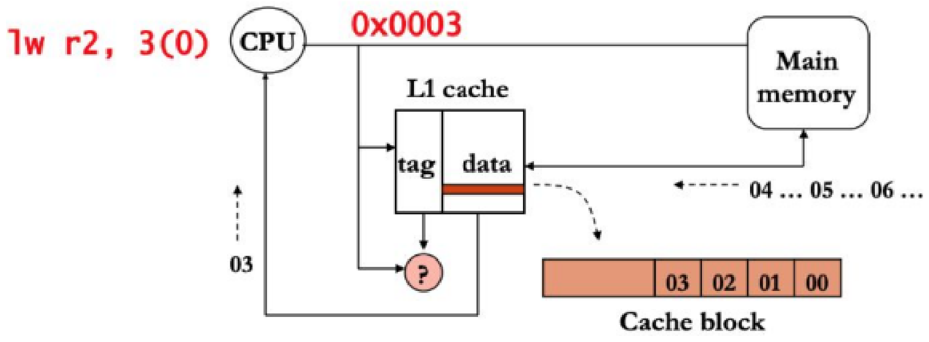
What happens on a store?

1. Data exists in the cache?
   - Should we update memory AND cache on every write?
     - Write through strategy
   - Update the memory only when the line is evicted.
     - Write back strategy
   - Tradeoff: Less writes vs. Storage overhead vs. Memory status.
2. Data does not exist.
   - Should we bring it to the cache? -write allocate
   - We probably don't need it anymore, so don't bring it. -write no allocate

- *Write back* often compined with *write-allocate.*
- *Write-through* often combined with *write-no allocate.*
- How to pick?
  - It depends!
    - Could be different for each level!
    - Can be optimized using simulation and architectural search!

Reducing Miss Penalty via Early Restart

- Instead of waiting for all bytes (in a block) to arrive, forward data to CPU as soon as the requested byte(s) arrives.

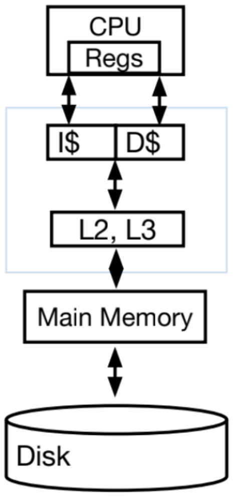


Early Restart *with critical word first*:

- Instead of waiting for all bytes (in a block) to arrive, forward data to CPU as soon as the requested byte(s) arrives.
- To further optimize this, first read the requested byte!

Multi-level Cache

- We can also reduce miss penalty by adding more levels of cache:
  - Adding L2, L3, etc.
- Higher level cache means bigger and slower. However, it is still both smaller and faster than the main memory.



Cache Performance

- Average time to access the cache:

$$AAT = HitTime + MissRate \times MissPenalty$$

- $HitTime$: Time it takes to access the (L1) cache.
- $MissRate$: The average frequency of misses (in L1).
- $MissPenalty$: The time required to access the main memory.

- What if there are multiple levels?
  - The miss penalty is the average time required to grab the data from either the other caches or main memory.

Inclusive vs. Exclusive Cache

- Inclusive: L_i is a subset of L_{i+1}
  - (pro) Easier to find data
  - (con) Wasted capacity
- Exclusive: Data is **only** in one of the levels.
  - (con) Difficult to find data
  - (pro) Efficient capacity

Modern Designs

- Split vs Unified "Caches"
  - L1 I/D caches commonly split and asymmetrical
    * Double bandwidth and no-cross pollution on disjoint I and D footprints
    * i-cache is smaller, simpler with more spatial locality. Usually a prefetcher and/or trace cache is connected to i-cache.
  - L2 and L3 are unified for simplicity
- "Havard" design referred to a microarchitecture with **separate** instruction and data memory.
- "Princeton" design referred to von Neumann's **unified** instruction and data memory. This is the most common design.

Sub-Blocking

- Higher block size improves miss rate but also increases miss penalty!
- Idea: keep a large block size, but divide it into smaller "subblocks". Bring only a subset of subblocks on a miss.
  - (pro) lower miss rate
  - (pro) lower miss penalty
  - (con) need separate storage for valid bits for each subblock
  - (con) more complex circuitry.

Reducing Hit Time via reducing associativity and size

- DM $<$ FA (direct mapping $<$ fully associative)
  - Use SA to balance between the two
- Use smaller cache in lower levels (L1, L2, ...)

