COS 217: Introduction to Programming Systems

The Process and Virtual Memory Abstractions

PRINCETON UNIVERSITY

Agenda

The process abstraction

The memory abstraction and memory management

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

Processes

Program

- Executable code
- A static entity

Process

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

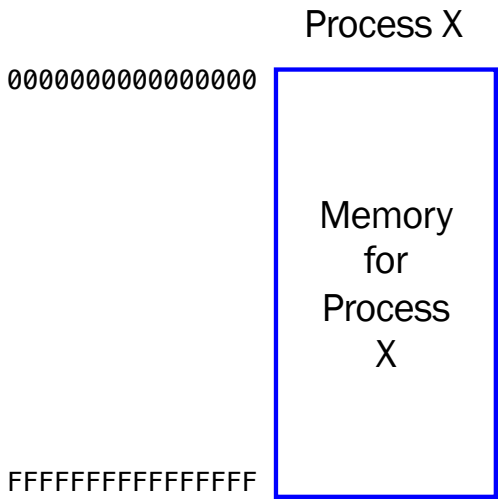
Every Process is Given Two Key Illusions

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)

This is a form of Abstraction

A process is a profound abstraction in computer science

The Address Space Illusion

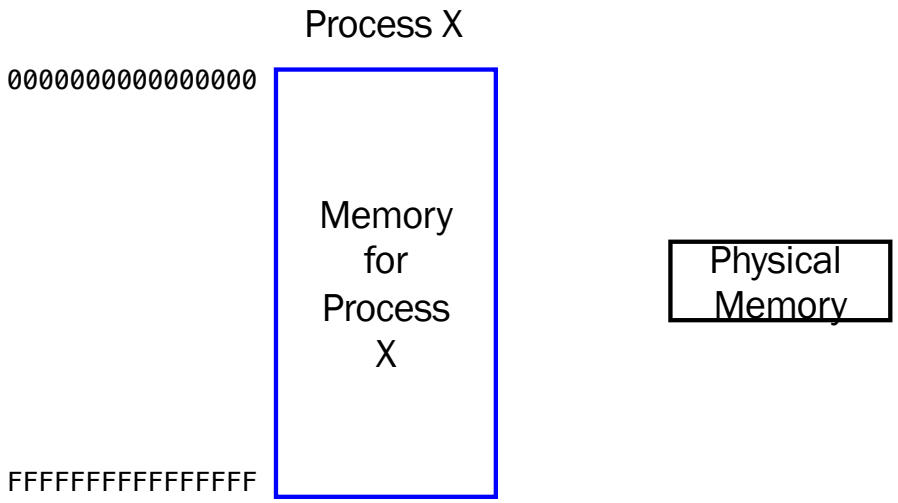


Every process sees memory as

Huge: 2^{64} = 16 EB (16 exabytes) ≈ 10^{19} bytes

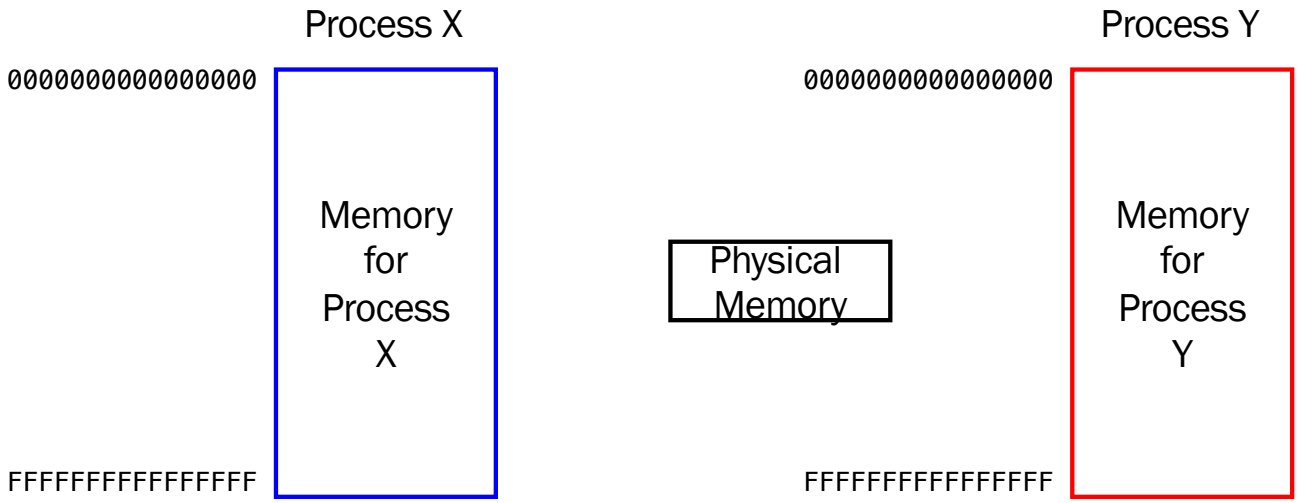
Uniform: contiguous memory locations from 0 to 2^{64}-1

