Systems and Internet Infrastructure Security

Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

# Unix and Operating Systems (Part 2)

Devin J. Pohly <djpohly@cse.psu.edu>

# Introduction to Systems...

- Crossover knowledge from CMPEN 331 and CMPSC 473.
- Last day of dry background material for now.
- The programming and cool Unix stuff is coming, I promise!

# Types of storage

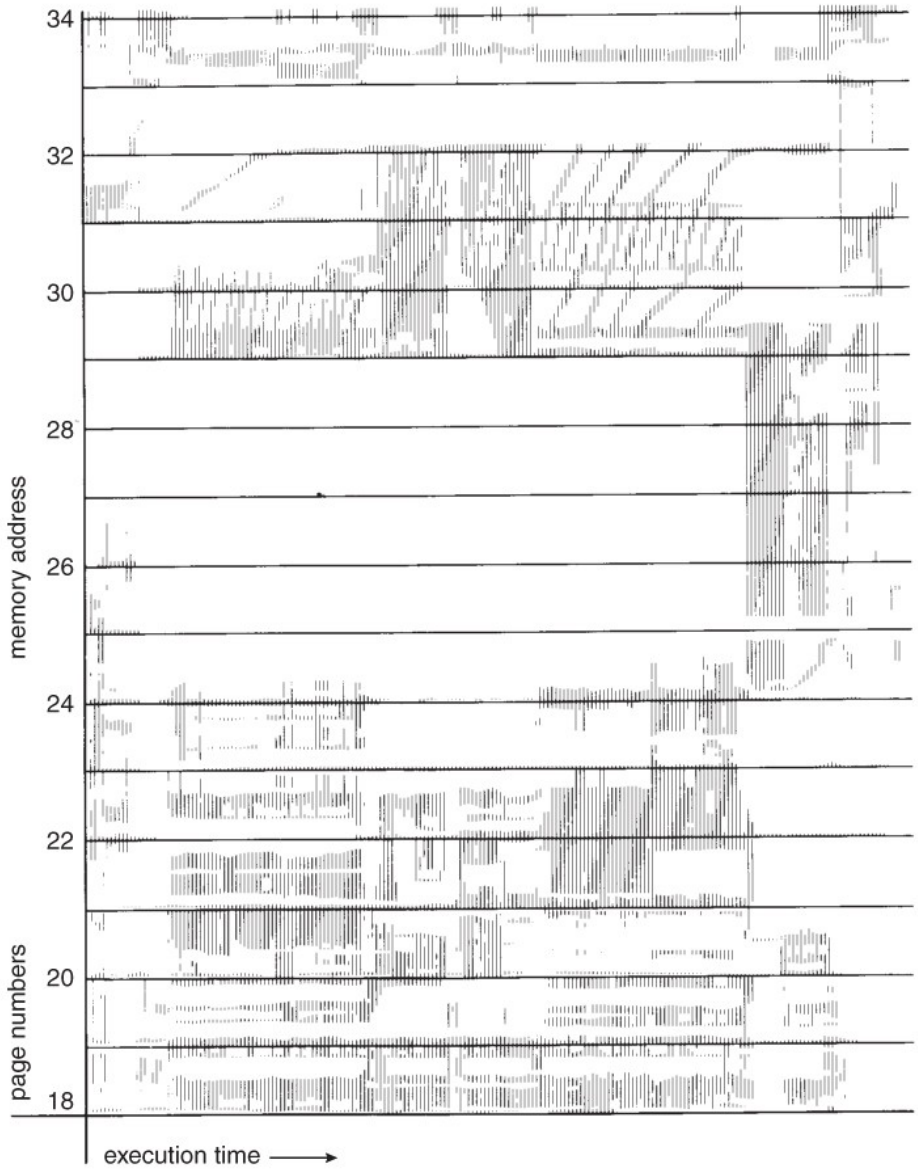
- Example: HDD vs. SSD
  - What practical differences are there?
- How about RAM?
- Google Drive?

# A few observations

- Some fundamental and/or enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (hotter!).
  - The gap between CPU and main memory speed is widening.
  - And something called *locality*...

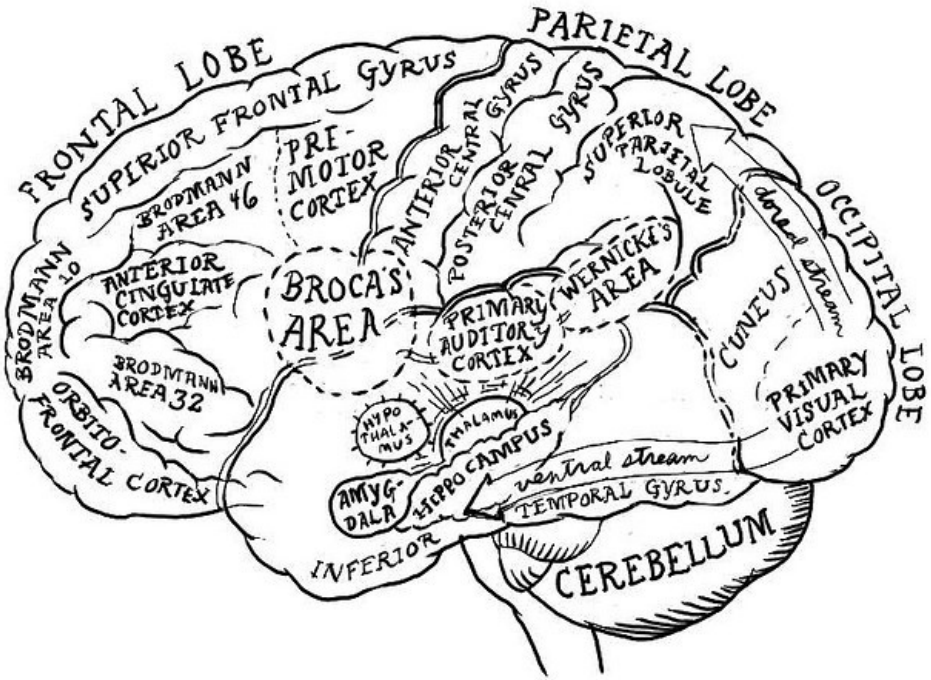
# Locality

- Temporal locality
  - Memory recently referenced is likely to be referenced again soon...
- Spatial locality
  - … and so is the memory around it.
- Well-written programs tend to exhibit good locality.



