COS 217: Introduction to Programming Systems

The Process and Virtual Memory Abstractions



PRINCETON UNIVERSITY

The Illusions and the Reality



We as programmers assume:

1. While our program runs, we "own" the CPU : Only our program is running on it
2. There is a huge amount of contiguous memory available to us (from address 0 to address 2^{64} -1), and it is all ours to use freely

The reality is:

A. Many programs run at the same time on and share the CPU
B. The amount of hardware memory is comparatively very small, and all programs share it and use the same addresses. (So why aren't there collisions?)
   - The hw configuration (#CPUs, memory) varies across machines

1. Abstracting the CPU: the Process
2. Abstracting memory: Virtual Memory

Abstracting the CPU: the Process



Program

- Executable code
- A static entity

Process

- An instance of a program in execution
- A dynamic entity that steps through the program
  - E.g. the process with Process ID 12345 might be running `emacs`
- A process has state at any point in time
  - Where am I in the code, values of registers, values in memory, …
- Multiple processes may be running the same program
  - E.g. PIDs 12345 and 23456 might both be running `emacs` – even for the same user
  - They have different state (currently at different points in the code, with different values)

Every Process is Given Two Key Illusions



1. That it owns the whole CPU, the whole time it runs
2. That it owns all the memory the machine has (or could have)

Abstracting the CPU



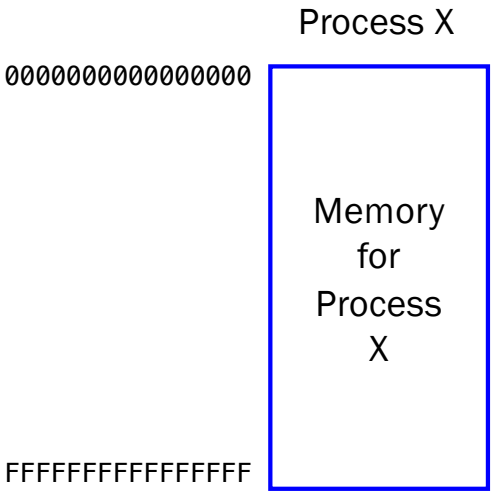
Many processes share the CPU(s)

- Time-sharing and space-sharing

The operating system and the hardware together manage:

- The multiplexing of processes
- The scheduling of processes
- The resource management and protection needed

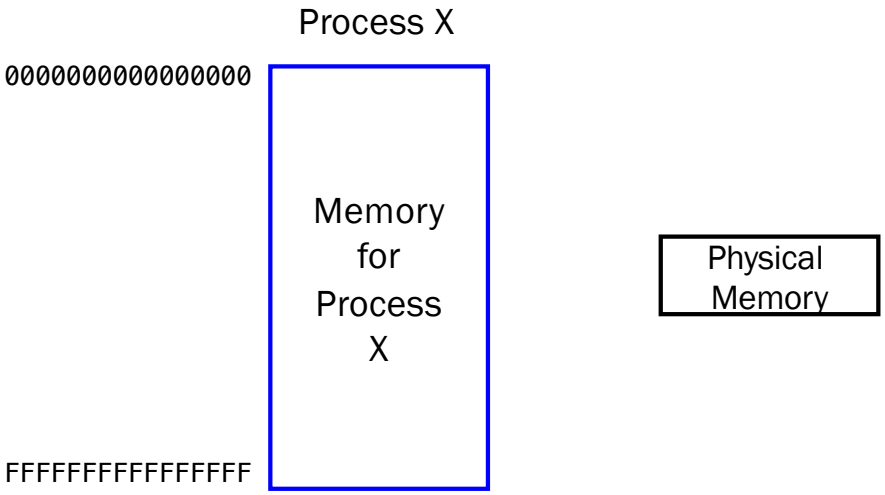
Abstracting Memory: The Address Space



Every process sees memory as

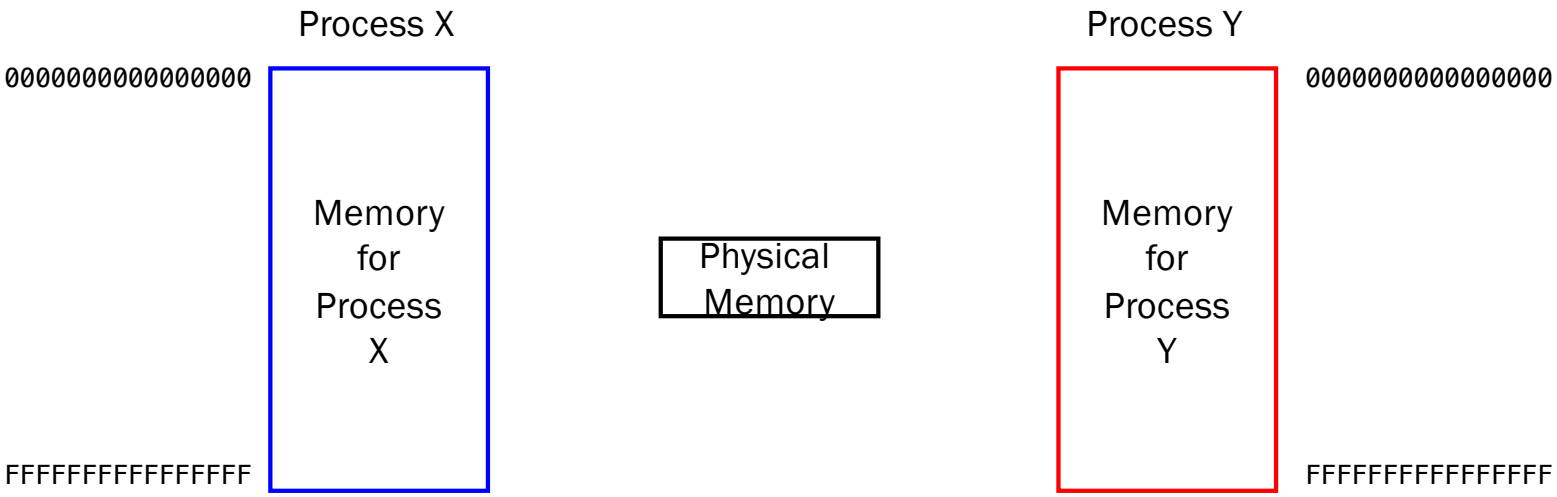
- Huge: 2^{64} = 16 EB (16 exabytes) ≈ 10^{19} bytes
- Uniform: contiguous memory locations from 0 to 2^{64}-1

Problem 1



But physical memory is *much* smaller (16GB?)

