# Memory

Nimish Mishra
@manwe sulimo

# Agenda

**Why memory hierarchy?**

An intuition...

**Registers & Caches**

Another intuition...

**Virtual memory**

A gentle introduction...

**Appendix**

Some extra things:

-Userspace memory profiling
-Intuition on microarchitectural attacks

**Questions**

-Technical?
-Non-technical?

- Recap: VA
- A new system design
- Memory hierarchy
- Processor and memory hierarchy communication

# Why memory hierarchy?

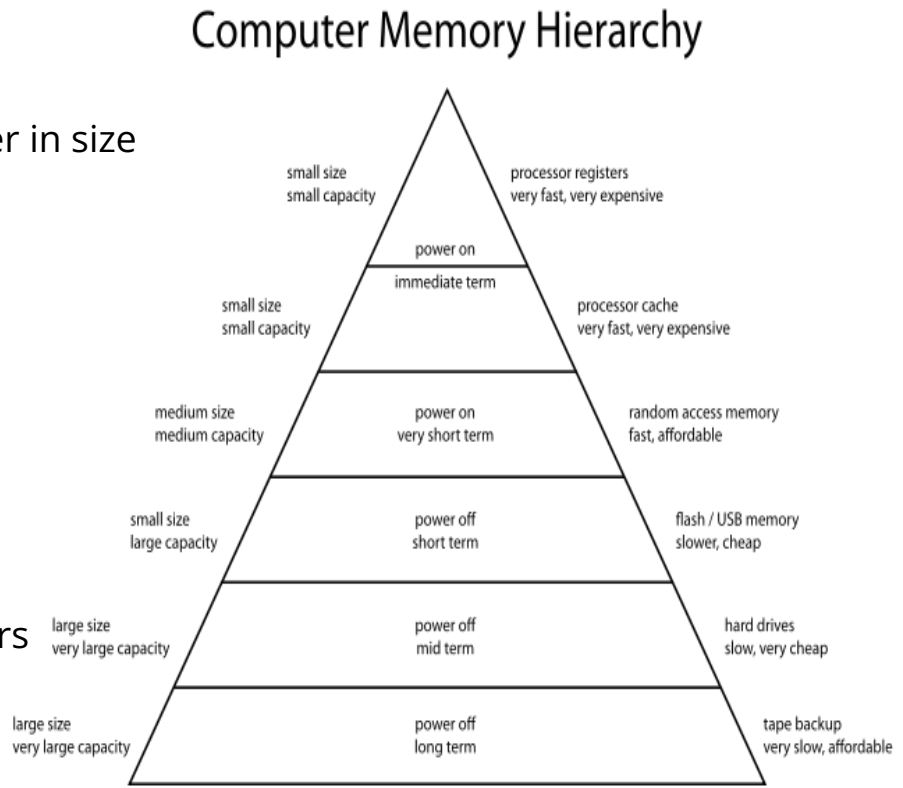
# A new system design

- A 256 GB pen drive costs around Rs. 1300. Means a 1 TB hard disk could be created in Rs 7200. Add Rs. 10000 for a processor. Can we create a new 1 TB system at Rs 20000 (there's profit too xD)?
    - So what's the catch? ***Speed***

- Another system design: We know our RAMs are pretty fast. But most individual systems have on average 16 MBs of RAM. Can we design a system with 1 TB RAM?
    - Again what's the catch? **Cost**