# Memory hierarchies

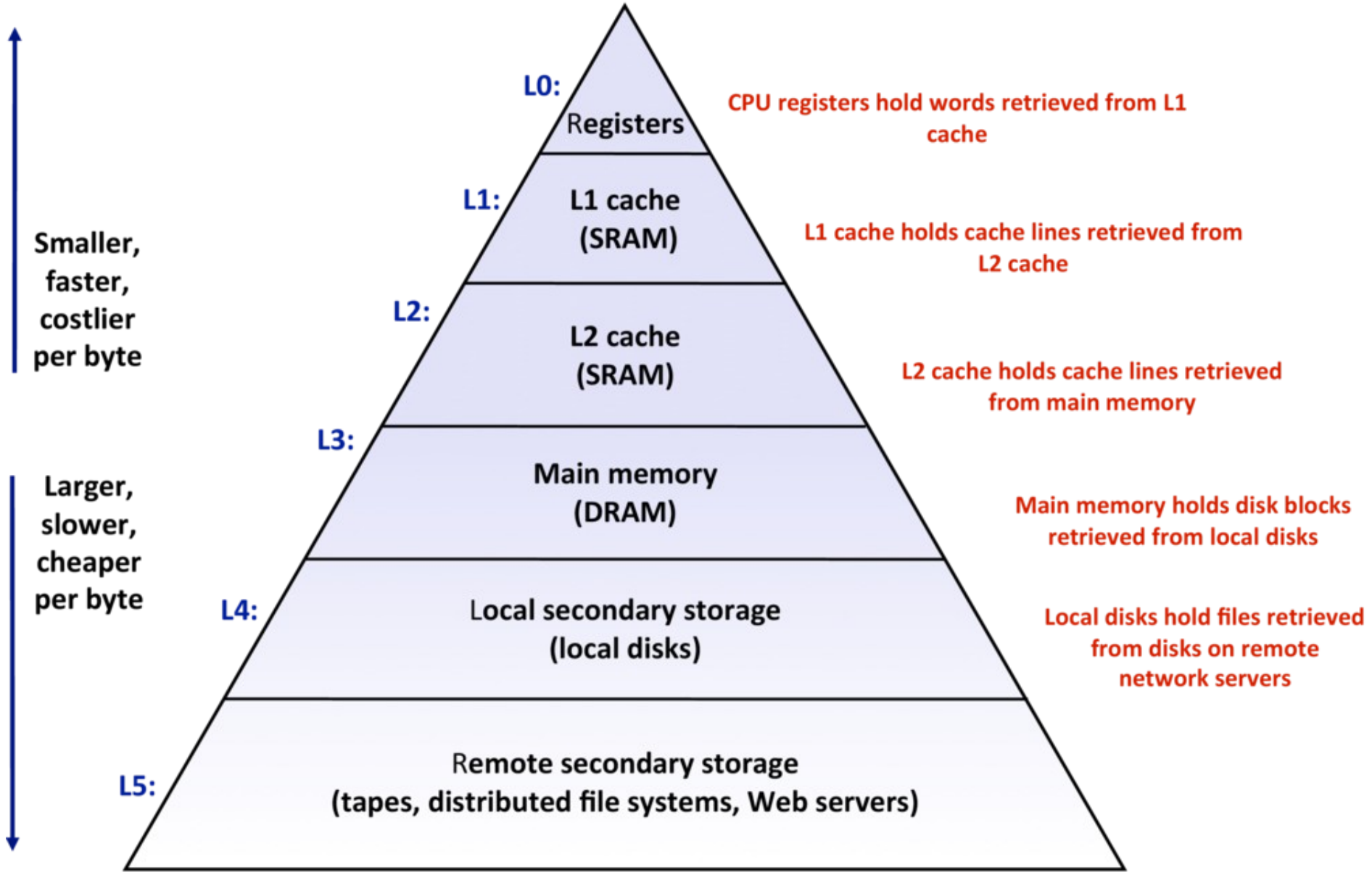
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.



# Caching

- *Cache*
  - a smaller, faster storage device that stores recently accessed data from a larger, slower device
- Fundamental concept of a memory hierarchy:
  - For each *k*, the faster, smaller device at level *k* serves as a cache for the larger, slower device at level *k*+1