Reducing Hit Time via Parallel lookup

- Access tag and data in parallel.
- Access each way in parallel.

Reducing Hit Time via Speculative load

- Instead of waiting for a store (potentially conflicting), issue the load speculatively.
  - Once store is resolved, check whether there was a conflict or not. Recover if there was.

Cache Summary

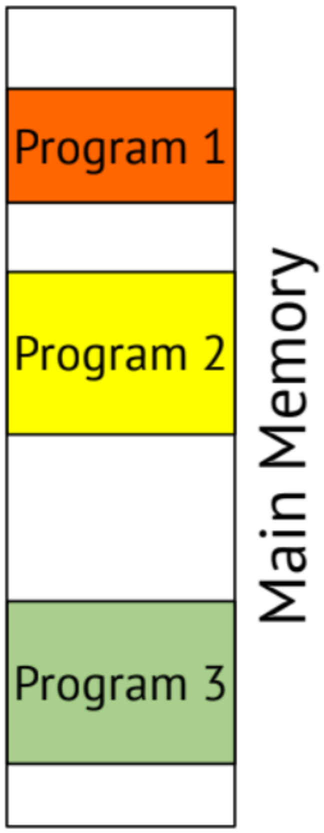
- Miss Rate
  - Increase block size
  - Increase associativity
  - Increase cache size
  - Prefetching
  - Victim cache
  - Compiler
  - Replacement Policy
- Miss Penalty
  - Write buffer
  - Early restart with critical block first
  - Adding more levels
  - Sub-blocking
- Hit Time
  - Set associative cache
  - Add more levels
  - Parallel lookup
  - Speculative loads

Main Memory

Current Technology

- We see combination of DRAM and DISK as the main memory (usually call teh DRAM part Main Memory and the HardDrive part DISK).
- Access to the Main Memory (combination of disk and DRAM) is *always a hit*.
  - As we will describe alter, data could be in disk and/or DRAM, and we need to handle that.
  - Huge difference between latency of DRAM to DISK (and of course to cache).

Where to store different programs?



The fundamental requirement is **Isolation**.

- Each program typically has *different* regions:
  - code
  - data
  - heap
  - stack
  - ...
- Portion of the memory code can be **shared** among multiple programs
- Portion of the memory can be **reserved** for the OS and/or other privileged activities.

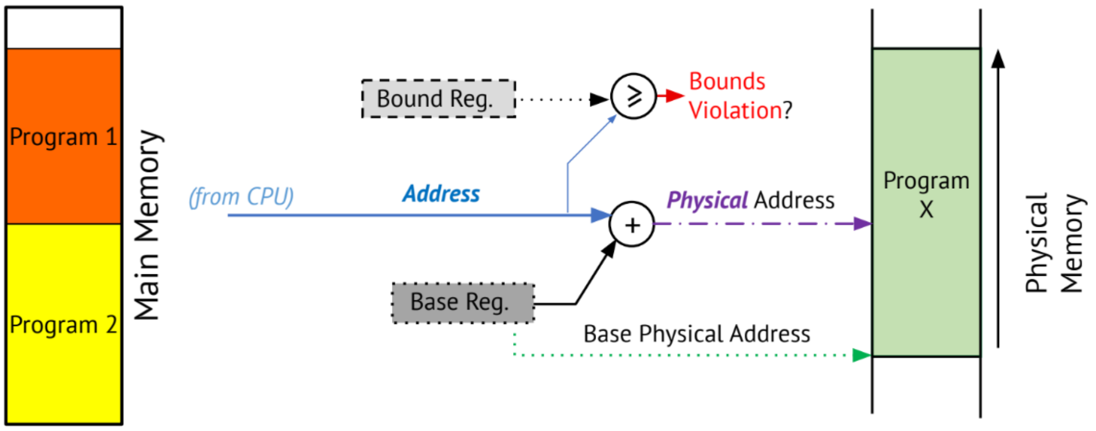
Memory Management

- Early (and simple) machines only run one program with unrestricted access to ALL memory.
- Modern systems run several programs, and memory should be shared between multiple programs.
- Memory management is controlled by OS or system software, or hardware.

How to share the memory between multiple programs?

Option 1: Partition the memory (statically)!