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

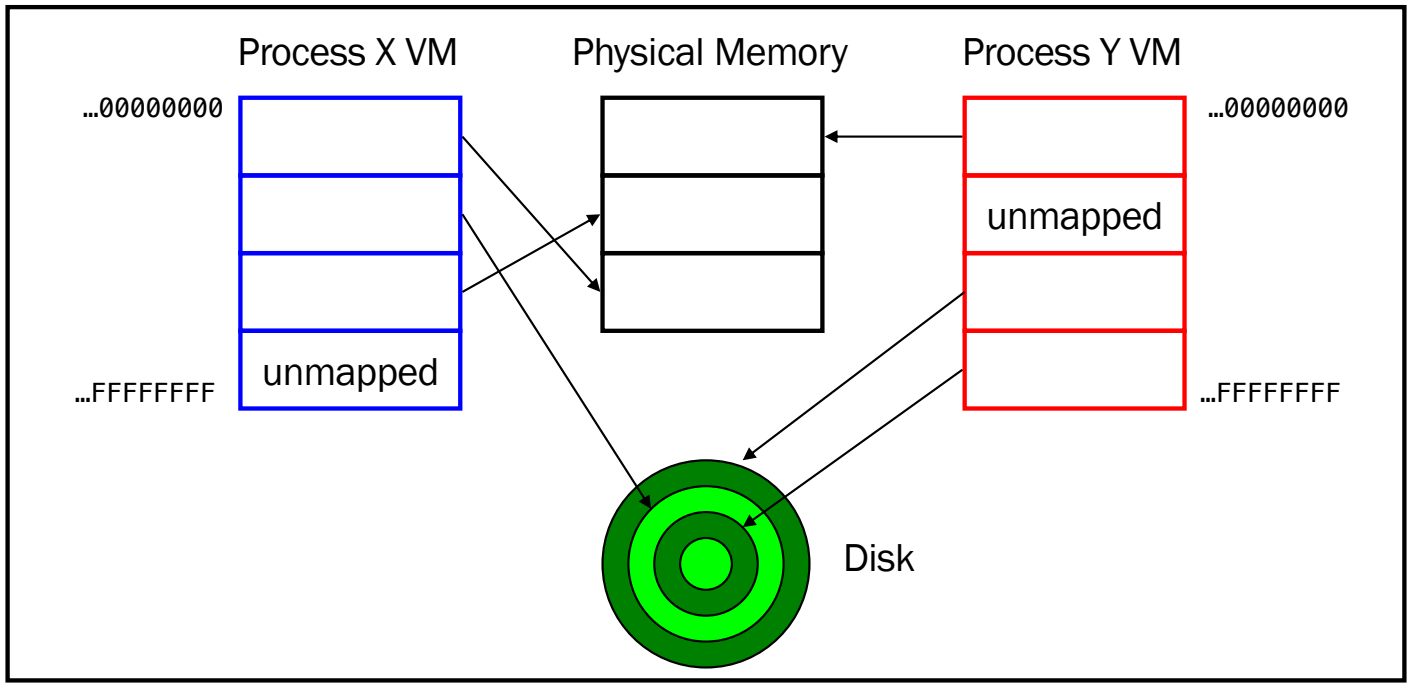
OS and hardware make sure data that are not in physical memory are kept on disk

