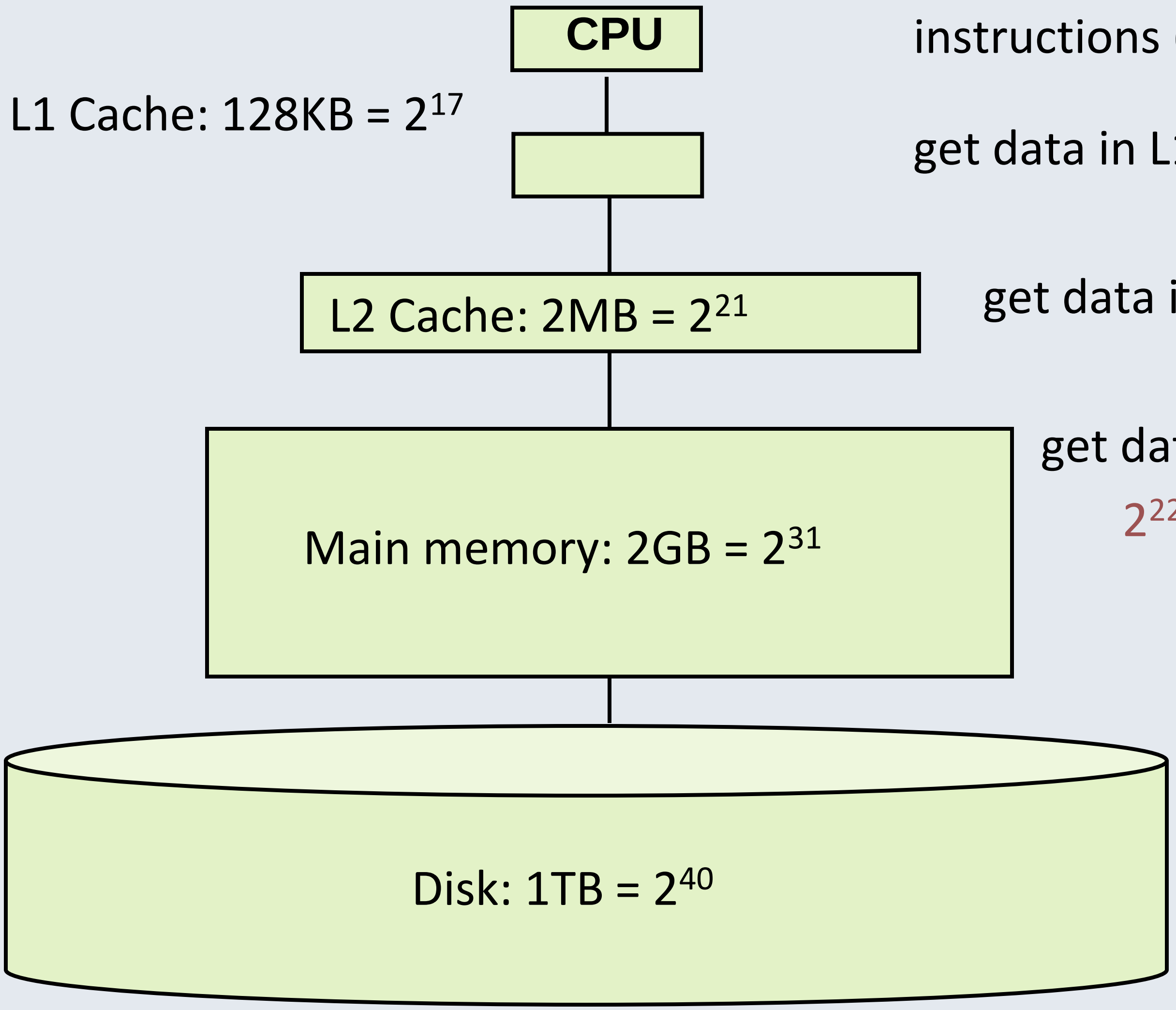
CSE 332: Data Abstractions Memory Hierarchy

Richard Anderson

Spring 2016

A typical hierarchy

Every desktop/laptop/server is different but here is a plausible configuration these days



instructions (e.g., addition): 2^{30}/sec

get data in L1: 2^{29}/sec = 2 insns

get data in L2: 2^{25}/sec = 30 insns

get data in main memory:

2^{22}/sec = 250 insns

get data from “new place” on disk:

2^{7}/sec =8,000,000 insns

“streamed”: 2^{18}/sec

Morals

It is much faster to do:	Than:
5 million arithmetic ops	1 disk access
2500 L2 cache accesses	1 disk access
400 main memory accesses	1 disk access

Why are computers built this way?

- Physical realities (speed of light, closeness to CPU)
- Cost (price per byte of different technologies)
- Disks get much bigger not much faster
  - Spinning at 7200 RPM accounts for much of the slowness and unlikely to spin faster in the future
- Speedup at higher levels makes lower levels *relatively slower*

Usually, it doesn't matter . . .

The hardware automatically moves data into the caches from main memory for you

- Replacing items already there
- So algorithms much faster if "data fits in cache" (often does)

Disk accesses are done by software (e.g., ask operating system to open a file or database to access some data)

So most code "just runs" but sometimes it's worth designing algorithms / data structures with knowledge of memory hierarchy

- And when you do, you often need to know one more thing…

Block/line size

- Moving data up the memory hierarchy is slow because of *latency* (think distance-to-travel)
  - May as well send more than just the one int/reference asked for (think “giving friends a car ride doesn’t slow you down”)
  - Sends nearby memory because:
    - It is easy
    - Likely to be used soon (think fields/arrays)

Principle of *Locality*

- Amount of data moved from disk into memory called the “block” size or the “page” size
  - Not under program control
- Amount of data moved from memory into cache called the “line” size
  - Not under program control

Connection to data structures

- An array benefits more than a linked list from block moves
  - Language (e.g., Java) implementation can put the list nodes anywhere, whereas array is typically contiguous memory
- Suppose you have a queue to process with 2^{23} items of 2^{7} bytes each on disk and the block size is 2^{10} bytes
  - An array implementation needs 2^{20} disk accesses
  - If "perfectly streamed", > 4 seconds
  - If "random places on disk", 8000 seconds (> 2 hours)
  - A list implementation in the worst case needs 2^{23} "random" disk accesses (> 16 hours) – probably not that bad
- Note: "array" doesn't mean "good"
  - Binary heaps "make big jumps" to percolate (different block)

BSTs?

- Looking things up in balanced binary search trees is $O(\texttt{log}\ n)$, so even for $n = 2^{39}$ (512GB) we need not worry about minutes or hours
- Still, number of disk accesses matters
  - AVL tree could have height of 55
  - So each **`find`** could take about 0.5 seconds or about 100 finds a minute
  - Most of the nodes will be on disk: the tree is shallow, but it is still many gigabytes big so the *tree* cannot fit in memory
    - Even if memory holds the first 25 nodes on our path, we still need 30 disk accesses

Note about numbers; moral

- All the numbers in this lecture are “ballpark” “back of the envelope” figures
- Even if they are off by, say, a factor of 5, the moral is the same: If your data structure is mostly on disk, you want to minimize disk accesses
- A better data structure in this setting would exploit the block size and relatively fast memory access to avoid disk accesses…