- **Protection:** Independent program should not be able to affect each other inadvertently. (check bounds!)
- **Location-Independence:** Program might be moved anywhere in the memory. (use a base pointer!)
- To do these, we use a *virtual* address, which is the base pointer plus whatever pointer in the program.
  - When switching programs, OS updates base and bound registers.
  - Example: $PA = Base\_Reg + Logical\_Adr$



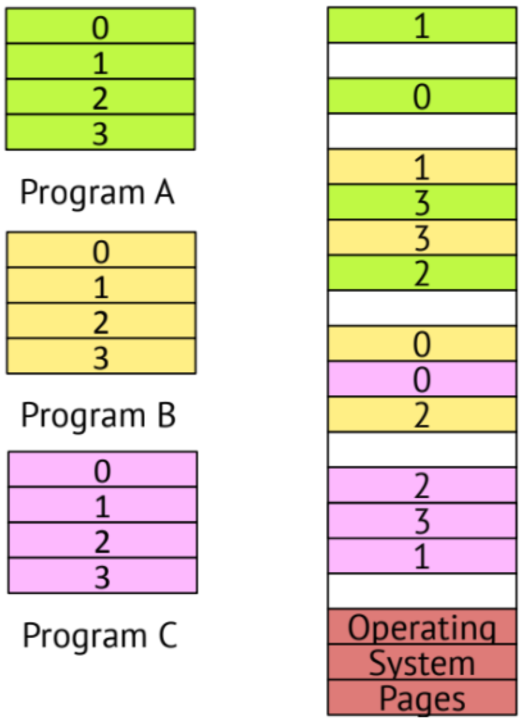
How to share code/libraries?

- Multiple programs can access (read-only) shared memory spaces.
  - Standard libraries (e..g, printf)
  - Drivers
  - ...
- To do this, the library will be stored in its own block of main memory, and each program has a pointer to it!
- What about the OS? (e.g., system commands) The OS is a program as well! Therefore, it can be shared.

What happens if we want to add a program or change the space the program takes up?

Option 2: Partition the memory (dynamically)!

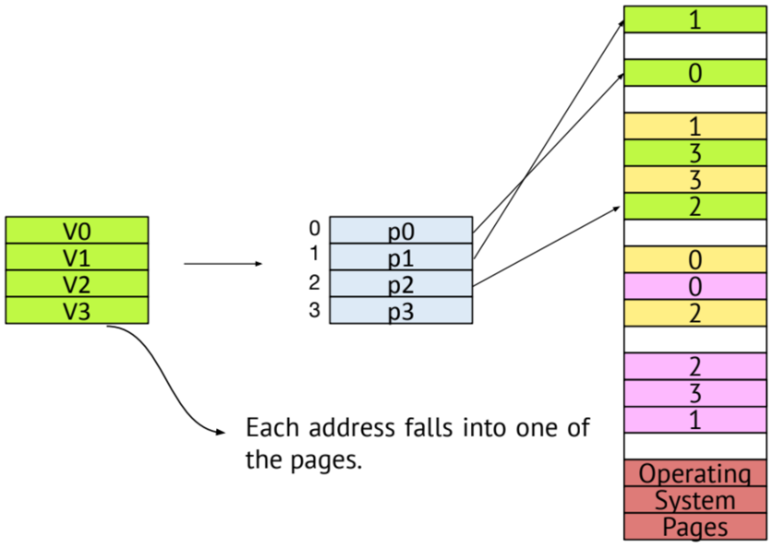
- Divide the memory into fixed-size blocks (called page).
- Assign pages to each program as needed.
- Pages can be *scattered* in the memory
- Typically 4KB per page.



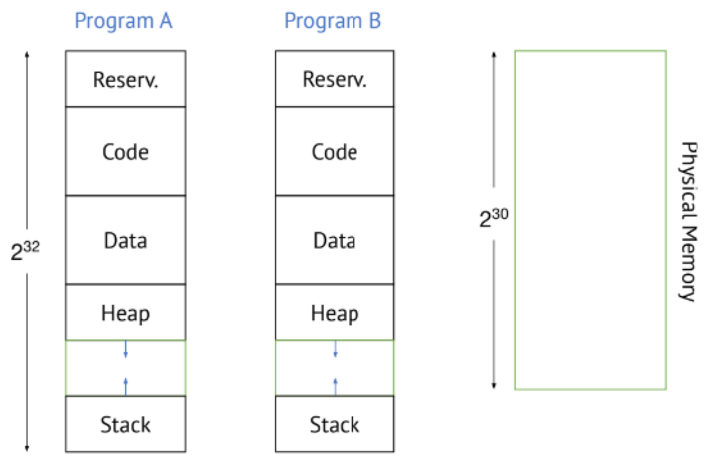
- Now we have another problem: each page can be *anywhere*!
- To solve this, we use a *page table* which tracks mapping!