# Example memory hierarchy

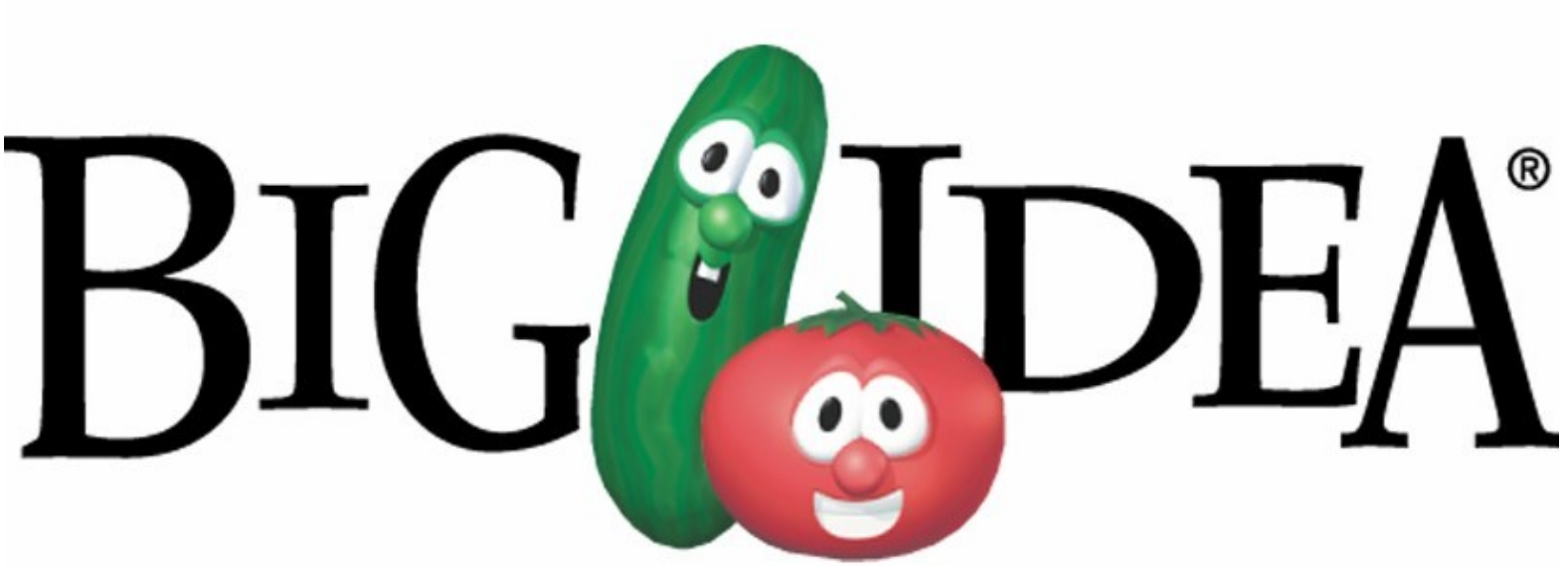


# How do they work?

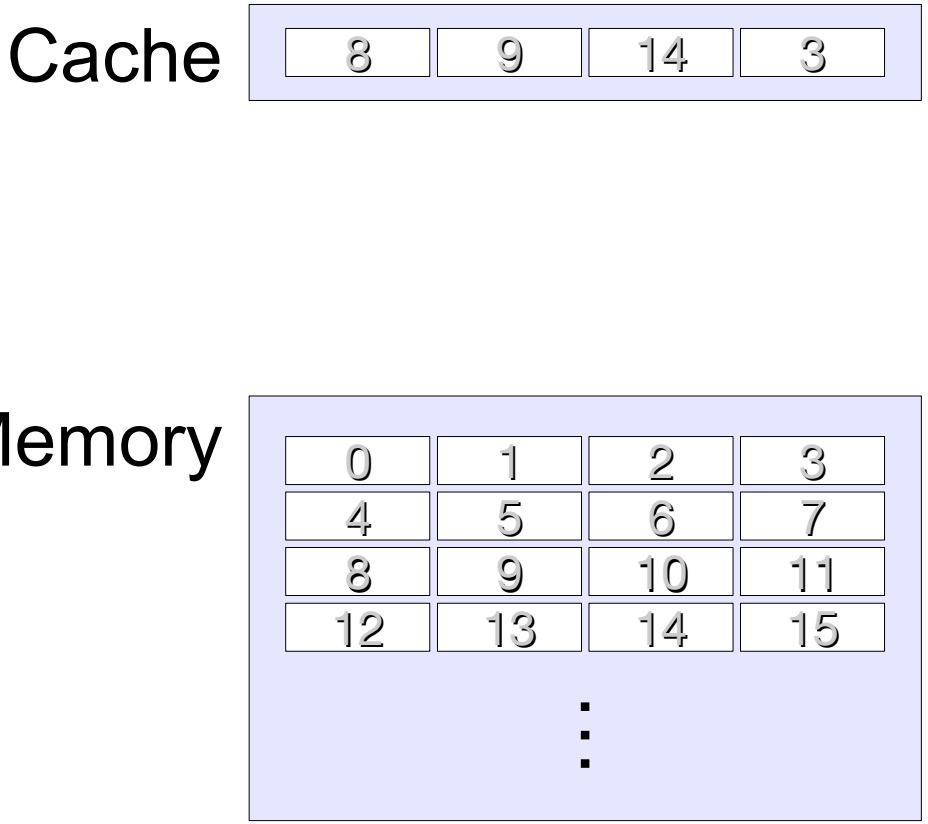
- Locality
  - Programs tend to access the data at level *k* more often than they access the data at level *k*+1.
- Thus, the storage at level *k*+1 can be slower, and thus larger and cheaper per byte.

# The Big Idea

- The memory hierarchy creates a large pool of storage that costs about as much as the cheap storage near the bottom, but that serves data to programs about as fast as the fast storage near the top.