- "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)

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

How Virtual Memory is Implemented



Memory is divided into pages

- At any time, some pages are in physical memory, some on disk
- OS and hardware swap pages between physical memory and disk as needed
- Multiple processes share the machine's physical memory

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**

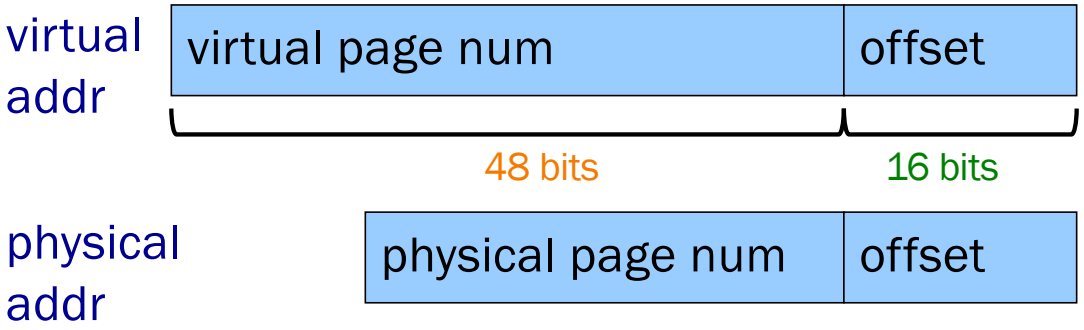
Physical address

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

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

- The offset is the same in virtual address and corresponding physical address

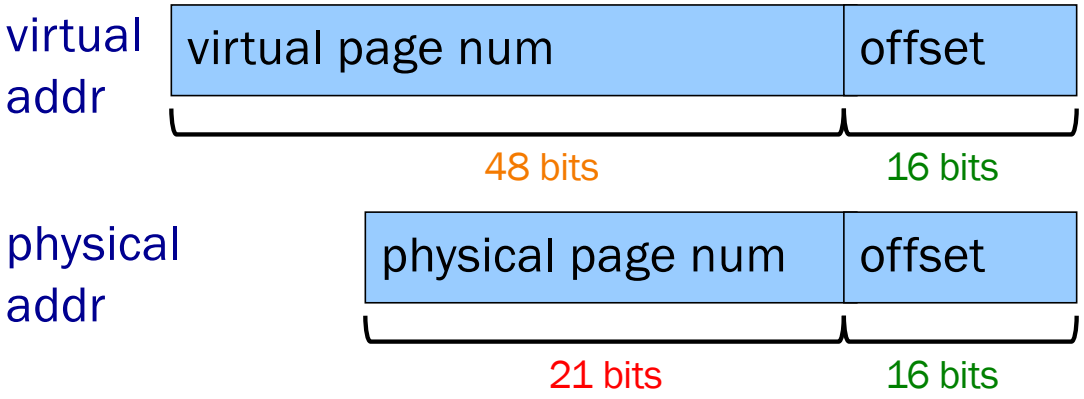
ArmLab Virtual & Physical Addresses



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)

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

Page tables for active processes are in main memory

Storing Page Tables

Page tables (mappings) are stored in memory

Process looks up mappings there

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

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

VM Effects on Security and Speed

Q: What effect does virtual memory have on the security and speed of processes?

	Security	Speed
A.	↑	↑
B.	↓	↑
C.	↑	↓
D.	↓	↓

Let's start by considering security…

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

VM Effects on Security and Speed

Q: What effect does virtual memory have on the security and speed of processes?

	Security	Speed
A.	↑	↑
B.	↓	↑
C.	↑	↓
D.	↓	↓

OK, so part of the answer is:

Security

↑

But what about speed?