Virtual Addressing

- Since we are using page tables for translations, we no longer need to use real physical memory addresses in each program.
  - Each program can start from (virtual) address 0x00.
  - Each program sees a *large, private*, and *uniform* memory. (this is just an illusion though!)

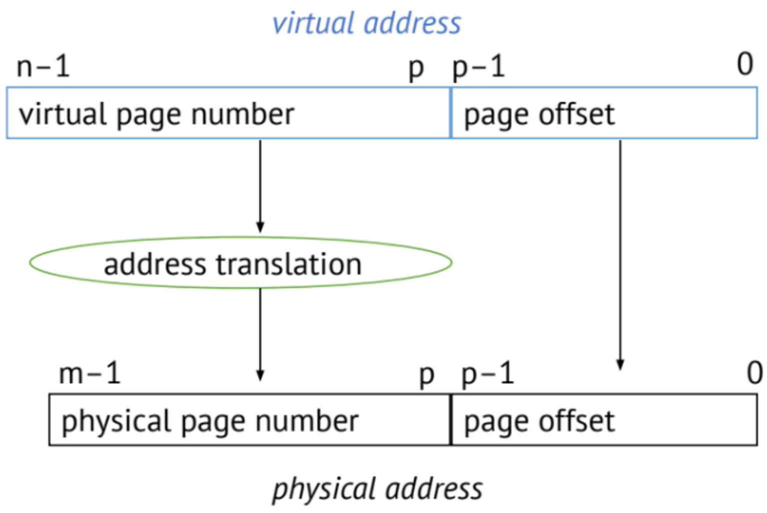


  - Each program sees a large, private, and unified memory.



Virtual to Physical Address Translation

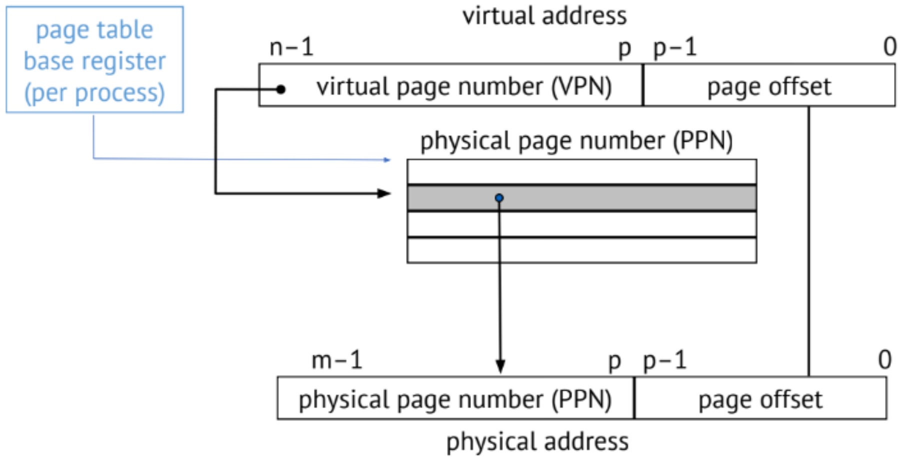
- 2^n: virtual address size
- 2^m: physical address size
- 2^p: page size



Virtual and Physical Memory

- System:
  - Virtual memory size: 4GB = 2^{32} bytes
  - Physical memory size: 256MB = 2^{28} bytes
  - Page size: 4KB = 2^{12} bytes
- Organization:
  - Virtual address: 32 bits
  - Physical address: 28 bits
  - Page offset: 12 bits
  - # Virtual pages = $2^{32}/2^{12} = 2^{20}$ (VPN = 20 bits)
  - # Physical pages = $2^{28}/2^{12} = 2^{16}$ (PPN = 16 bits)

Page Table


page table base register (per process)
virtual address
n−1
p
p−1
0
virtual page number (VPN)
page offset
physical page number (PPN)
m−1
p
p−1
0
physical page number (PPN)
page offset
physical address

- Note that the Virtual Page Number (VPN) is the index to the table.

What else to store in each page table entry (PTE)?