Problem 2



There are *many* processes, each with the address space of 2^{64} bytes
But only the same one little physical memory

The Solution: Virtual Memory



Process references a virtual address (any address in its 0 to 2^{64} - 1 range)

OS + hardware translate (map) it to a physical address (say 0 to 2^{24} – 1)

Process 12345's virtual 5213 will translate to a different physical address than process 23456's virtual address 5213

- Even if the processes are both running emacs; OS/hw track process IDs too in mapping

Some VAs map to an address in physical memory right now; others are on disk

- OS and hardware "swap" data in and out of physical memory, to/from disk, as needed

OS and hardware make sure my data can't clobber yours (provide protection)

OS and hardware maintain these mappings

But, keeping mappings for every virtual address would be a huge amount of storage

Do it at bigger granularity: map chunks of contiguous addresses, not individual ones

The Solution: Virtual Memory



OS and hardware make sure my data can't clobber yours (provide protection)

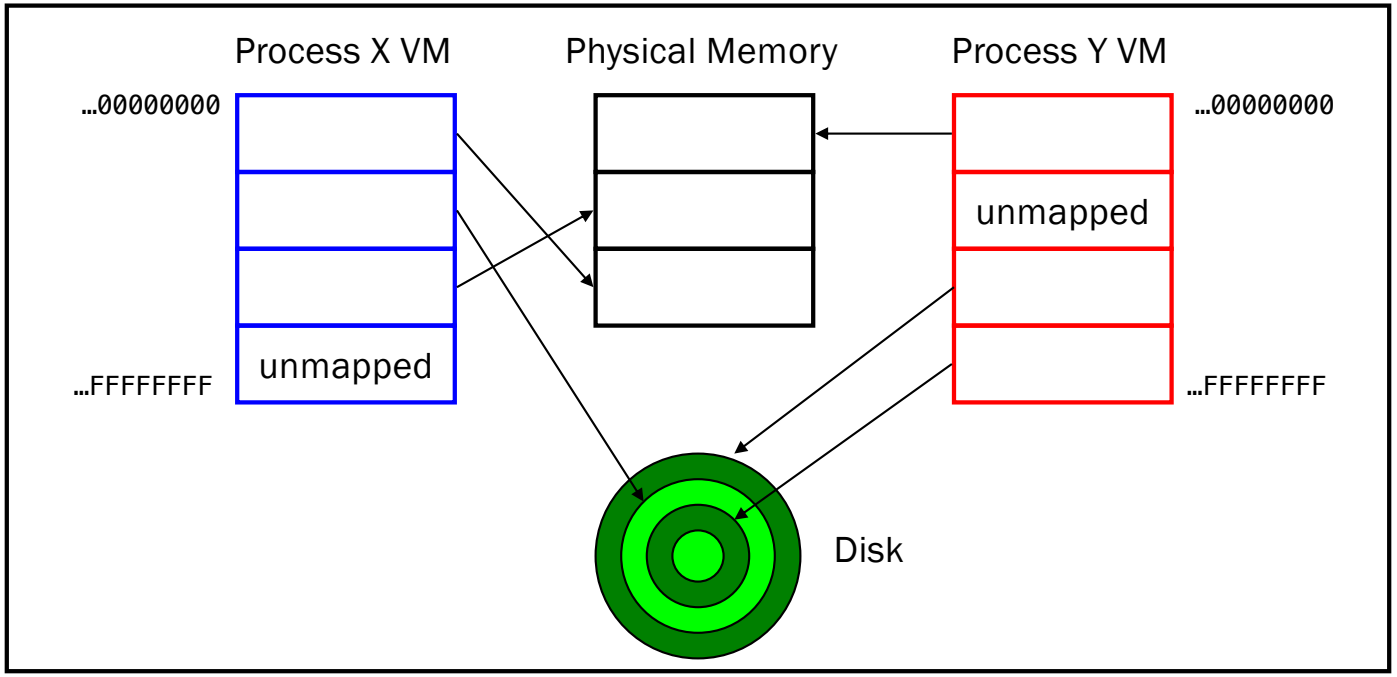
OS and hardware maintain these mappings

But keeping mappings for every virtual address would be a huge amount of storage

Do it at bigger granularity: map chunks of contiguous addresses, not individual ones

- **Fixed-size pages, treated as indivisible units. Say 16KB each.**
- **Map (track) pages, not bytes; move pages, not bytes**

How Virtual Memory is Implemented



Memory is divided into pages: fixed-size groups of contiguous addresses

- At any time, some pages are in physical memory, some on disk
- OS and hardware swap (entire) pages between physical memory and disk as needed
- Note: a virtual page is the same size as a physical page

How Virtual Memory works



Program issues a virtual address (by referencing a variable, say), that is valid in its virtual address space

OS + HW translate convert that address to a virtual page number in that VA space

- by stripping out the lower 16 bits (the byte offset within the page, which stays same from V → P)

They:

- Map the virtual page number to a physical page number in the PA space
- Check its validity for the type of access the program is trying to make, and
- Bring it into memory and set it up there if it wasn't already
- Convert the physical page number to physical byte address by appending the 16 bits of the byte offset within the page
- Access the data and the physical byte address

Virtual & Physical Addresses (cont.)



