CMPSCI 630 Systems **Fall 2019**

Lecture 3

Lecturer: Emery Berger *Scribe: Arta Razavi*

3.1 Memory Hierarchy

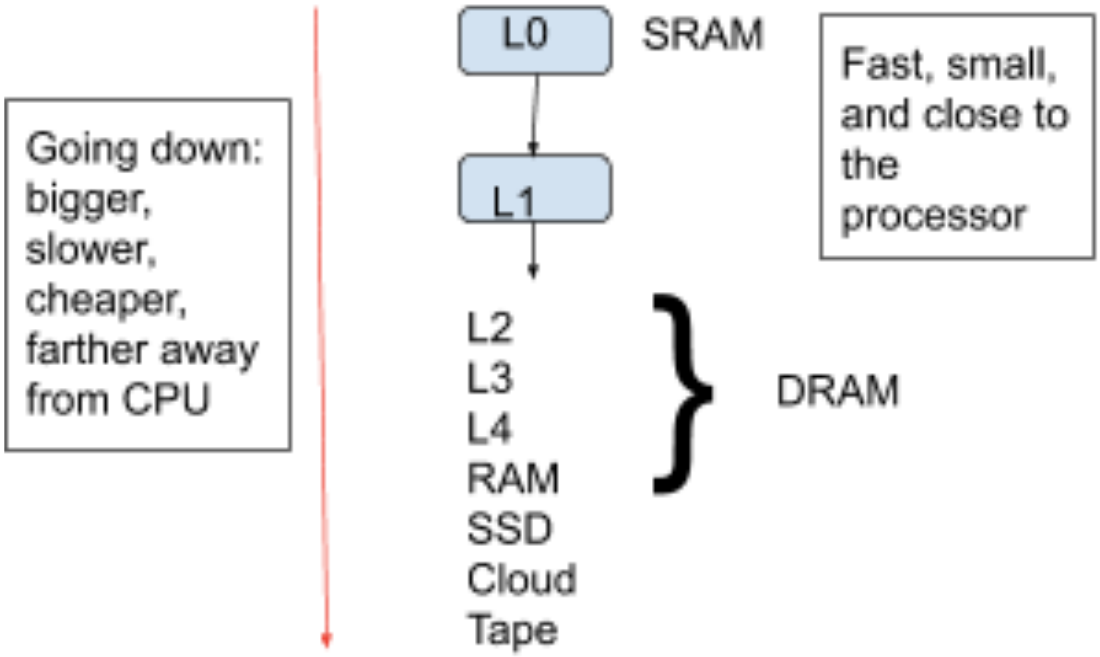

Figure 3.1: memory hierarchy

Non volatile Memory (NVM): persistent, when power is turned off the memory is still there. This is viewed as a new possible replacement for RAM
transparency: programs running on a computer do not know how many levels of cache there are or how big they are or anything about memory. There is an abstraction where programs think they are running on RAM.

C and C++ have direct access to virtual addresses but programming languages like Python, Java, and JavaScript do not.