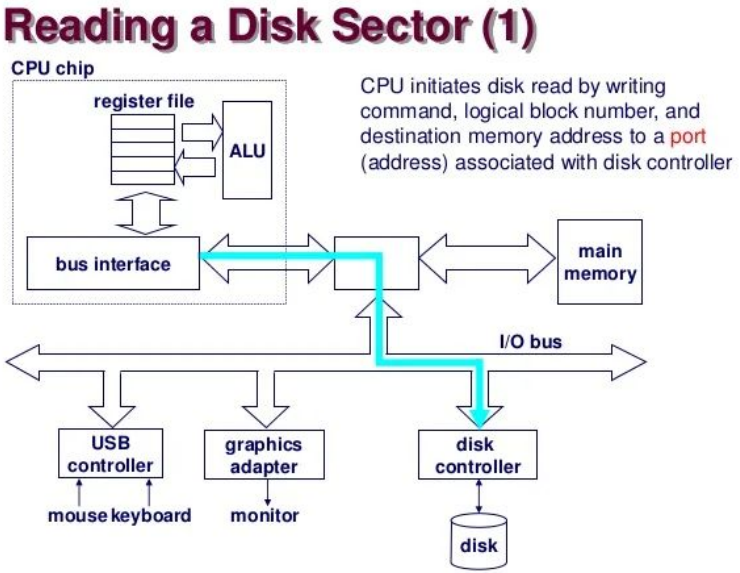
# Memory hierarchy

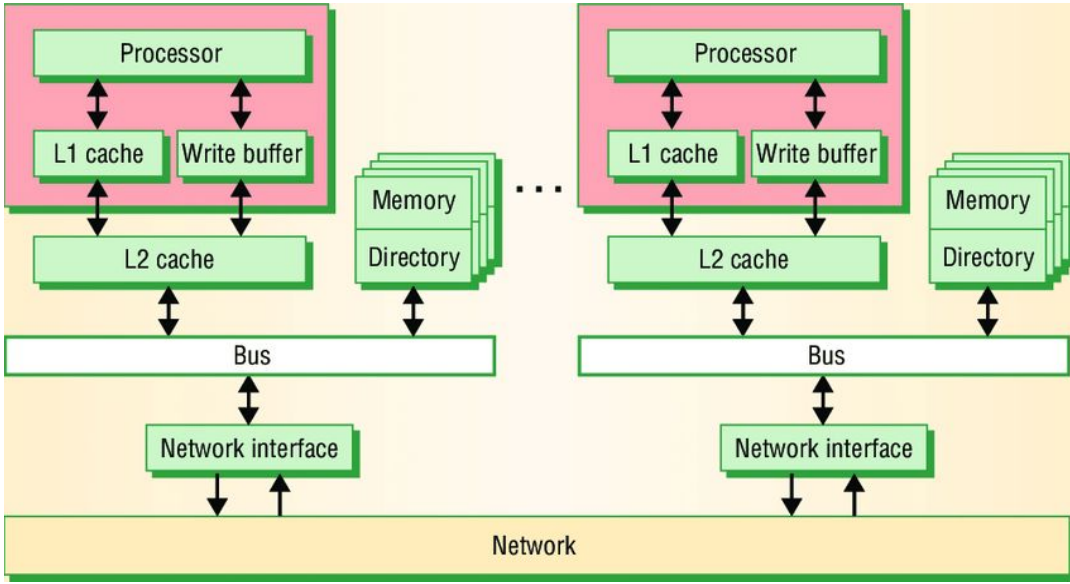
- There's a strict tradeoff between speeds and cost
- Faster memory is *costlier*.
- Solution: a hierarchy of memories
  - Faster memory closer to processor but smaller in size
  - Slower memory further away but large in size
- **Principles of locality**
  - Power entire designs of memory systems
  - **Temporal locality** (recency in *time*)
  - **Spatial locality** (recency in *space*)
- What do we keep in higher memories?
  - Recently accessed data
  - Data that is very frequently accessed
  - Critical software like kernel's interrupt handlers



# Processor and memory hierarchy communication

- Through **buses**



# Registers and caches

- x86 register file
    - Caches
- How do cache mappings happen?
    - Type of misses
- A game of tradeoffs
- Cache coherency