Virtual address



- Identifies a location in a particular process's virtual memory
  - Independent of size of physical memory
  - Independent of other concurrent processes
- Consists of virtual page number & offset
- Used by **application programs**

Physical address



- Identifies a location in physical memory
- Consists of physical page number & offset
- Known only to **OS** and **hardware**

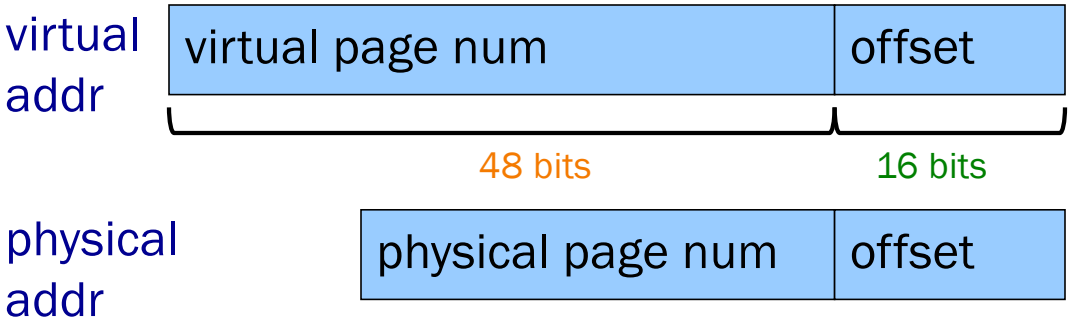
Mapping:



Note: To allow mappings to be maintained at larger granularity (that of pages):

- The offset in a virtual address and its corr. physical address is the same (byte offset within page)
- The physical page number may be (is) different than the virtual one, but the offsets are the same

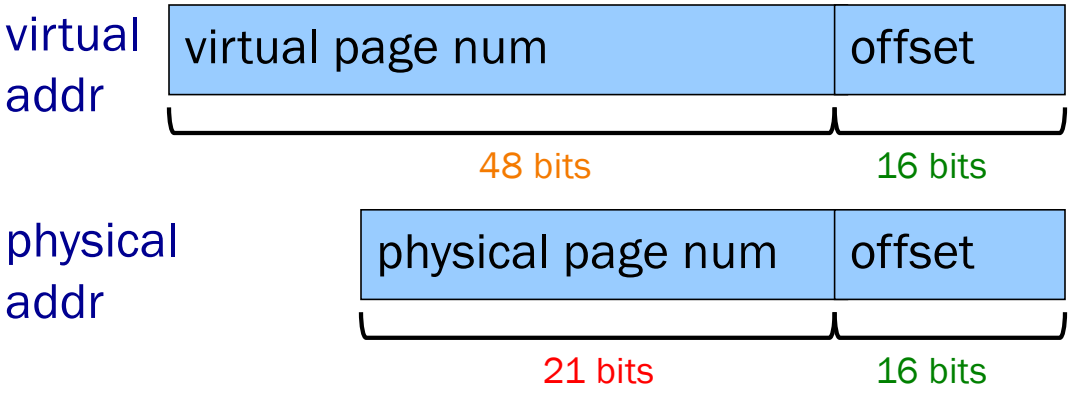
ArmLab Virtual & Physical Addresses



On ArmLab:

- A virtual address is 64 bits long (2^{64} bytes of virtual memory per process)
- The offset is 16 bits long (A page has 2^{16} bytes)
- A virtual page number is 64 – 16 = 48 bits long (2^{48} virtual pages per process)

ArmLab Virtual & Physical Addresses



On ArmLab:

- A physical address is 37 bits long (2^{37} bytes, or 128GB of physical memory in every computer)
- The offset is 16 bits long (A page has 2^{16} bytes, same as a virtual page)
- A physical page number is 37 – 16 = 21 bits long (2^{21} physical pages in every computer)



Page Numbers and Offsets



On Armlab, if the VA of variable a in process 78 is page 7053, byte offset 12,652, and the physical page number mapped to is 1046, what is the physical address of a?

A. Page 7053, byte offset 12,652

B. Page 1046, byte offset 12,652

C. Page 1046, byte offset 7053

D. Page 1046, byte offset unknown

B. Byte offset stays the same.

That's the whole point, that allow us to maintain mappings and protection at the granularity of pages, not bytes.

How is the Mapping Maintained: Page Tables



A page table per process

Page Table for Process 1234

Virtual Page Num	Physical Page Num or Disk Addr
0	Page 5 in phys mem
1	(unmapped)
2	Page X on disk
3	Page 8 in hys mem
...	...

Page table maps each in-use virtual page to:

- A page in physical memory, or
- A page on disk

Again, no need to map the offsets, just page numbers

Storing Page Tables



Page tables (mappings) for active processes are stored in memory

Process looks up mappings there

If the mapping isn't there, it can be painful for performance

- Also, it's the page table that tells you the page is on disk, but now the page table is on disk …

OS manages this

- Special logic in OS "pins" active page tables to physical memory, so they aren't swapped to disk

Anatomy of a Memory Reference



- Process executes instruction that references virtual memory
- CPU determines virtual page
- CPU looks up page table (let's assume it is in memory)
- CPU checks if required virtual page is in physical memory: no
  - CPU generates page fault
  - OS gains control of CPU
  - OS (potentially) evicts some page from physical memory to disk, loads required page from disk to physical memory
  - OS returns control of CPU to process – to same instruction
- Process executes instruction that references virtual memory
- CPU checks if required virtual page is in physical memory: yes
- CPU does load/store from/to physical memory

Virtual Memory also Provides Protection



Memory protection among processes

- Process's page table references only physical memory pages that the process currently owns
- Process can't accidentally/maliciously affect physical memory used by another process

Memory protection within processes

- Permission bits in page-table entries indicate whether page is read-only, etc.
- Allows CPU to prohibit
  - Writing to RODATA & TEXT sections
  - Access to protected (OS owned) virtual memory

Revisiting Page Tables…



Question

- Doesn't every logical memory access require **two** physical memory accesses: one to access the page table, and one to access the desired page?

Answer

- Conceptually, yes
  (And page tables are big, and stored hierarchically as trees, so it can be worse than 2)

Question

- Isn't that inefficient?