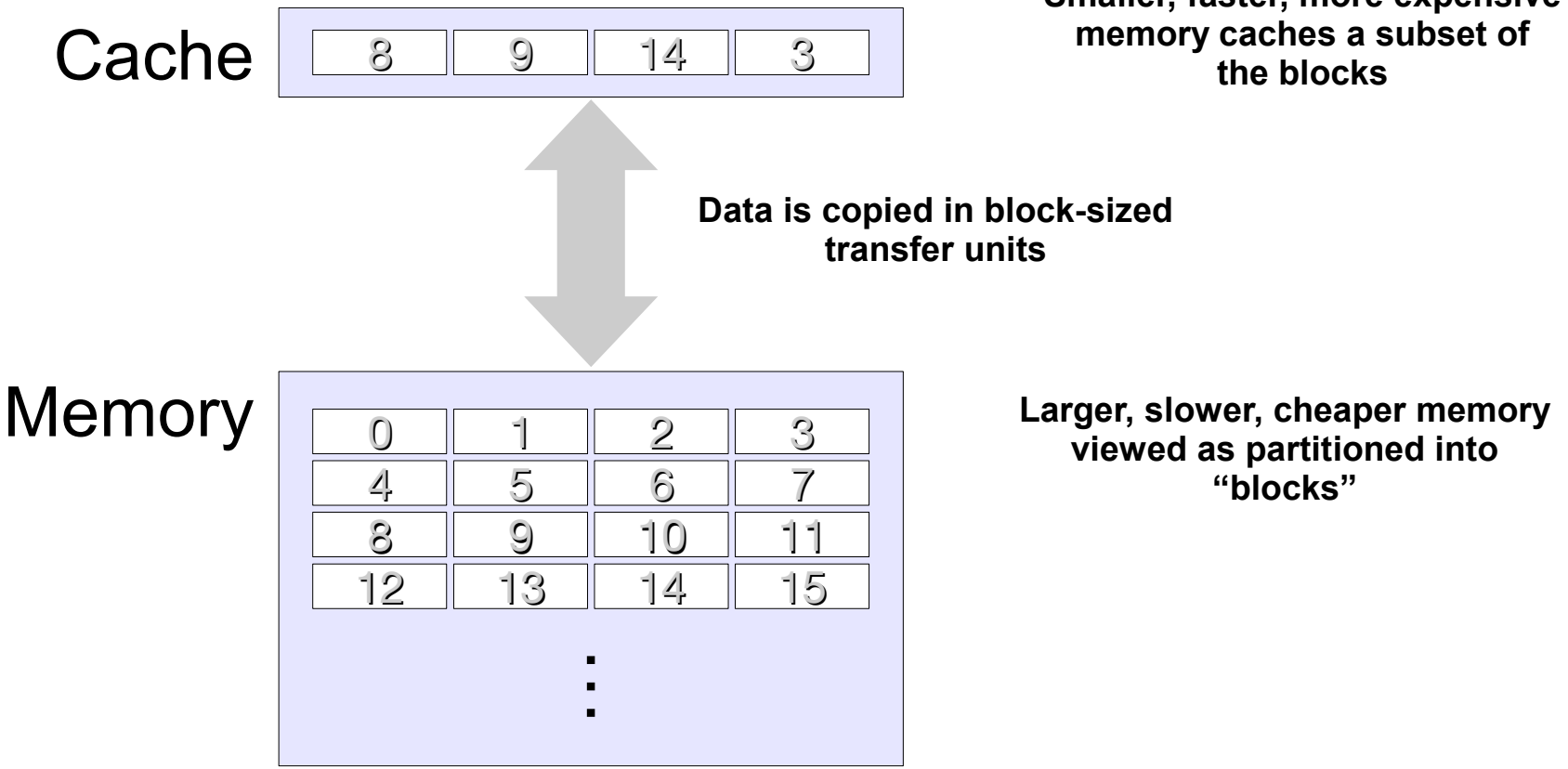
# Caching example



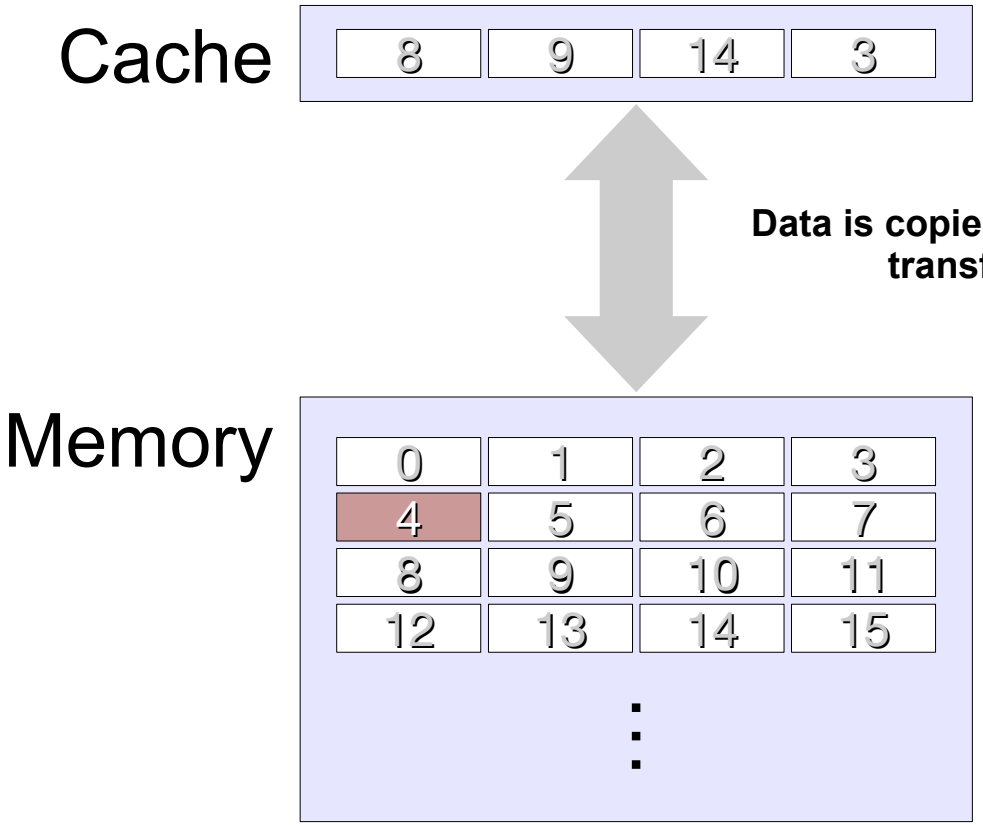

**Smaller, faster, more expensive memory caches a subset of the blocks**

**Larger, slower, cheaper memory viewed as partitioned into "blocks"**

# Caching example



# Caching example

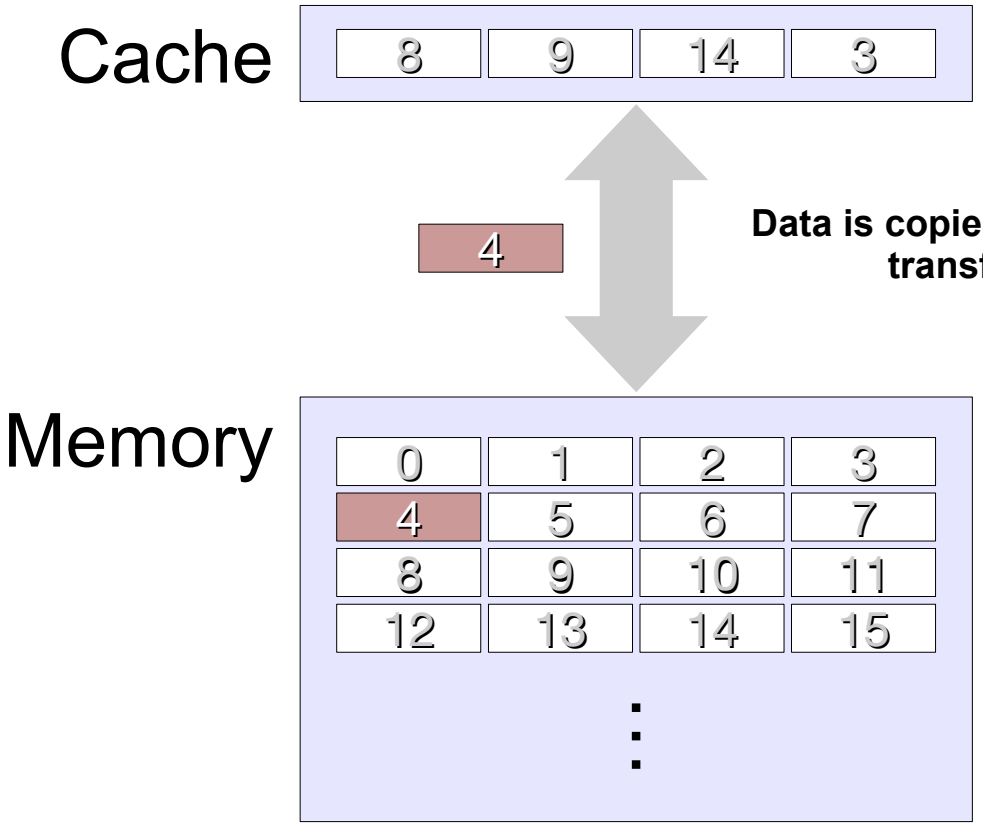