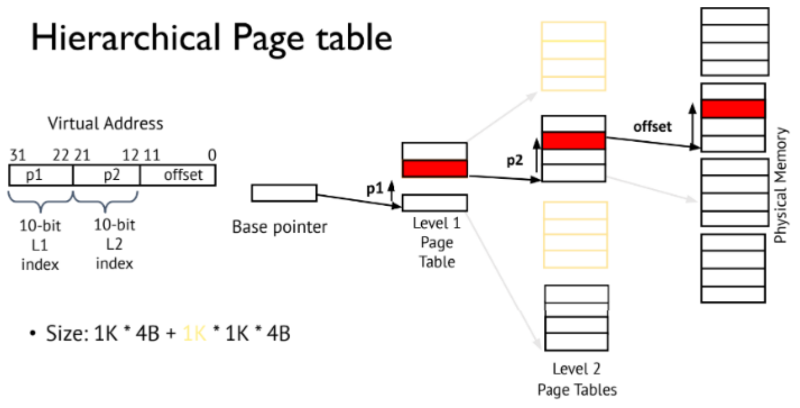
- Protections and flags
  - Whether we can write to this page or not (read-only)?
  - Who is the owner? Is it shared?
  - Can we execute it?
- Page tables must also be stored in the memory.
- Each memory access becomes two accesses, one for accessing the page table (i.e., adr + VPN + base). The other for accessing the translated address (i.e., adr = PPN + page offset).

How large is a page table?

- # Virtual pages = $2^{32}/2^{12} = 2^{20}$ (VPN = 20 bits)
- Physical address: 28
- Each entry in page table (PTE) = 28 - offset + flags $\approx$ 32 bites = 4 bytes.
- $\rightarrow 2^{30} \times 32 =$ 4MB for each application!
- Number of bits double in a 64-bit system!

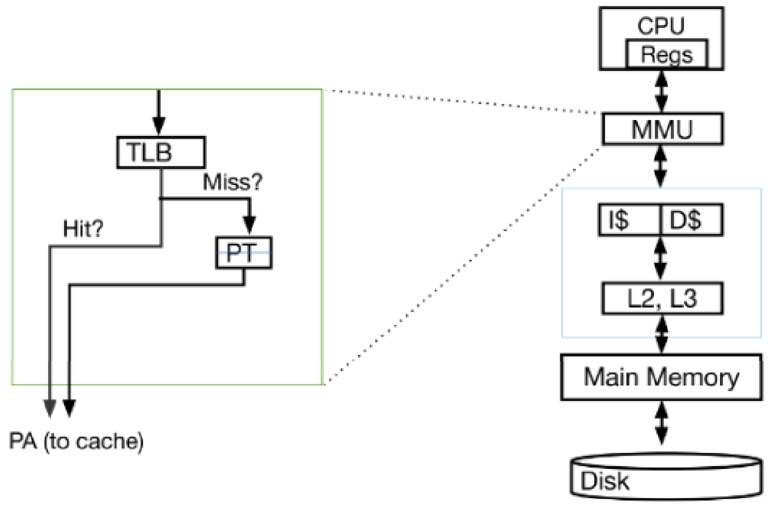
Hierarchical Page Table

- Store the page table as a page table

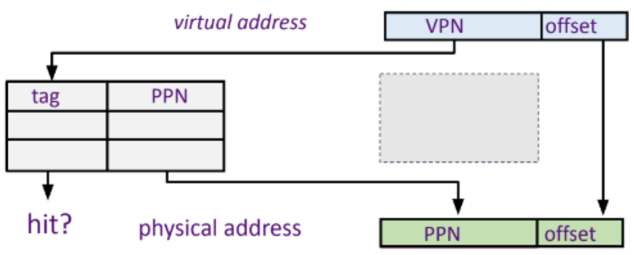


- Now we divide one translation into multiple.
- The process of going from one level of page table to the next is called "walking".
- The problem is that now the page table is huge, and memory is limited.
- Solution: Remove some second level pages! Empty page tables can be immediately removed, and reallocated in memory if it is needed.
- Hierarchical page tables can leverage the existing sparsity of virtual addressses and make the page table storage compact!
- New problem: Each memory access (e.g., L1 cache) needs multiple memory accesses!
  - To address this, **cache** the most recent translations!

Translation Lookaside Buffer (TLB)



- The hit time is $t_{TLB} + t_{L1}$.
- Cache translations in TLB
  - TLB hit $\Rightarrow$ Single-cycle translation
  - TLB miss $\Rightarrow$ Page-Table Walk to refill



TLB Design

- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages.
  - Larger TLBs (256-512 entries) are 4 or 8 way set-associative.
  - Larger systems sometimes have multi-level (L1 and L2) TLBs.
  - Random or FIFO replacement policy
  - **TLB Reach:**
    - Size of largest virtual address space that can be simultaneously mapped by TLB
    - Example: 64 TLB entries, 4KB pages, one page per entry. TLB Reach = 256 KB
  - TLB miss causes an exception and results in a page table walk.
  - OS typically is responsible to handle TLB miss (software handling).
  - Alternatively, memory management unit (MMU) can handle TLB miss.