Answer

- Conceptually: yes, but actually not so much …

Agenda

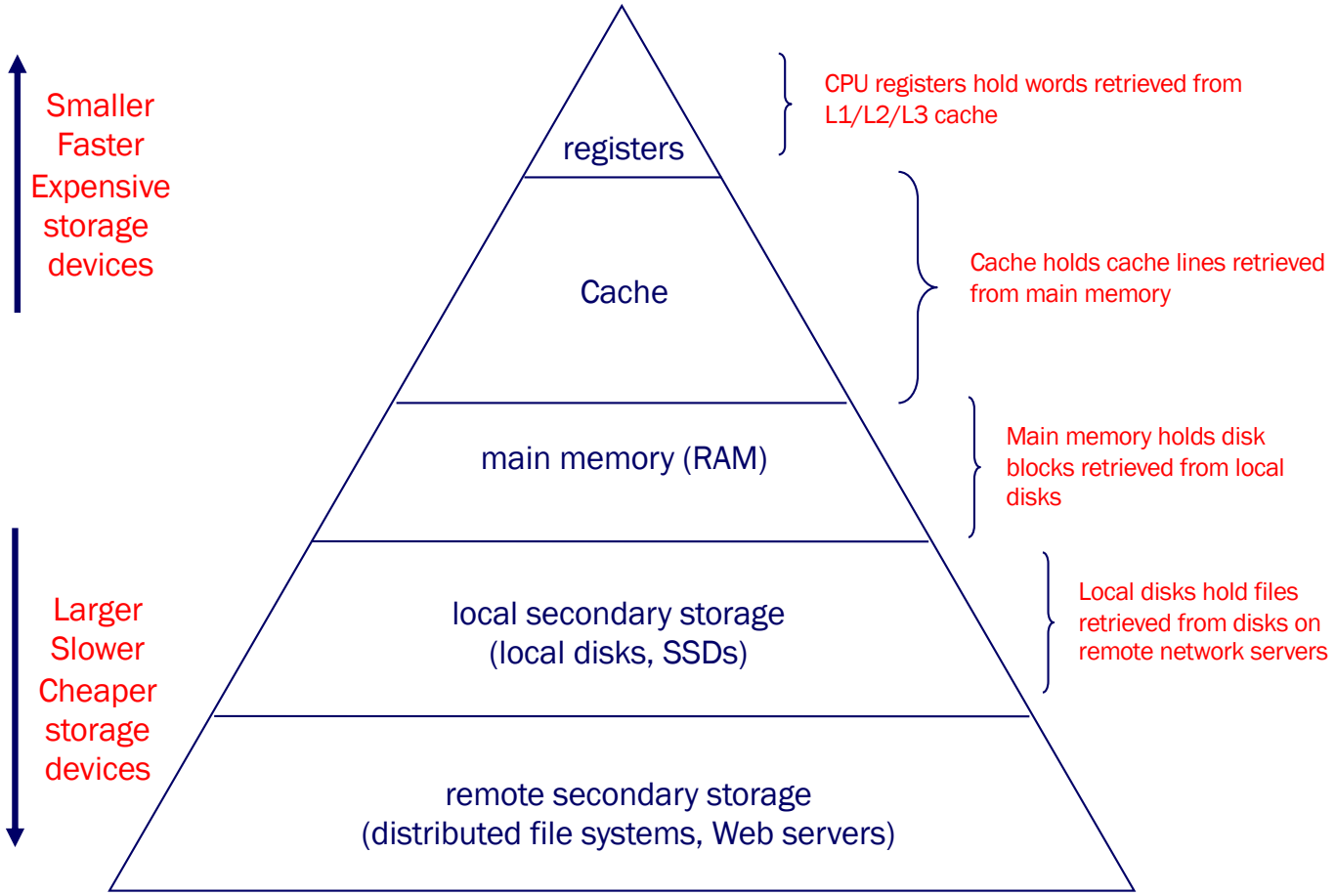


The process abstraction

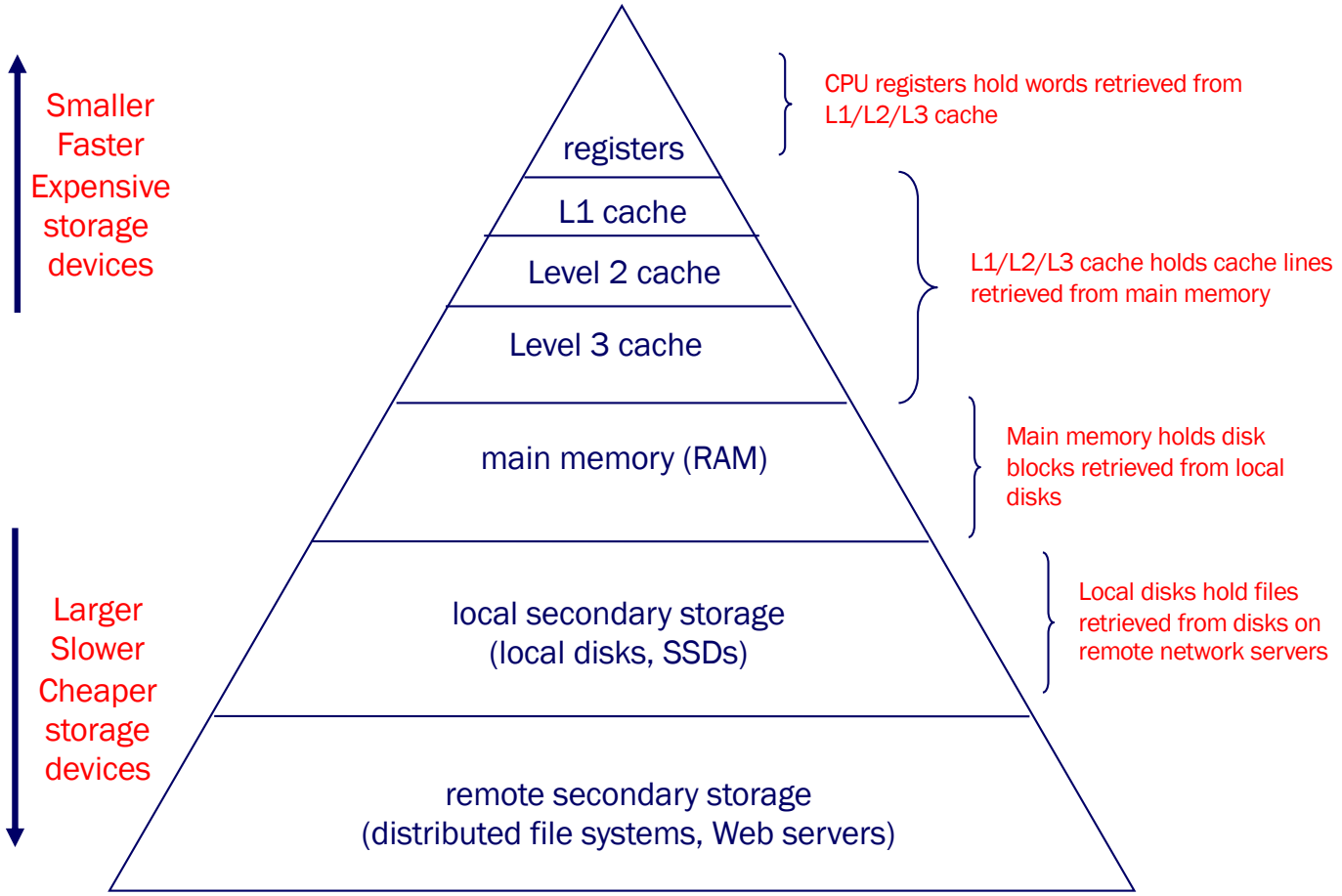
The memory abstraction and memory management