**Smaller, faster, more expensive memory caches a subset of the blocks**

**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into "blocks"**

# Caching example

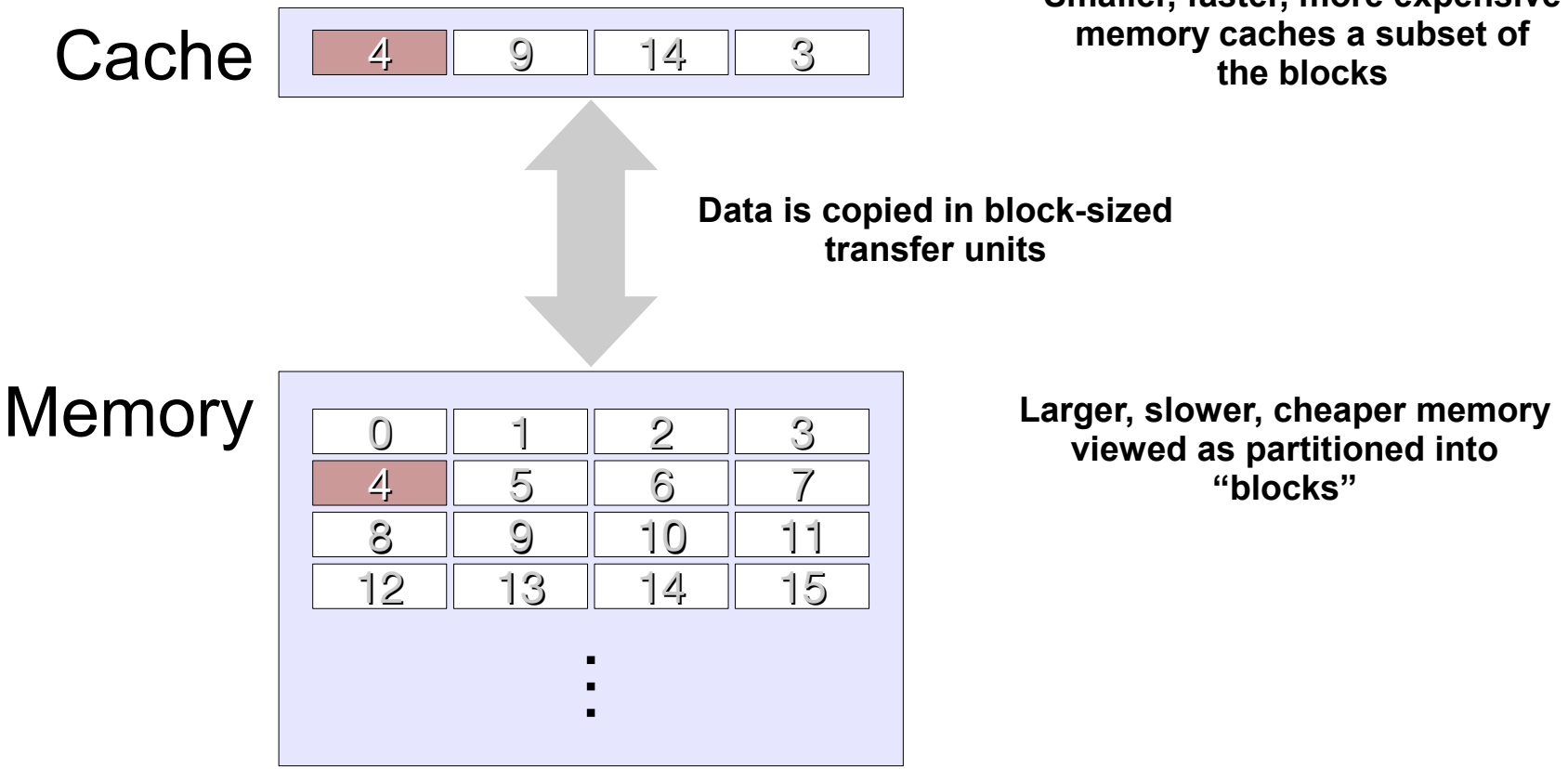



**Smaller, faster, more expensive memory caches a subset of the blocks**

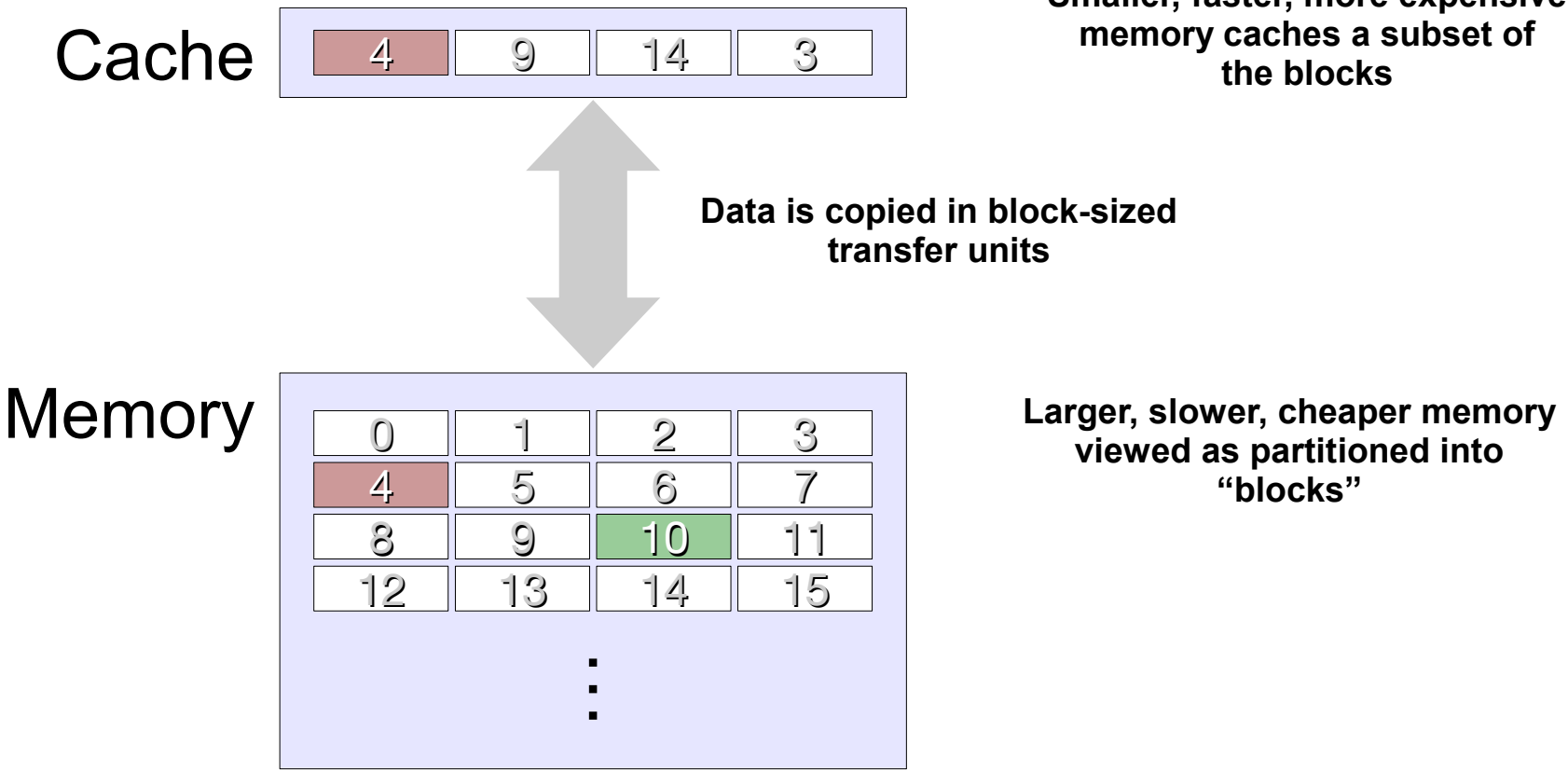
**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into "blocks"**