# x86 register file

- x86 GPR register file. GPR = general purpose registers. **Other registers?**

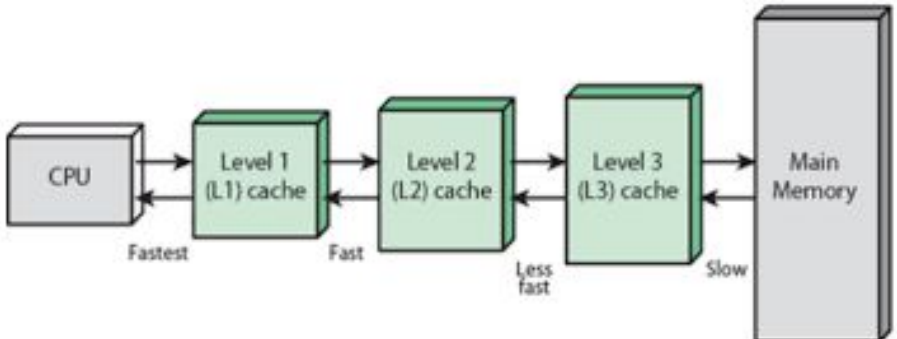
| Register | Accumulator | Counter | Data | Base | Stack Pointer | Stack Base Pointer | Source | Destination |
|---|---|---|---|---|---|---|---|---|
| 64-bit | RAX | RCX | RDX | RBX | RSP | RBP | RSI | RDI |
| 32-bit | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |
| 16-bit | AX | CX | DX | BX | SP | BP | SI | DI |
| 8-bit | AH, AL | CH, CL | DH, DL | BH, BL | SPL | BPL | SIL | DIL |

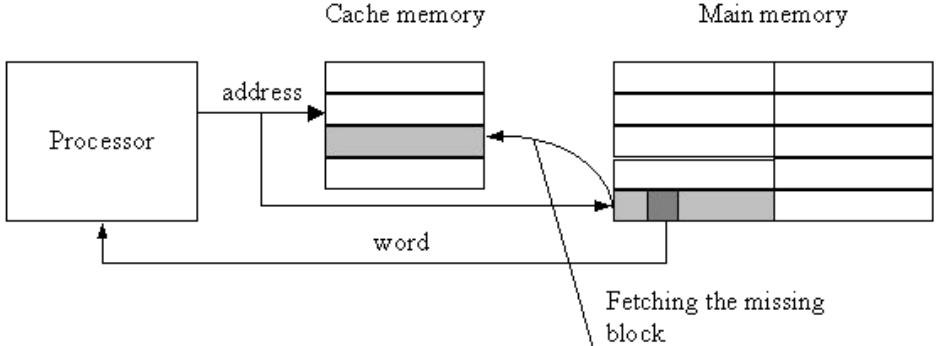
- Accumulator: for arithmetic
- Counter: used in places like loops
- Data: used in both arithmetic and I/O operations
- Base: the base pointer (pointing to the base of the process image)
- Stack pointer: points to the current memory location in stack
- Stack base pointer: points to the base of the stack
- Source and destination: used for data streams (I/O streams etc)