- Virtual Memory
- The Storage Hierarchy, Locality and Caching
- Cache Management

Typical Storage Hierarchy



Typical Storage Hierarchy



Typical Storage Hierarchy



Factors to consider:

- Capacity
- Latency (how long to do a read)
- Bandwidth (how many bytes/sec can be read)
  - After first byte is read, how fast are subsequent bytes read
  - Generally, reading a reasonable amount of data is not much slower than reading one byte
- Volatility
  - Do data persist in the absence of power?

Typical Storage Hierarchy





Registers

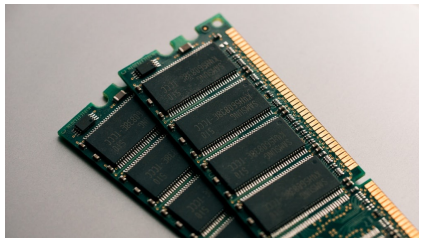
- **Latency**: 0 cycles
- **Capacity**: 8-256 registers (31 8-byte general purpose registers in AArch64)

L1/L2/L3 Cache

- **Latency**: 3-50 cycles
- **Capacity**: 32KB to 256MB

Main memory (RAM)

- **Latency**: ~ 200-400 cycles
- **Capacity**: up to 100s of GB



Typical Storage Hierarchy



Local secondary storage: disk drives



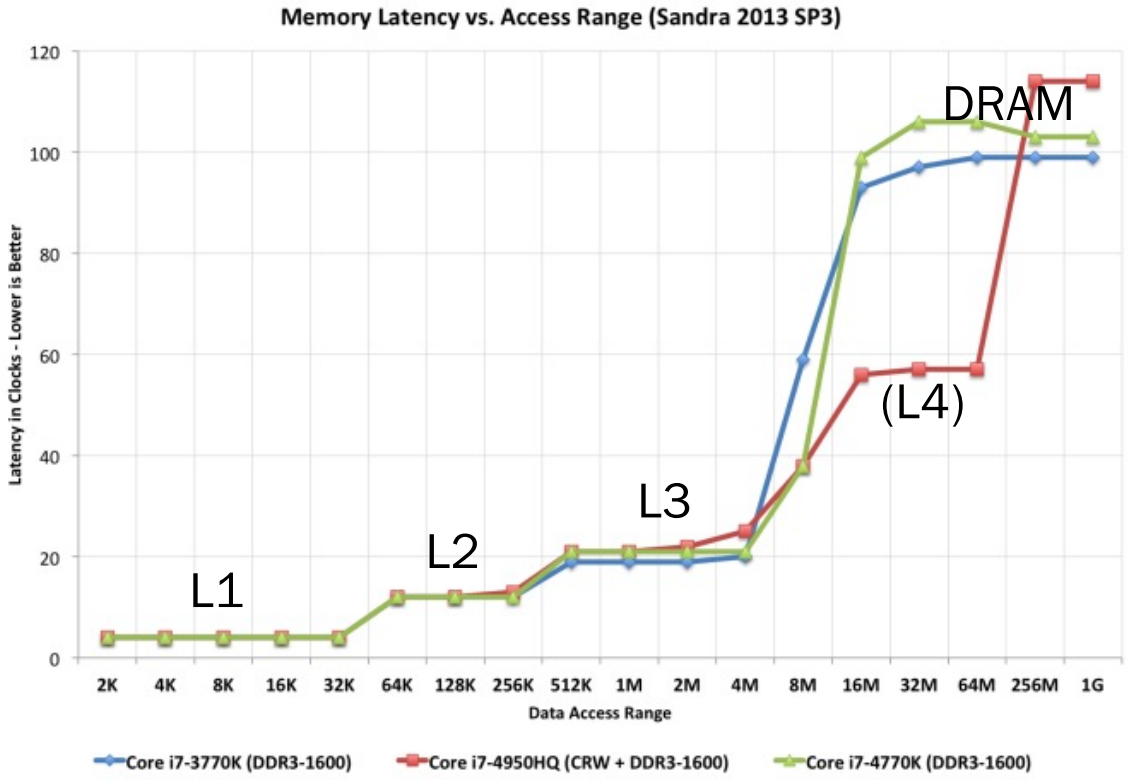
- Solid-State Disk (SSD):
  - Flash memory (nonvolatile)
  - **Latency**: 0.1 ms (~ 300k cycles)
  - **Capacity**: 128 GB – several TB