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 …

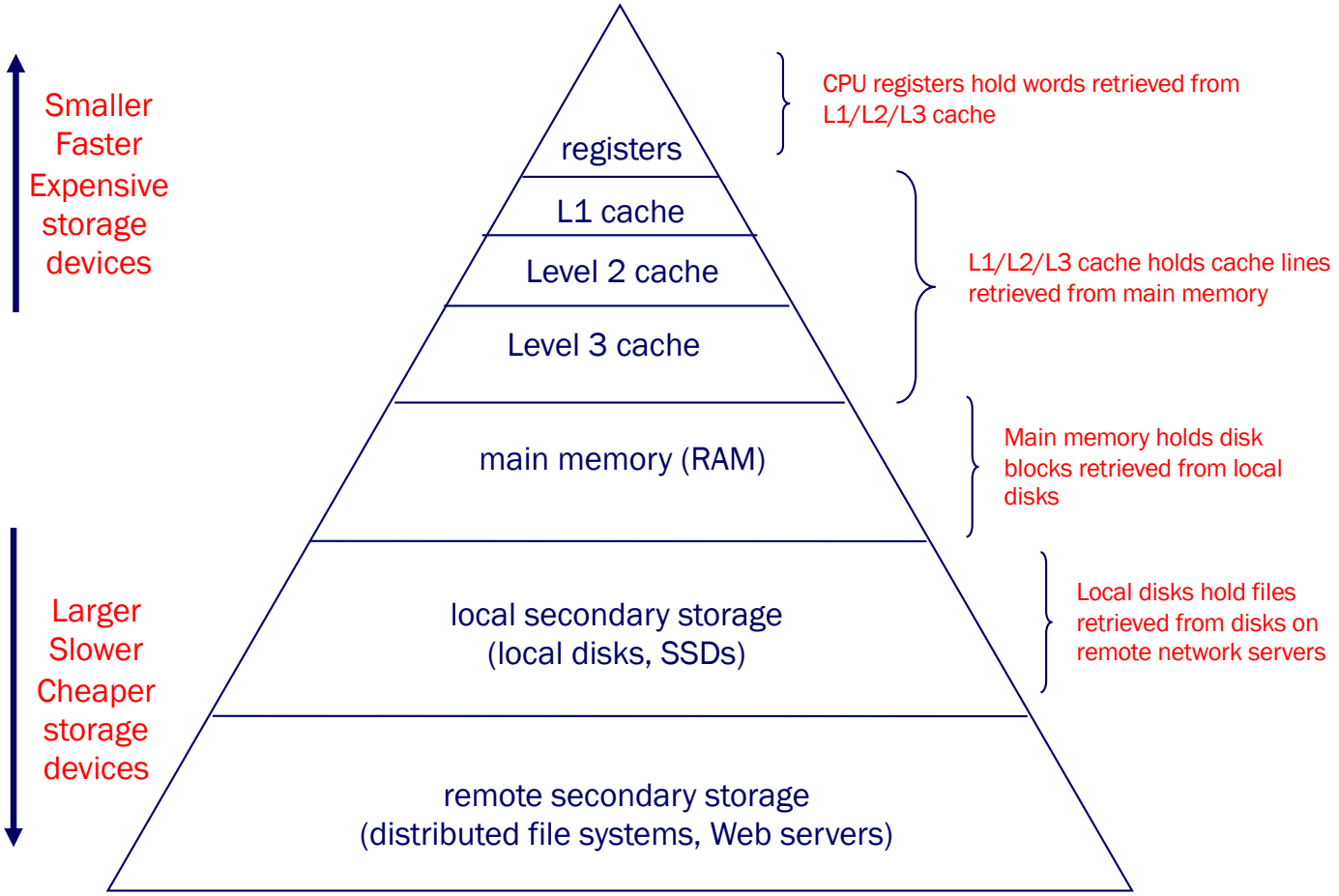
Agenda

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

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

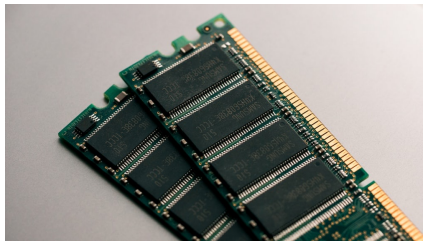
- **Latency:** 0 cycles
- **Capacity:** 8-256 registers (31 8-byte general purpose registers in AArch64)

L1/L2/L3 Cache

- **Latency:** 1 to 40 cycles
- **Capacity:** 32KB to 32MB

Main memory (RAM)

- **Latency:** ~ 50-100 cycles
  - 100 times slower than registers
- **Capacity:** 10s of GB, can be larger