3.2 Cache Lines

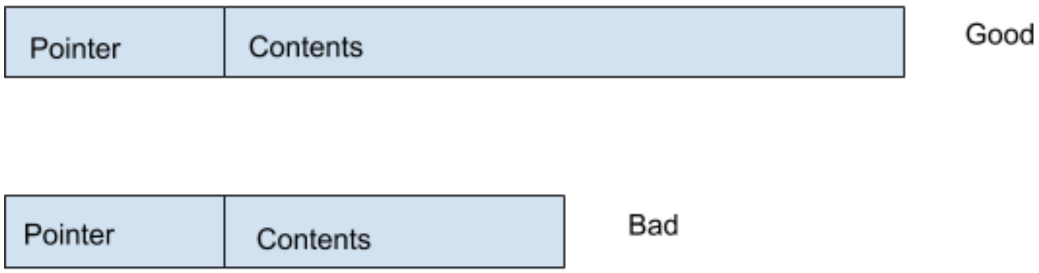

Figure 3.2: cache lines

There is a directory for which things are present in the cache. In the directory addresses are stores as pointers which point to a bunch of memory (contents). These contents are known as cache lines. Cache Lines are usually large, this amortizes management overhead of storing pointer addresses known as metadata. We want the overhead of storing pointer addresses to be smaller so we take larger chunks of content as cache lines. We wish for the size of the data to be larger than the size of the metadata.
Addresses are rounded up to multiples 128 or 256 bits and you store that chunk. If asking for address 3 its going to be rounded to cache line 0 and 259 would return cache line 256.

3.3 Modern Chip

CPU cores on modern chips each have their own individual caches as well as shared caches between pairs.
Die Stacking: The idea if that instead of having chips be on the same level stack them on top of each other to promote closeness. Less communication and more proximity of data to the chip.
Bus: The way you get contents on the chip from the RAM and back. CPUs are spinning fast and there are many cores and they are all working hard but everything is bottle necked by the bus.
Registers: Inside CPU there are registers which are faster memory that is accessible in one cycle (this is even faster than L0). They are assigned named variables A-F and you can read from and write to them.

3.4 Cache Storage

CPU asks for a byte from an address then at first it asks L0 and if L0 doesn't have it then L1 and so on and if none of them have it eventually the CPU asks the RAM. Where is this content going to go after it was retrieved from RAM?
Inclusive: every cache level has a super set of data addresses stored in (level - 1).
LRU: caches get filled up pretty quickly so we must use a replacement policy to move new data in and old data out of caches. The standard policy is least recently used (but not really there is a variation of this). As caches grow a bigger and bigger LRU is not favored because we do not want to be doing a linear search