**Question:** In case of a context switch, why is it necessary to *save* these?

# Caches

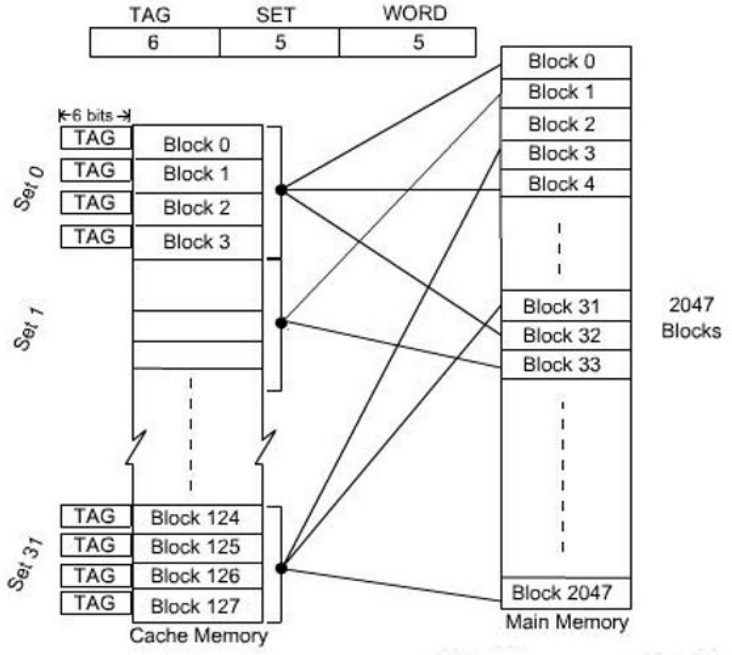
- *Caches* the most recently used data *close* to the processor
- Usually multi-leveled with the system bus acting as the interface between all
- The processor sends out a **virtual address**:
  - **Hit time**: If data exists in cache, it is returned to the processor
  - **Miss time:** If data doesn't exist, fetch it from higher memory and return to the processor
- How do principles of locality help?
  - **Temporal locality**: Recently accessed data is highly likely to be accessed again soon. So keep it in cache
  - **Spatial locality:** Data *close* to recently accessed data is likely to be accessed again soon, so bring such data into cache (**look at dark grey and light grey blocks in the right figure**)

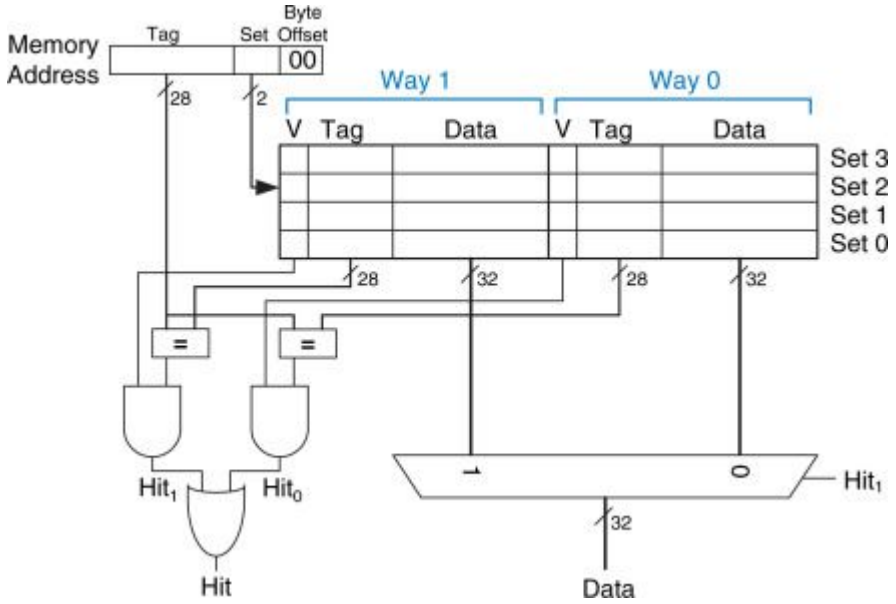


# How do cache mappings happen?

- Another tradeoff? *Directly mapped* cache or *completely associative* cache
  - Directly mapped: One virtual address maps to **exactly one** cache slot. *Fast but has faults*
  - Fully associative: One virtual address maps to **all** cache slots. *Slow but has less faults*
- Solution: a set associative cache (**the right image down here**)



# Types of misses

- Compulsory misses: At the beginning of a process, the cache is not populated. Hence, all memory references result in a **compulsory** cache miss

- Conflict miss: Assume a process that accesses data **x** and **y** alternatively, where both map to the same set in a *directly mapped cache.* Means first **x** will be brought into cache, then **y** is brought (**x** evicted), and this repeats.