- Hard Disk (HDD):
  - Spinning magnetic platters, moving heads
  - **Latency**: 10 ms (~ 30M cycles)
  - **Capacity**: 1 – tens of TB

@benjaminlehman , Samsung Belgium

Cache / RAM Latency



1 clock = $3 \cdot 10^{-10}$ sec https://www.anandtech.com/show/6993/intel-iris-pro-5200-graphics-review-core-i74950hq-tested/3

Disks

HDD

1 ms

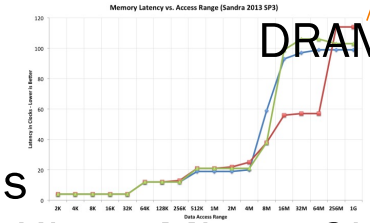


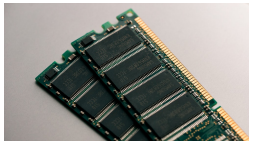
SSD

1 µs



DRAM





1 ns

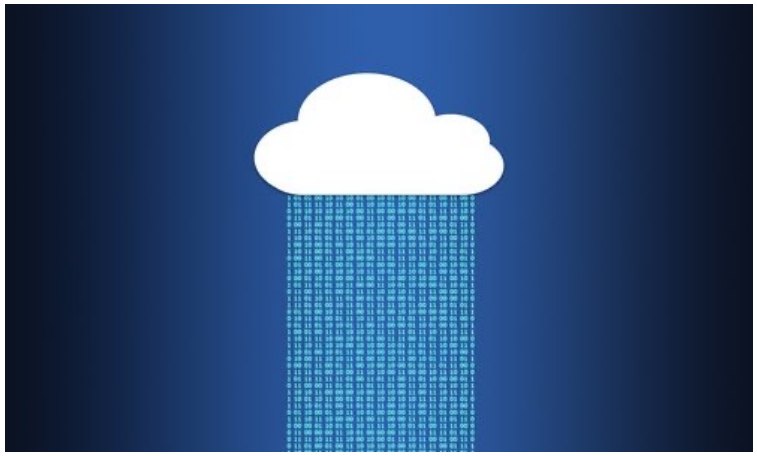
Kb Mb Gb Tb

Typical Storage Hierarchy



Remote secondary storage (a.k.a. “the cloud”)

- **Latency**: tens of milliseconds
  - Limited by the speed of light (and network bandwidth)
- **Capacity**: essentially unlimited



@TheDigitalArtist

Storage Device Speed vs. Size



Facts:

- **CPU** needs sub-nanosecond access to data to run instructions at full speed
- **Fast** storage (sub-nanosecond) is small (100-1000 bytes)
- **Big** storage (gigabytes) is slow (15 nanoseconds)
- **Huge** storage (terabytes) is *glacially* slow (milliseconds)

We need another illusion:

- Many, many gigabytes of memory
- And fast (sub-nanosecond) average access time

Solution: **caching**

- Most of the time, data accesses are satisfied in fast, small levels of memory hierarchy
- But why, if accesses are random or uniform?
- Answer: they are not. Most programs exhibit good **locality of access** to data
  - Such programs benefit from **caching,** which enables good **average** performance

Locality



Two kinds of **locality**

- **Temporal** locality
  - If a program references item X now, then it probably will reference X again soon
  - So once X is accessed, keep X in the fast, small memory for a little while
- **Spatial** locality
  - If a program references item X now, then it probably will reference item at address X±1 soon
  - So fetch more than just X into the fast, small memory when X is brought in

Most programs exhibit very good temporal and spatial locality, on code and on data