Typical Storage Hierarchy

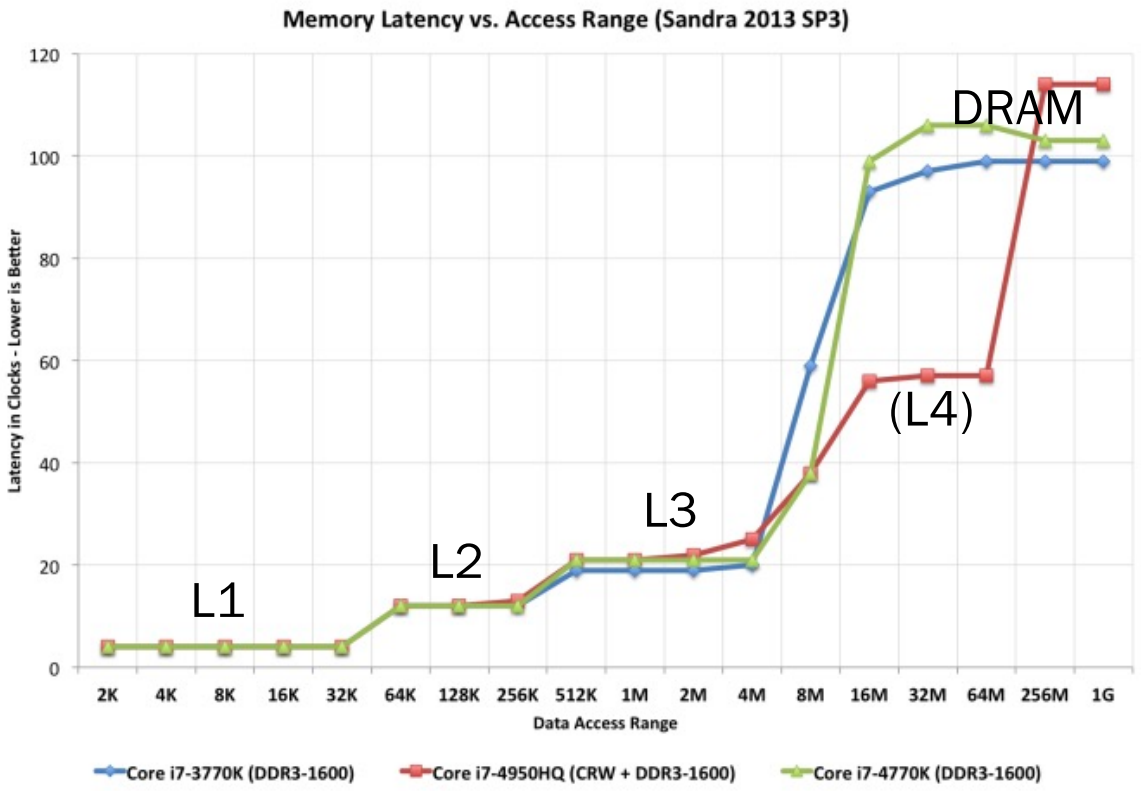
Local secondary storage: disk drives

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

- Hard Disk:
  - 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

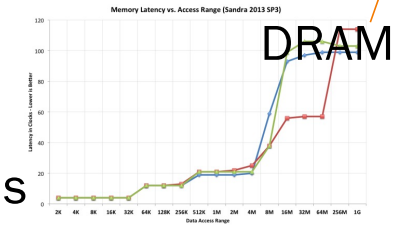
HDD

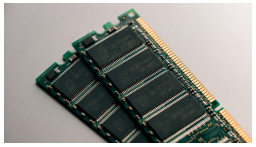
1 ms

SSD

1 µs

DRAM





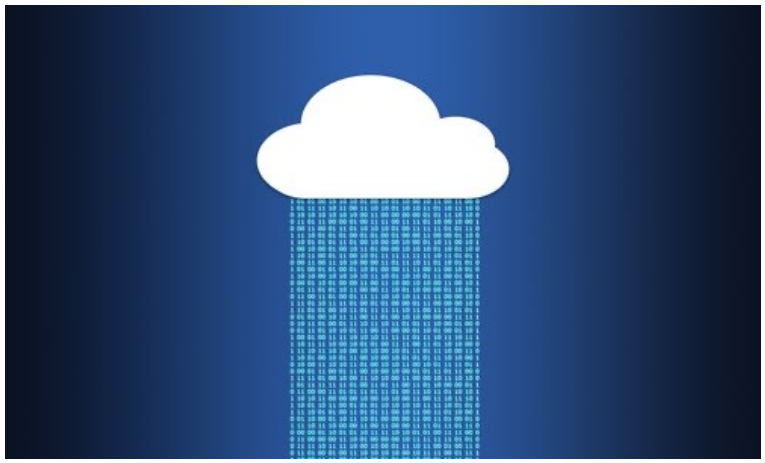
1 ns