How to utilize disk?

- Disk is our secondary storage unit with much bigger size, but much larger access time.
- How do we use this efficiently?

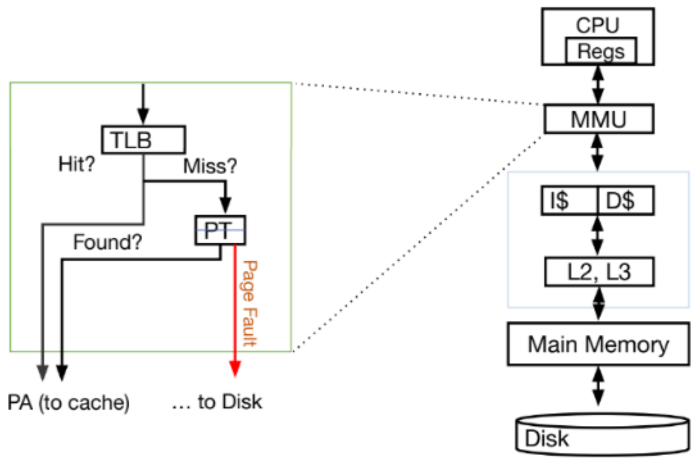
Demand Paging

- Use main memory and "swap" pages in the disk as automatically managed memory hierarchy levels.
- $\rightarrow$ Analogous to cache vs. main memory
- M (DRAM + Disk capacity) bytes of storage.
  - keep most frequently used C bytes in DRAM ($C << M$)
  - keep the rest in disk.
  - If the page is not in DRAM, we call it a *page fault*.
  - Bring a page (from disk to main memory) when "demanded".
  - In the page table, if the valid bit is 0, then the page is not in memory
    - We call this a **page fault**.

Demand Paging Design

- Same basic issues as before (in cache)
  - When to bring a page into DRAM?
  - Which page to evict (we call it "swap") from DRAM to disk to free-up DRAM for new pages?
  - Page size?
  - ...
- OS handles everything (easier, fast enough)
  - Pseudo-LRU replacement policy

Memory Hierarchy Summary

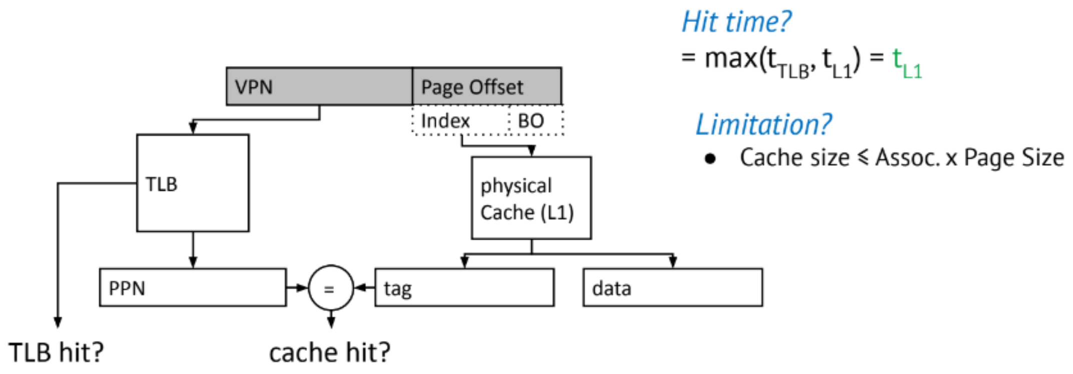


- Page tables
  - Why we need them and how to translate?
  - How to store them?
  - How to reduce access time?

How to decrease hit time?

- Can we access L1 before TLB?
  - Address in L1 should be stored with VA (i.e., no translation!)
  - Problem? Aliasing. Therefore, L1 **cannot** cannot be accessed *before* TLB.
    - * However, we can access them *in parallel*.

Virtually-Indexed Physically-Tagged Cache



With and Without MMU

- Most embedded processors and DSPs provide *physical addressing* only!
  - Can't afford area/speed/power budget for virtual memory support
  - Often there is no secondary storage to swap to!