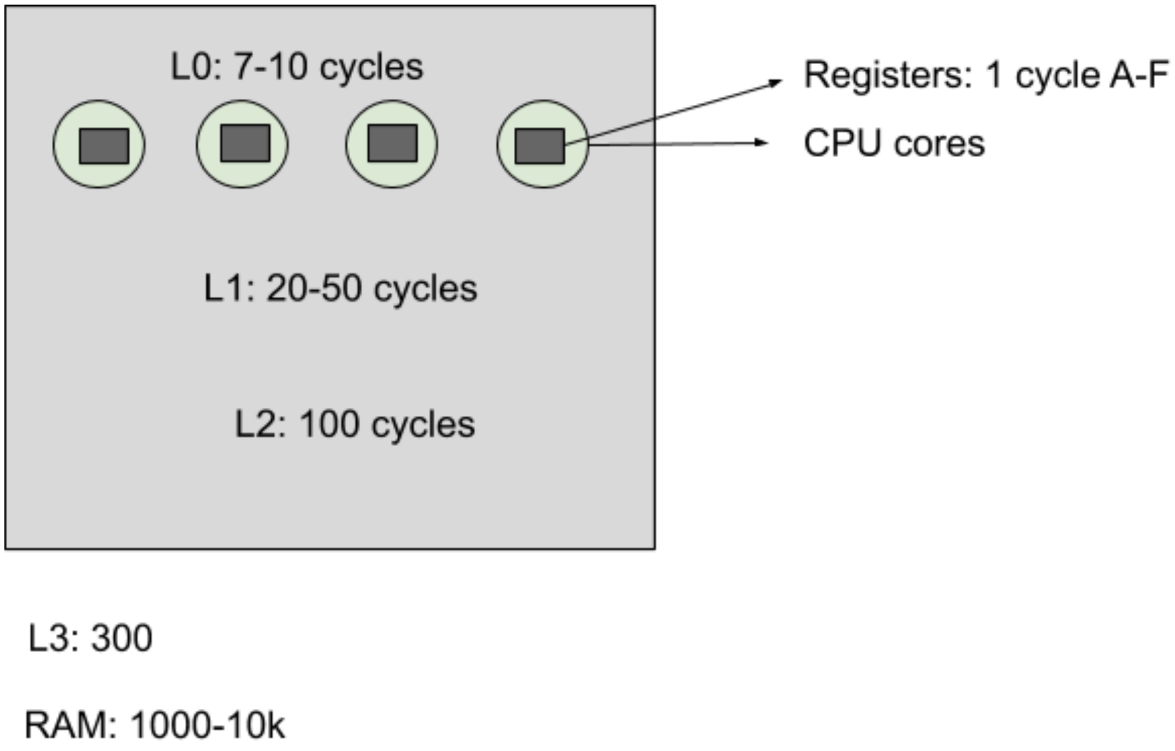

Figure 3.3: modern chip

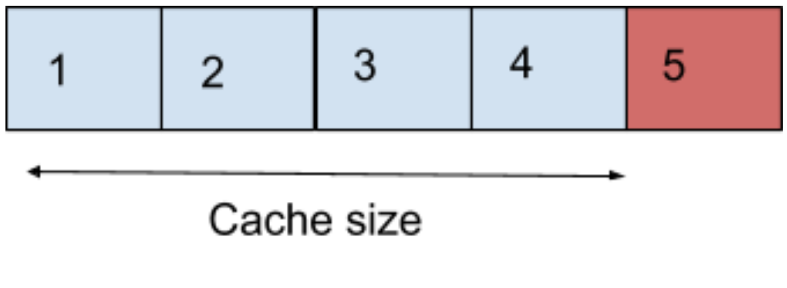

Figure 3.4: cache storage

to find least recently used items.

Exclusive: Data is only stored in one level of cache. If data is evicted and we need access to it again we

might have to go all the way back to RAM to get that data again (bad).
Latency: The time spent waiting for data. Caches are a way to hide latency.

As seen in Figure 3.4: Say we have 5 chunks of data but only 4 spots in our cache. If we access the data in order 1,2,3,4,5 when 5 comes in 1 is evicted. If we circle back and ask for the data again from the beginning we would have to get every piece again from RAM because based on the LRU policy each one would get evicted right before its needed again. This is the worst case scenario for the exclusive policy. The inclusive policy performs better in this situation because there are multiple copies of the data stored across multiple caches.

3.5 Multiple Processors/Cores

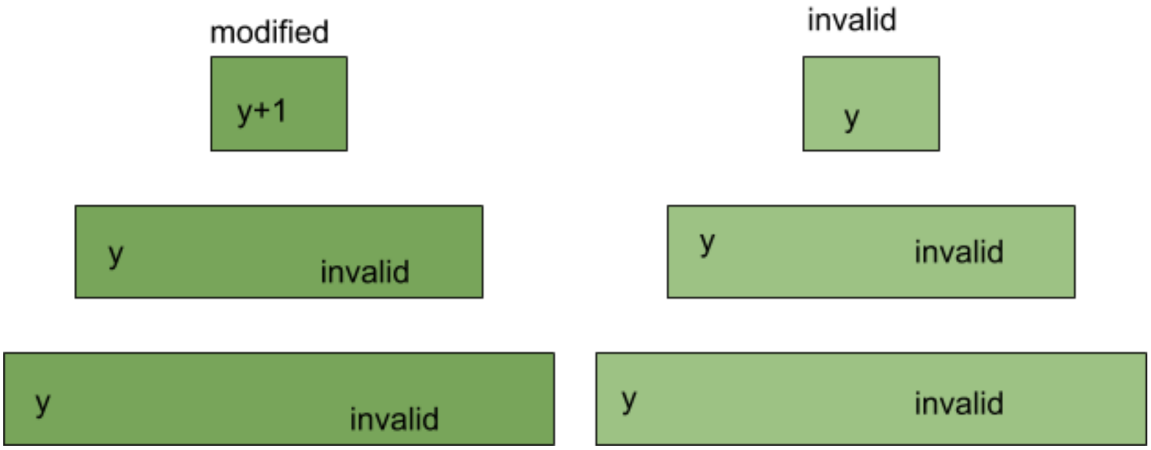

Figure 3.5: multiple processors

Suppose there are two processors and they are both storing values for the variable y. When y gets modified in one processor then the other values of y stored in the other caches becomes invalid.
Coherence: Everyone in the world must see the same view. The processor assures everything has the same view of memory in the machine at all times.