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)

The illusion we want:

- 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

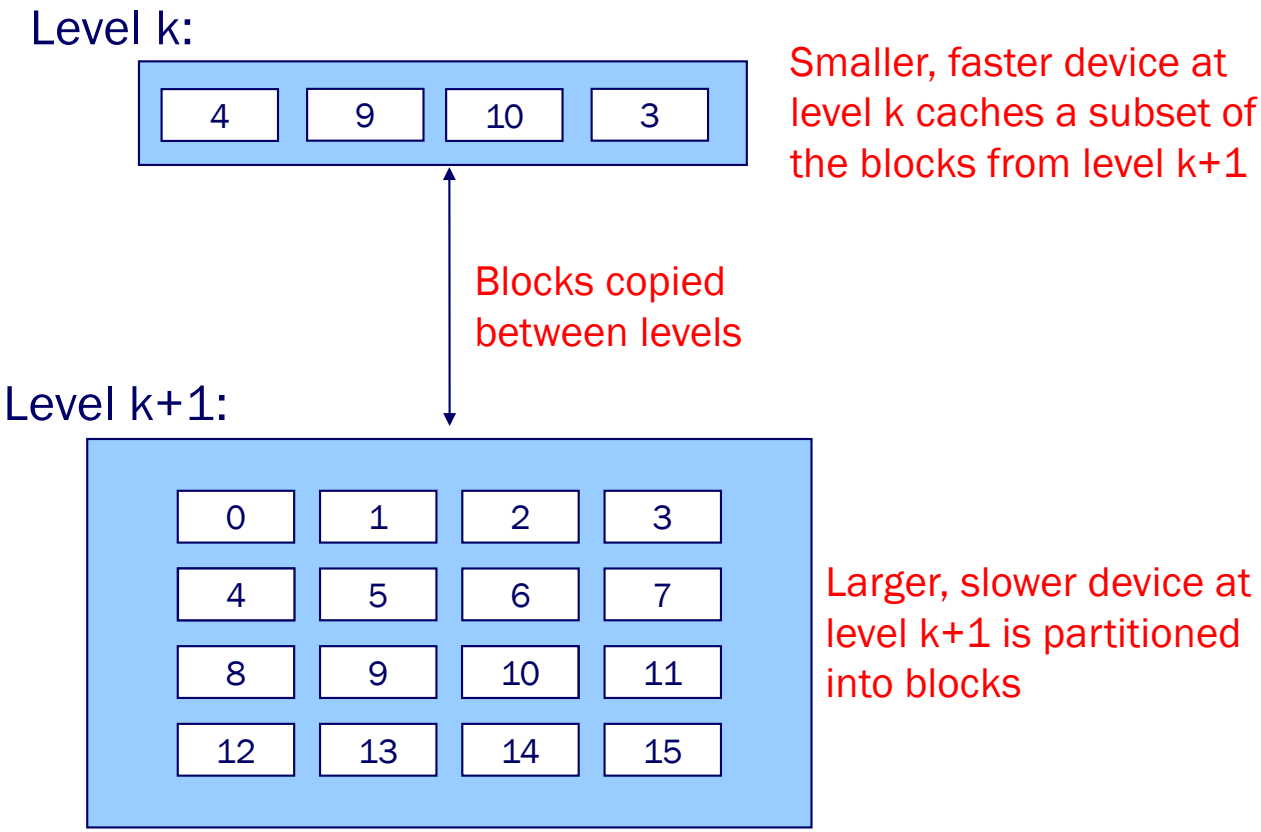
- Fast access, small capacity storage device
- Acts as a staging area for items in a slow access, large capacity storage device
- Items are brought in when they or nearby items are accessed, and kicked out when needed due to limited cache capacity