- Capacity miss: Try running 15 tabs on Google Chrome on a 4 GB RAM :)
    - basically cache is smaller than the space needed to run a program

# A game of tradeoffs

- Assuming total capacity of cache to be the same
  - Increasing block size
    - **Decreases** number of sets
      - **decreases** compulsory misses
      - **increases** conflict misses
    - **Increases** throughput (and application of spatial locality)
      - requires **higher bus size** and more time to bring in data.
  - Increasing associativity
    - **Decreases** number of sets
      - more blocks per **set**
      - **decreases** conflict misses
    - **Increases** hit time
      - since we need more comparators and one multiplexor
- Increasing total capacity of cache?
  - decide what to sacrifice and what to gain

# Cache coherency

- So far we considered **read** operations only
- Problem with **writes**: processor writes to cache, so how long do you wait before reflecting the updated value in memory
- **Where coherency problem arises?**
  - Assume multicore setting, running two threads of the same process sharing some data.
  - If one thread writes, ideally you want to **write** from cache to memory before second access
- Two kinds of cache writes:
  - **Write through:** write to memory as soon as the block gets written
    - Low throughput, since you do an expensive memory operation every time
  - **Write back:** write to memory **when a *dirty* cache block has to be evicted**
    - Maintain an extra **bit** in cache line indicating whether a block is **dirty** (or written over)
    - When the time comes to evict a *dirty* block, first write it to memory and then evict
    - May face problems of cache coherency: since you **wait** for a while before writing, in the meantime someone else may access the outdated data
- *Cache coherency across cores*: *spy on cache accesses to know when to write back*

# Virtual memory

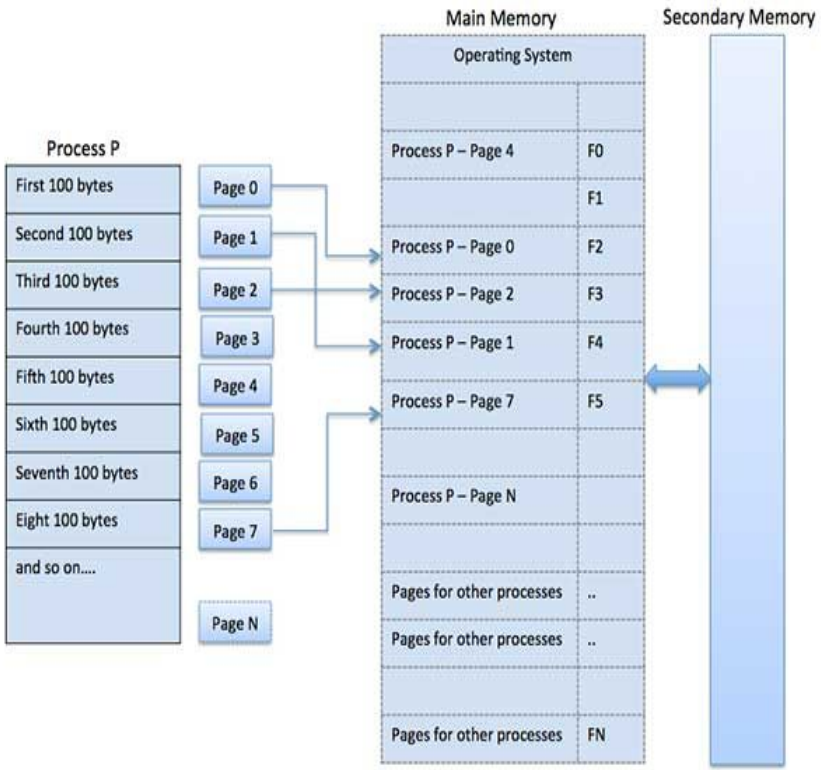
- Why we need virtual memory?
- An intuition to memory management unit