Locality Example



Locality example

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
```

Typical code
(good overall locality)

Temporal locality

- *Data:* When CPU accesses `sum`, it accesses `sum` again shortly thereafter
- *Instructions:* When CPU executes `sum += a[i]`, it does that again shortly thereafter

Spatial locality

- *Data:* When CPU accesses `a[i]`, it accesses `a[i+1]` shortly thereafter
- *Instructions:* When CPU executes `sum += a[i]`, it executes `i++` (the next machine language instructions) shortly thereafter

This is Amazing, for Caching



Cache

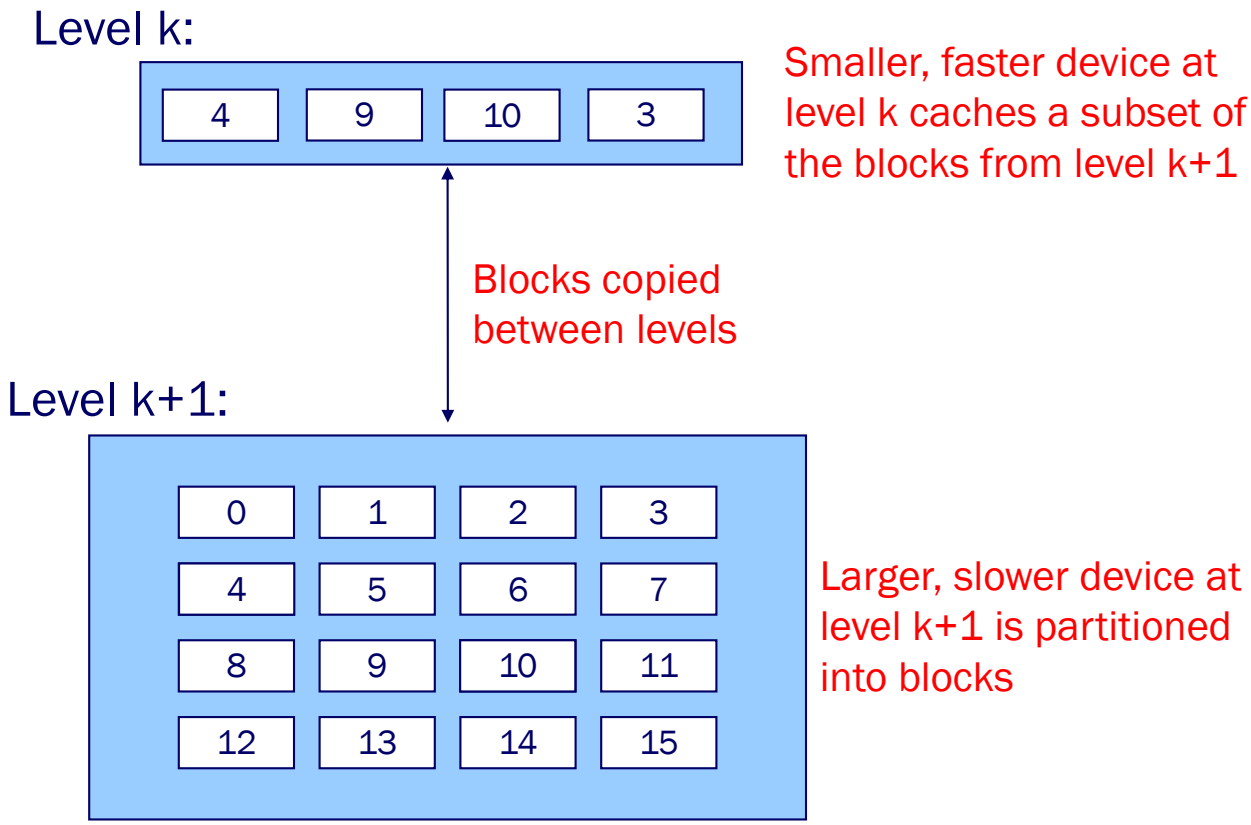
- Fast access, small capacity storage device
- Acts as a staging area for items in a slow access, large capacity storage device
- Items are brought in when they or nearby items are accessed, and kicked out when needed due to limited cache capacity

Good locality + proper caching

⇒ Most storage accesses can be satisfied by cache

⇒ Overall storage performance improved

Caching in a Storage Hierarchy



Cache Hits and Misses



Cache hit

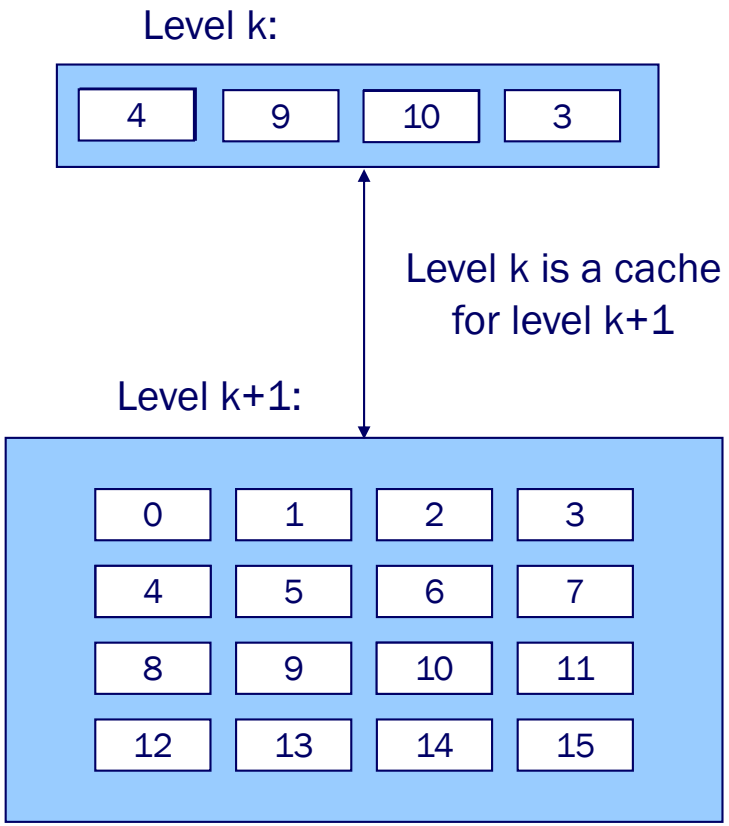
- E.g., request for block 10
- Access block 10 at level k
- Fast

Cache miss

- E.g., request for block 8
- **Evict** some block from level k
- Load block 8 from level k+1 to level k
- Access block 8 at level k
- Slow
- But hopefully next time block 8 is accessed it will still be in cache

Caching goal:

- Maximize cache hits
- Minimize cache misses



Agenda



The process abstraction

The memory abstraction and memory management

- Virtual Memory
- The Storage Hierarchy, Locality and Caching
- Cache Management

Cache Management



Device	Managed by:
Registers (cache of L1/L2/L3 cache and main memory)	Compiler, using complex code-analysis techniques Assembly lang programmer
L1/L2/L3 cache (cache of main memory)	Hardware, using simple algorithms
Main memory (cache of local sec storage)	Hardware and OS, using virtual memory with complex algorithms (since accessing disk is expensive)
Local secondary storage (cache of remote sec storage)	End user, by deciding which files to download

Cache Block Size



Large block size:

+ do data transfer less often
+ take advantage of spatial locality
- longer time to complete data transfer
- can hurt temporal locality, since more data have to be evicted

Small block size: the opposite

Typical: further away memory ⇒ slower data transfer ⇒ larger block sizes

Device	Block Size
Register	8 bytes
L1/L2/L3 cache line	128 bytes
Main memory page	4KB or 64KB
Disk block	512 bytes to 4KB
Disk transfer block	4KB (4096 bytes) to 64MB (67108864 bytes)

Cache Eviction Policies



Best eviction policy: **"oracle"**

- Always evict a block that is *never* accessed again, or…
- Always evict the block accessed the *furthest in the future*
- Impossible to know in the general case

Worst eviction policy

- Always evict the block that will be accessed next
- Causes **thrashing**
- Also doesn't happen in general case

Cache Eviction Policies



Reasonable eviction policy: **LRU policy**

- Evict the "Least Recently Used" (LRU) block
  - With the assumption that it will not be used again (soon)
  - Future behavior will look kind of like recent behavior
- Good for straight-line code
- (Can be) bad for (large) loops
- Expensive to implement
  - Often simpler approximations are used
  - See Wikipedia "Page replacement algorithm" topic

Ordering Operations and Spatial Locality



Matrix multiplication example

- Matrix = two-dimensional array
- Multiply n-by-n matrices A and B
- Store product in matrix C

Performance depends upon

- Effective use of caching (as implemented by **system**)
- Good locality (as implemented by **you**)

Spatial Locality Example: Matrix Multiplication



Two-dimensional arrays are stored in either **row-major** or **column-major** order

a	0	1	2
0	18	19	20
1	21	22	23
2	24	25	26

row-major		col-major	
a[0][0]	18	a[0][0]	18
a[0][1]	19	a[1][0]	21
a[0][2]	20	a[2][0]	24
a[1][0]	21	a[0][1]	19
a[1][1]	22	a[1][1]	22
a[1][2]	23	a[2][1]	25
a[2][0]	24	a[0][2]	20
a[2][1]	25	a[1][2]	23
a[2][2]	26	a[2][2]	26

C uses **row-major** order

- Access in row order ⇒ good spatial locality
- Access in column order ⇒ poor spatial locality

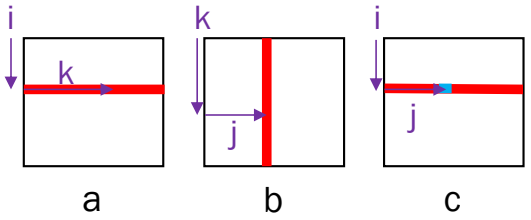
Spatial Locality Example: Matrix Multiplication