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

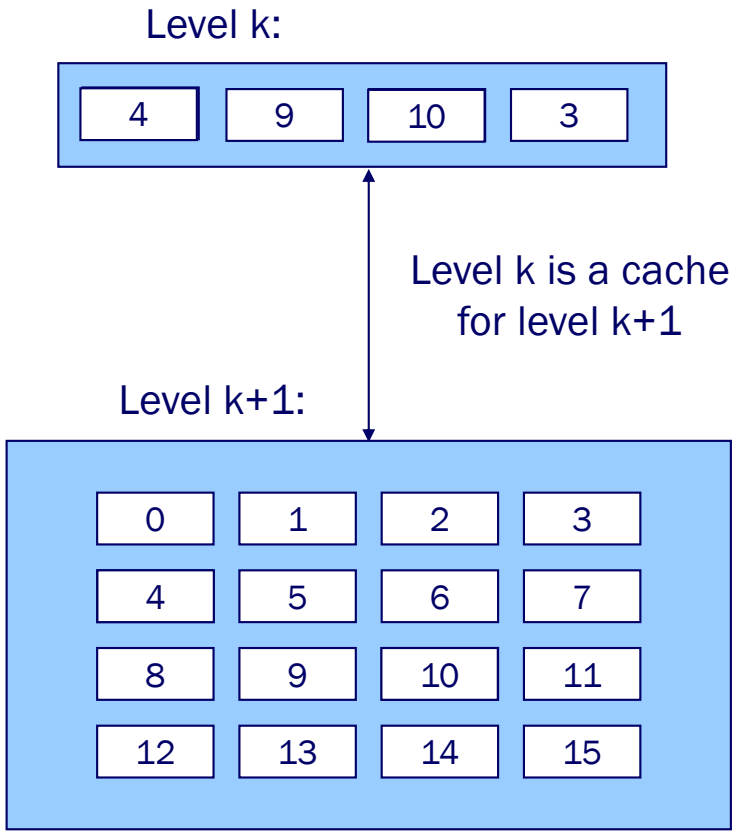
- E.g., request for block 10
- Access block 10 at level k
- Fast

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



VM Effects on Security and Speed

Q: What effect does virtual memory have on the security and speed of processes?

	Security	Speed
A.	↑	↑
B.	↓	↑
C.	↑	↓
D.	↓	↓

So, with caching, we *finally* arrive at the answer:

Security	Speed
↑	not so much

Do Exam Questions Exhibit Temporal Locality?

Here's a real question from an old exam:

For caching in a memory hierarchy, what is the best motivation for a *larger* cache block size?

A. Temporal Locality

B. Spatial Locality

C. Both

D. Neither

B

Spatial locality makes use of subsequent data after a given read, so having more data to keep reading is a win.

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

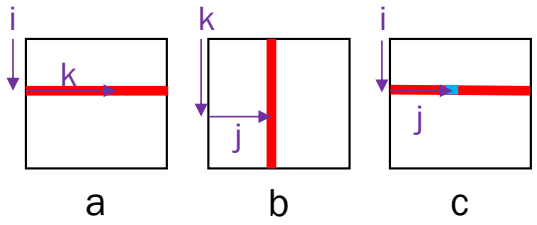
- Access in row order ⇒ good spatial locality
- Access in column order ⇒ poor spatial locality

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 locality for A
- Bad locality for B
- Good locality for C