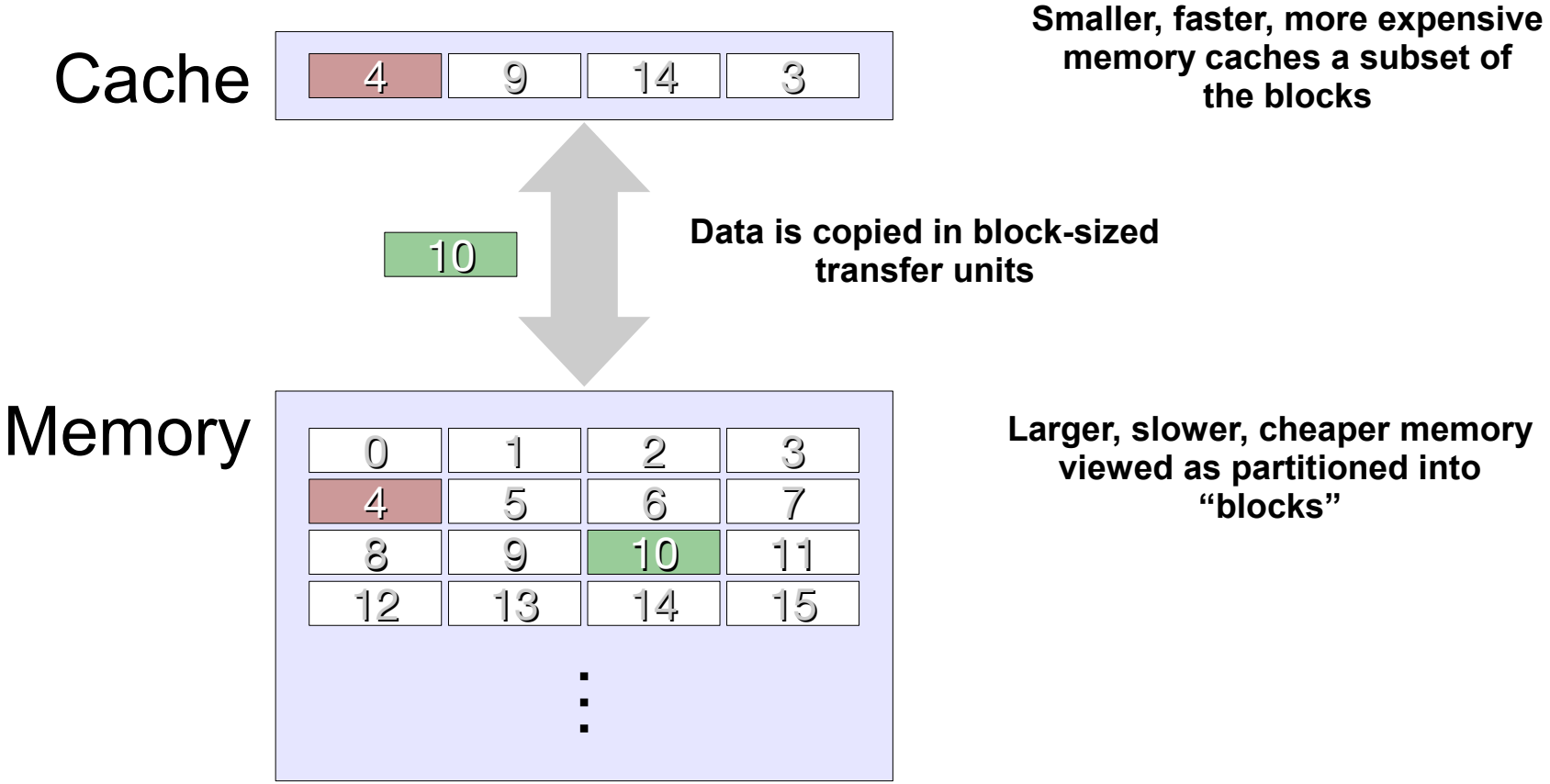
# Caching example



# Caching example



# Caching example

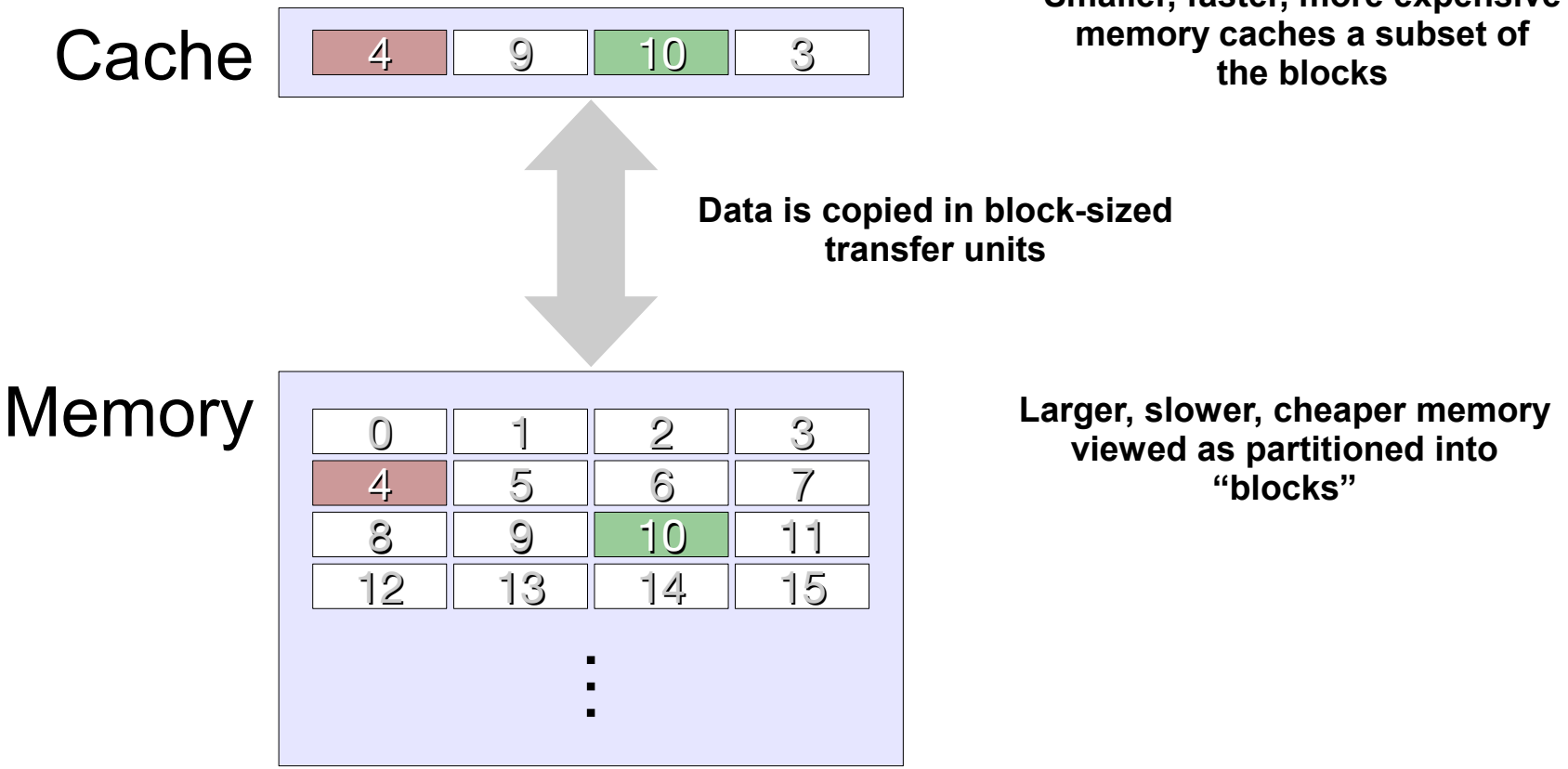