There are different policies for when data gets propagated down levels of cache
Write back: changes only local (faster most typically used)
Write through: changes are propagated all the way down the hierarchy (slower)

3.6 MESI Protocol

A write back protocol that to keeps track of modified data and sync up memory. This is because if there has been a modification made to data it eventually needs to make its way out to RAM. This has 2 bits that determine the value these bits signify 4 variables:
Modified (M): The data is only present in the current cache and it has been modified (M) from value that was stored in memory. This change needs to be propagated to main memory before allowing any farther read of the no longer valid memory. If a write back occurs this changes to shared state (S).

Exclusive (E): The data is only present in the current cache but it matches the data in main memory. This can respond to read calls.
Shared (S): This data may be present in other caches on the machine. The data matches main memory.
Invalid (I): This data is invalid and does not match main meory.
If the 2 bits are modified and exclusive it means I changed the data and I'm the only one that has a copy of it.
If the 2 bits are shared and invalid it means Several CPUs have an unmodified version of it. (Trash)
In an inclusive cache every single level is implicated but in an exclusive cache only one level is implicated. When this type of sharing across multiple processors is taking place an exclusive cache performs much faster.

3.7 Approximate LRU Implementation

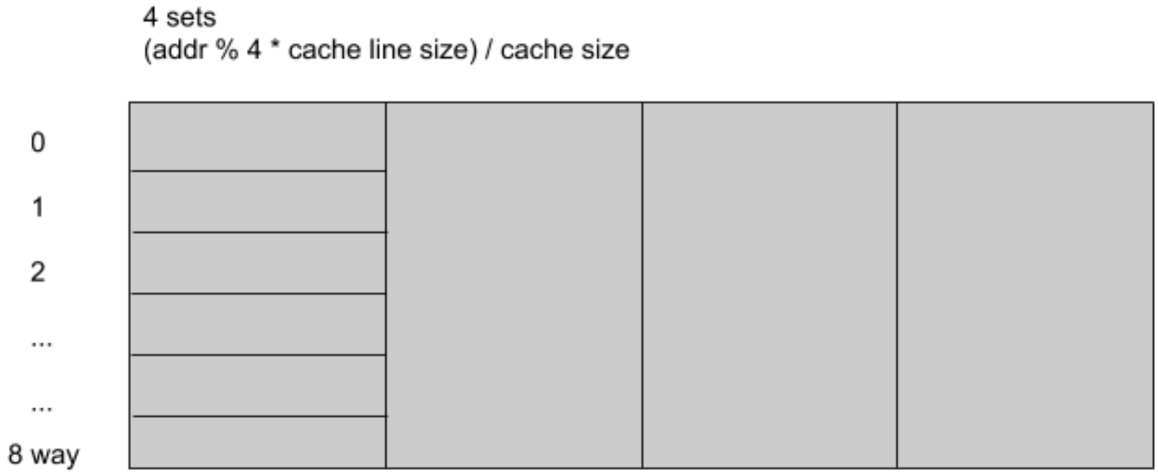

Figure 3.6: approximate LRU

Storing a bunch of addresses and doing a linear search of all addresses upon eviction is not efficient. Instead, we divide up addresses by hashing them into different bins which are refereed to as sets. Now memory address management is divided up into groups and there is only a fixed number of addresses in each set to track instead of linearly scanning all addresses. In Figure3.6 above in a 8 way set we are only performing LRU on 8 addresses instead of all of them.
We do this by storing counters and every time we access an address stored we will increment the counter. The counter starts at 00 and goes all the way up to 11.
Saturated Counter: when the counter reaches 11 we no longer update it and its considered to be a saturated counter.
When it comes time to evict we start evicting items with the lowest counters first. Every so often we reset all the bits back to 0.
bad scenario: you have a program where all the memory accesses only hashes into one of the sets. Then you are only using a fraction of your cache space by sheer bad luck and this is inefficient.