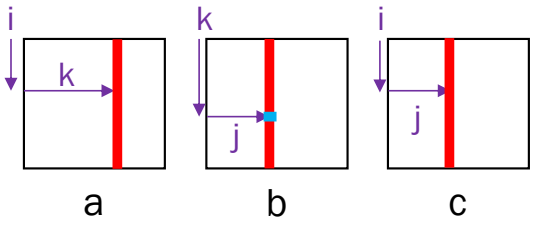


Spatial Locality Example: Matrix Multiplication

```
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 locality for A
- Bad locality for B
- Bad locality for C

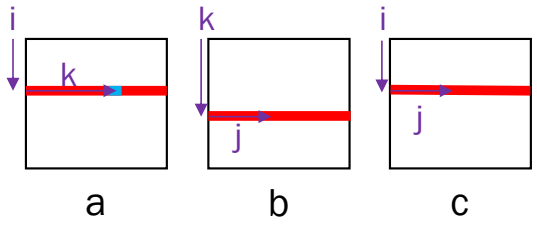


Spatial Locality Example: Matrix Multiplication

```
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 locality for A
- Good locality for B
- Good locality for C



Another Ghost of Exams Past …

Suppose that C laid out arrays in column-major order instead of row-major order. What would be the *most efficient* loop ordering for matrix multiplication to maximize performance through good locality?

```
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];
```

A. i k j (Same as row-major)

B. i j k

C. j k i

D. j i k

E. k i j

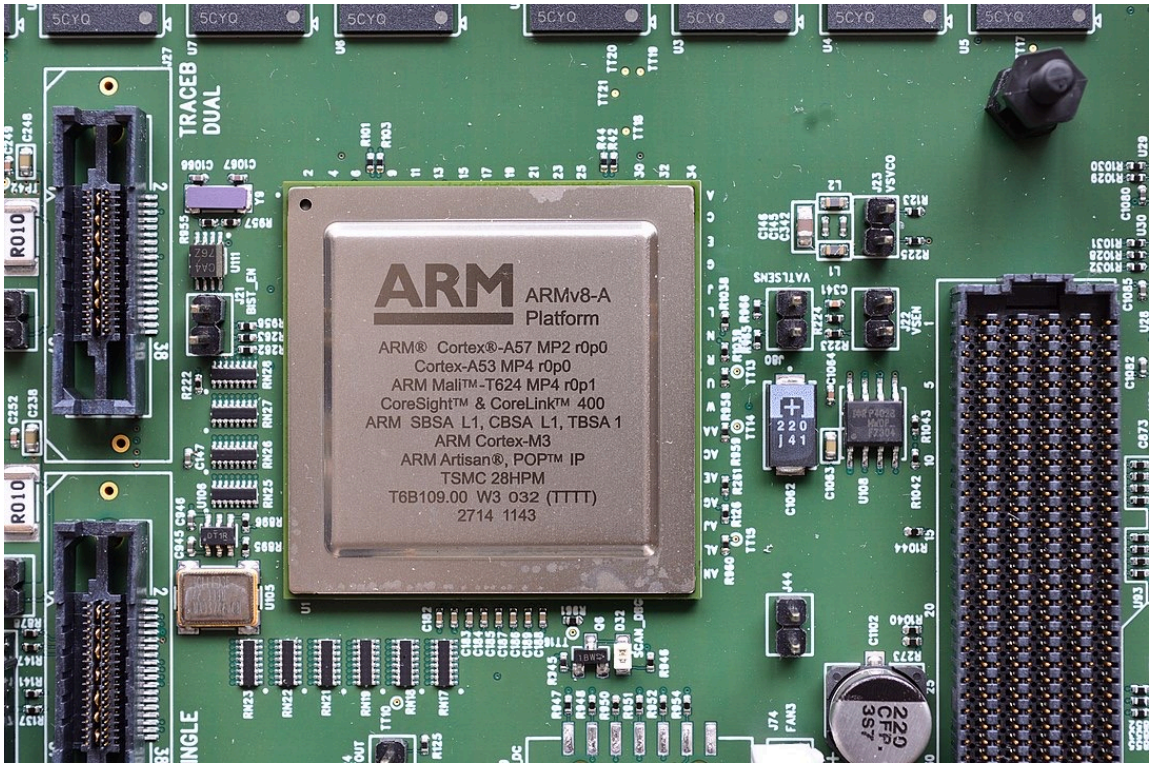
F. k j i

C: j k i

Exactly what makes this bad for all three in row-major makes it ideal for column-major:
a and c have good spatial
b has good temporal, spatial

Next time …

Getting started with ARM assembly language



Lobsterthermidor , Raysonho