**Smaller, faster, more expensive memory caches a subset of the blocks**

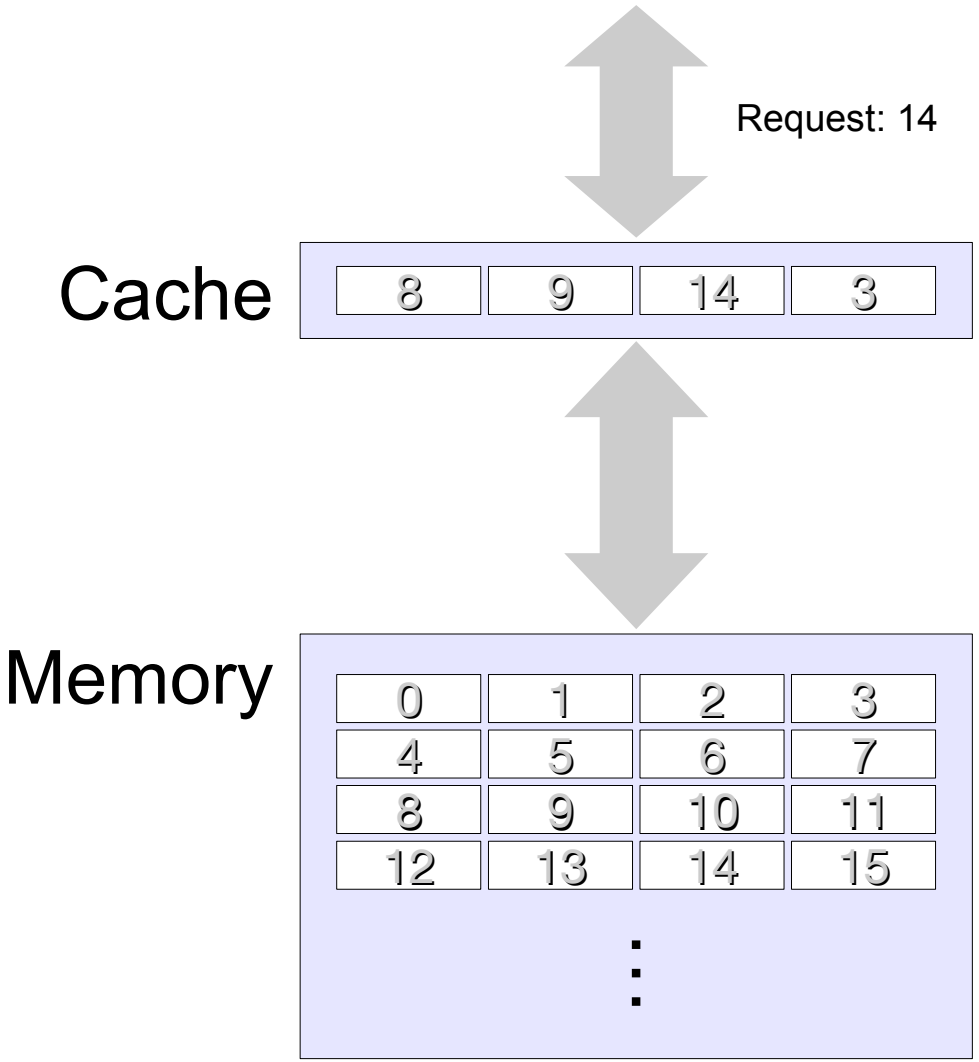
**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into "blocks"**