3.8 Cache Misses

3 Cs model:
Capacity: An attempt was made to access memory that is too large and could not fit into the cache
Conflict: hash collisions in the LRU approximation table. A bunch of addresses hashed to the same spot in the table and collided.
Coherence: Because of sharing, when someone goes to access something then it gets modified then their copy is invalid.

3.9 Cache packing

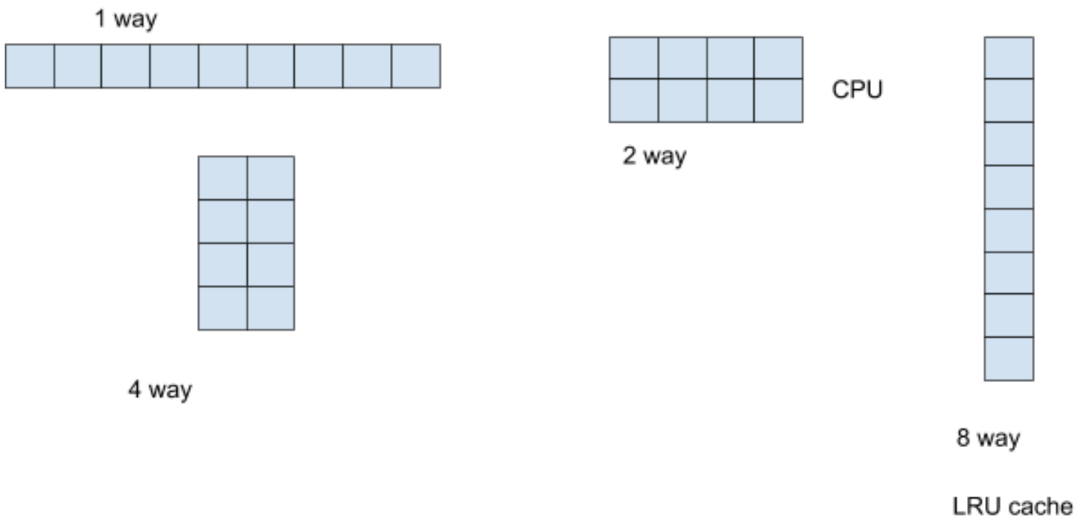

Figure 3.7: cache packing

These are many degrees of packing that goes on when you divide up all caches.
Direct Mapped (1-way)
2-way
4-way
8-way
16-way
Fully associative (infinity way)
Going down the hierarchy the caches lengths are halved and they are made twice as tall. In the case of a hash collision you store the values in the bins. In the 1 way hash if everything hashed to the same bin that would be really bad because it can only hold one value. In the 4 way case there are 4 spots for collisions. Within each x-way bin you perform LRU.

3.10 Translation Lookaside Buffer (TLB)

This is a fully associative cache because we do not want to have misses in this cache. Fully associative means that data can be stored in any unused space in the cache instead of forcing data to be stored into specific spots in the cache and there are no conflicts. The TLB is a cache of mappings from virtual to physical addresses.
When memory is accessed on a processor, processes are never actually accessing memory by its real address they are instead accessing it by its virtual address.