```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      for (k=0; k<n; k++)
         c[i][j] += a[i][k] * b[k][j];
```

Reasonable cache effects

- Good spatial locality for A
- Bad spatial locality for B
- Good spatial (and temporal) locality for C



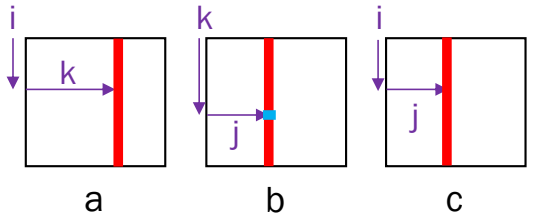
Spatial Locality Example: Matrix Multiplication



```
for (j=0; j<n; j++)
   for (k=0; k<n; k++)
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * b[k][j];
```

Poor cache effects

- Bad spatial locality for A
- Bad spatial locality for B
- Bad spatial locality for C



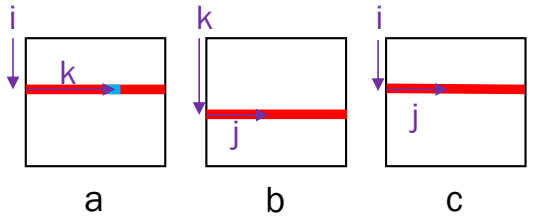
Spatial Locality Example: Matrix Multiplication



```
for (i=0; i<n; i++)
   for (k=0; k<n; k++)
      for (j=0; j<n; j++)
         c[i][j] += a[i][k] * b[k][j];
```

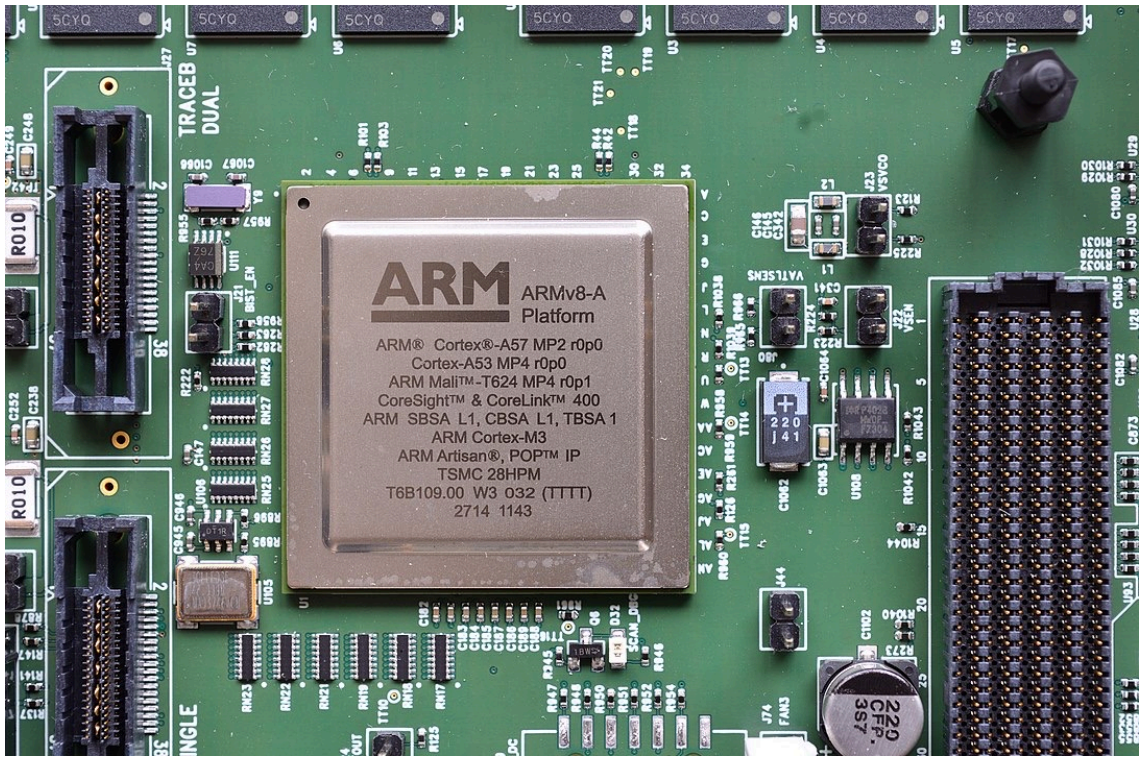
Good cache effects

- Good spatial locality for A
- Good spatial locality for B
- Good spatial locality for C



Next time …

Getting started with ARM assembly language



Lobsterthermidor , Raysonho