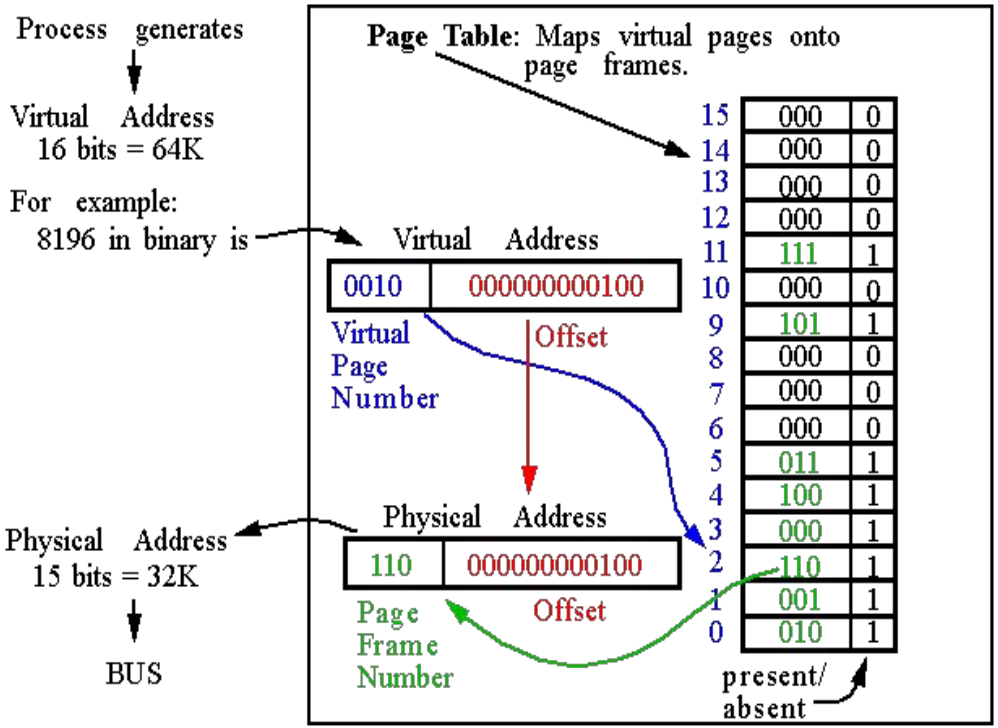
# Why we need virtual memory?

- Several reasons:
  - **Protection**: if you use virtual memory space, you can protect one userspace program from being operated upon by another userspace program (**recall the V bit in cache design?**)
  - **Illusion:** A virtual memory can give an illusion of more memory than the actual memory present
    - How? A good intuition is that each cache set maps to several main memory blocks (if you look back to our slide on cache design, 2048 blocks for 128 cache sets)
  - **Address space randomization:** since we have *mappings*, the program can always access a certain data **xyz** at address **0x00000000deadbeef**, and the **mapping can change**, thereby placing this data anywhere in physical memory
- So what's the catch?
  - **Additional hardware:** you need hardware to *convert* virtual memory addresses to actual *physical addresses*
  - **Burden on the operating system:** the OS shares the burden of creating and managing page tables for each process. **Page tables** are *mappings* from the virtual address to the physical addresses

# An intuition to memory management unit

- A **page** is usually in KB. **Offset** means which **byte *inside*** the page



- Userspace memory profiling
- Intuition of cache based recovery of AES key

# Appendix

# Userspace memory profiling

- Designers of critical software (like cryptographers) don't usually take into account the difference between theoretical and actual execution environments
- Can we exploit?
  - Probably
  - But before that, we need to profile our memory
    - **Verify:** can timing measurements help us differentiate **cache** accesses?
    - **Verify:** can we understand what bytes are being used by a certain program?
    - A sample repository for the same: https://github.com/IAIK/cache_template_attacks
- It's all fun and games until...
  - The victim code is critical software like AES (advanced encryption standard)
  - Attack vector:
    - Profile the cache and get the difference between hit and miss times
    - Fill the cache with your bogus data
    - Run AES. It uses T-tables which which will evict your bogus data from cache
    - After AES ends, know which portion of your data is evicted. Can help in AES key recovery

Link to everything:
https://github.com/NimishMishra/exploit-dev/tree/master/notes/sessions/mmu

# Questions?