3.11 Process Isolation

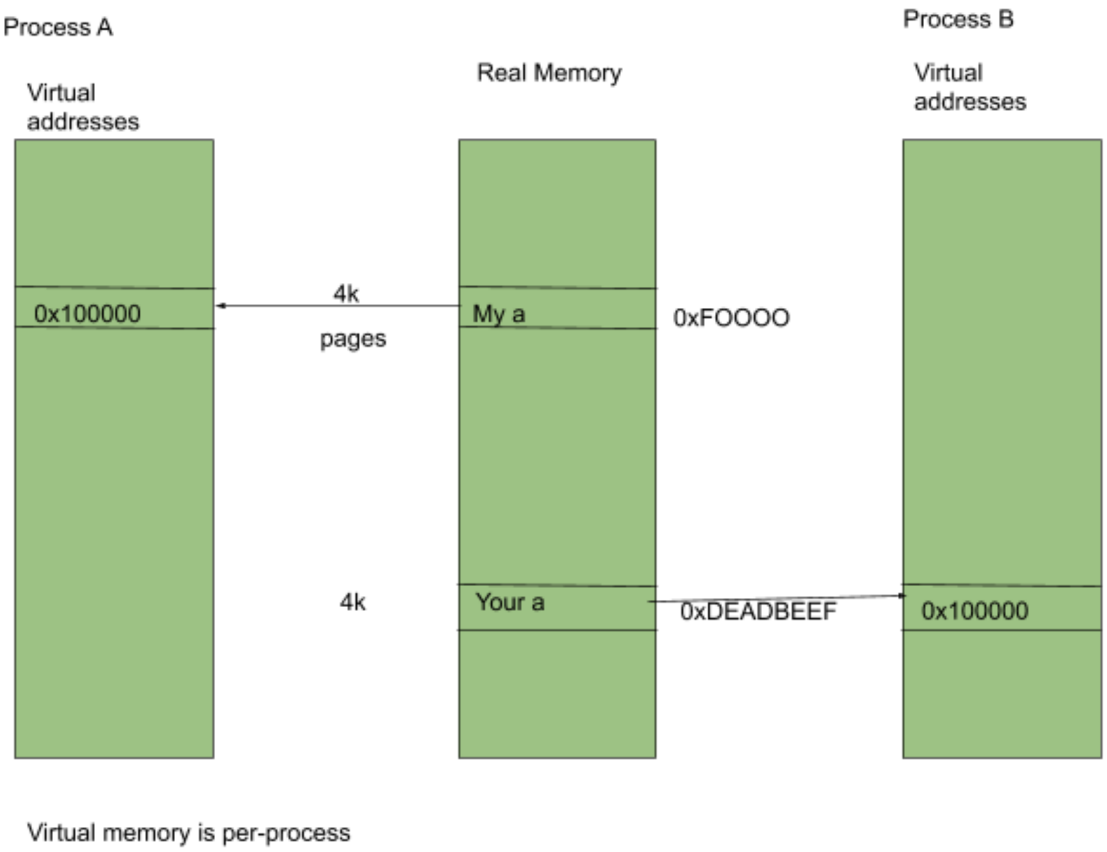

Figure 3.8: process isolation

Every process has their own view of the world and can never see the content of other processes running on the machine. Pieces of memory from different processes can have the same address because processes use virtual address which are just pretend addresses.
pages: real memory holds processes in different chunks of memory referred to as pages and they are a range of memory which have their own unique addresses and are stored in multiples of 4k.
Processes create a one per process virtual map referred to as virtual memory which is a mapping of locations

in real physical memory to the virtualized memory of the process.
Page Table: data structure that is stored in a TLB which contains virtual address to real physical memory address pairs. The structure of this table is defined by the hardware.
Multi-level TLBs: When data does not fit in the TLB the process has to go to main memory to get the data and that was very slow. TLBs have 1 cycle access time which is very fast and desirable. There are TLBs with multiple levels starting from a smaller TLB ranging down to larger TLBs so we can store more data and avoid going out to main memory.

3.12 Process Switching

The content of the TLB is also per-process. If a virtual address appears in the TLB it means the process has access to it (it works like a key) this is referred to as capability. Very important when we change from one process to another process they do not get access to each others addresses.
Invalidate TLB entries: Go through the TLB and invalidate all addresses from old process and load in valid addresses from new process.
TLB Flush: When there is a process switch and the TLB needs to get cleared it discards all entries from the previous process and loads the new process.
Invalidation is good in the case of 2 processes sharing TLB back and fourth and one of them needs access to lots of memory and the other needs access to just a few things. This is better than performing a TLB Flush because you don't have to re-load all memory for the expensive process just go through and re-validate those entries in the TLB saves time going back to main memory.

3.13 Scheduling

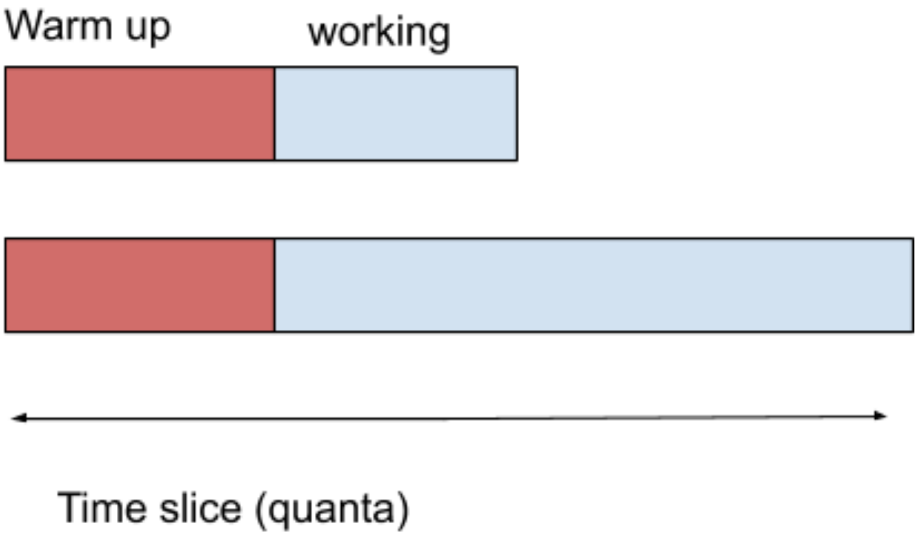

Figure 3.9: scheduling

Time Slicing: The CPU constantly is switching between processes allowing each process to run for a certain

amount of time and that is how scheduling works.
This amount of time is hard to chose because we want to minimize the slice of time per process but at the same time each process needs warm up time to fill its caches and TLBs but also to needs to have time to perform some tasks. With warm up time in mind, if this slice of time is too narrow the processes will warm up and be short on time to perform tasks. If this time slice is too long processes get too much time on the CPU and other processes get stuck waiting. There has to be a perfect balance so we wish to amortize the warm up across the time quanta. Multi cores fix this issue because the machine could be warming up for some processes and running other processes at the same time.
This switching is known a context switch and the cost of warming caches up is referred to as context switch overhead
Context switch overhead is high for processes and low for threads. This is because in the case of threads the TLB only has to to save registers and all the process state stays the same so there is practically no context switching overhead.
Warming up the cache: When processes are given their turn on the CPU they need to get a bunch of memory from main memory referred to as a "working set" onto their cache.
Cold Cache: The cache for a freshly loaded process is empty and its called cold cache.

3.14 Registers:

There is a fixed number of registers example: A B C D
There is some number of variables in your program example: x y z a b c q r s
We do not have a register for every variable (registers < variables) so we must figure out which copy of variables get stored in which registers and when to switch them out for other variables.

Register allocation: maximise the number of time variables are held in registers. When things do not fit in registers the contents of that register would have to be written out to main memory in order to allow other variables to use that register. This is kind of like scheduling.
Spill: copy contents of register to main memory (eviction)
If the program is straight line code register allocation is predictable and this is reduced down to a graph coloring problem which is NP complete. For small problems NP complete is fine but for high numbers NP complete is not good.

Linear scan: scan through the registers once and assign registers