# Caching example

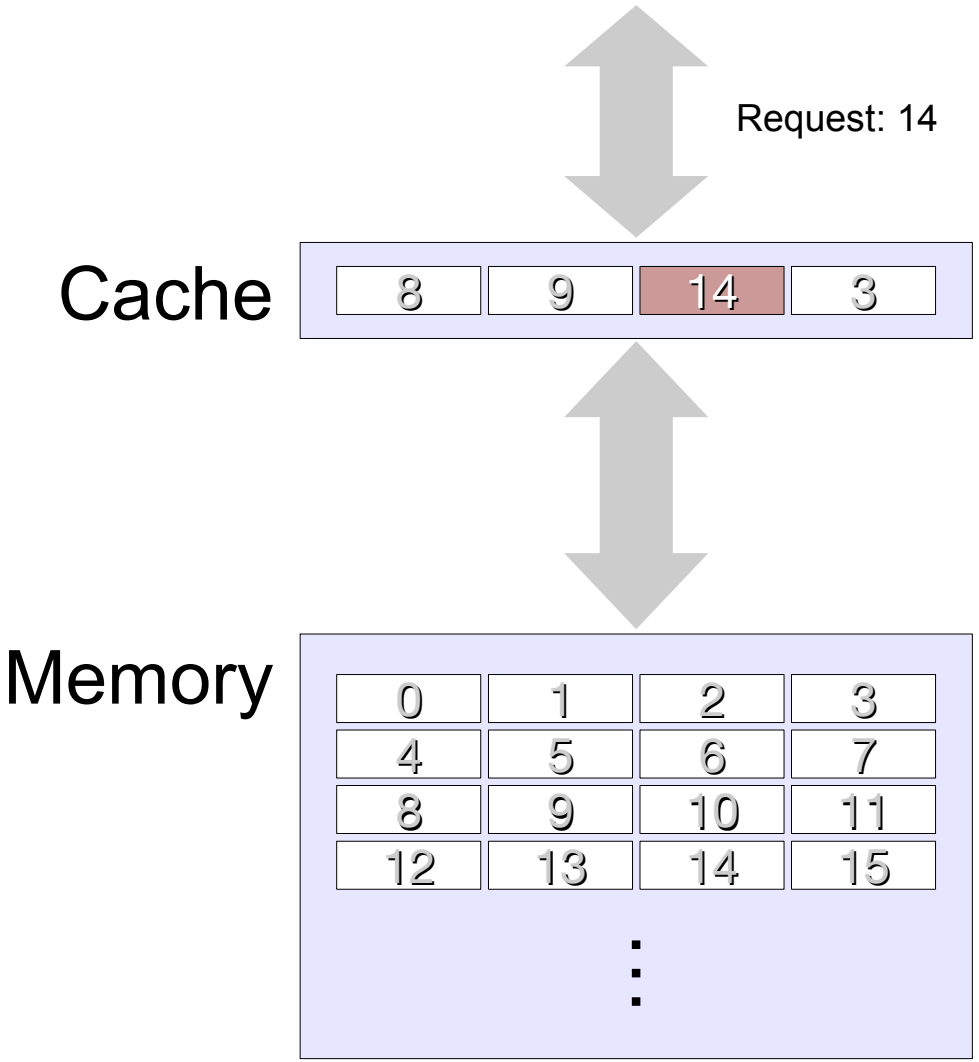


# It's a hit!




**Data in block *b* is needed**

# It's a hit!

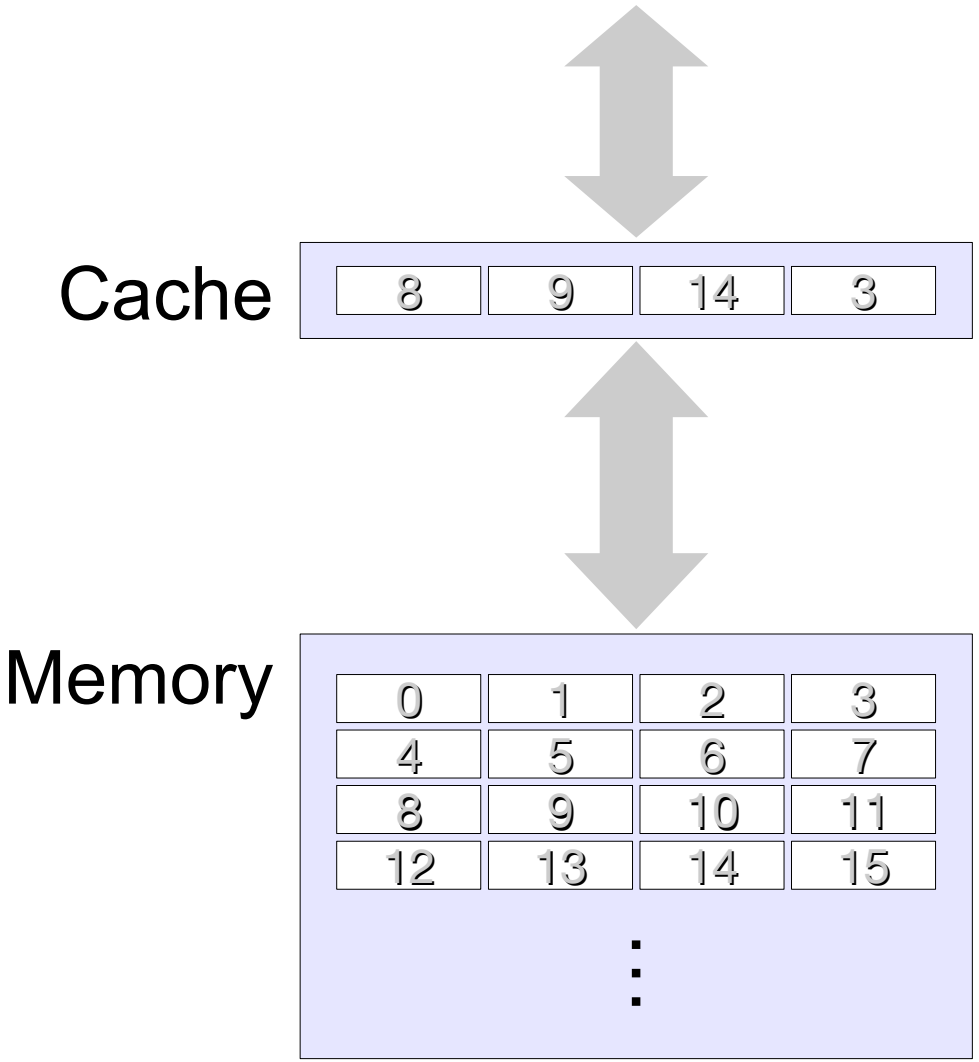



**Data in block *b* is needed**

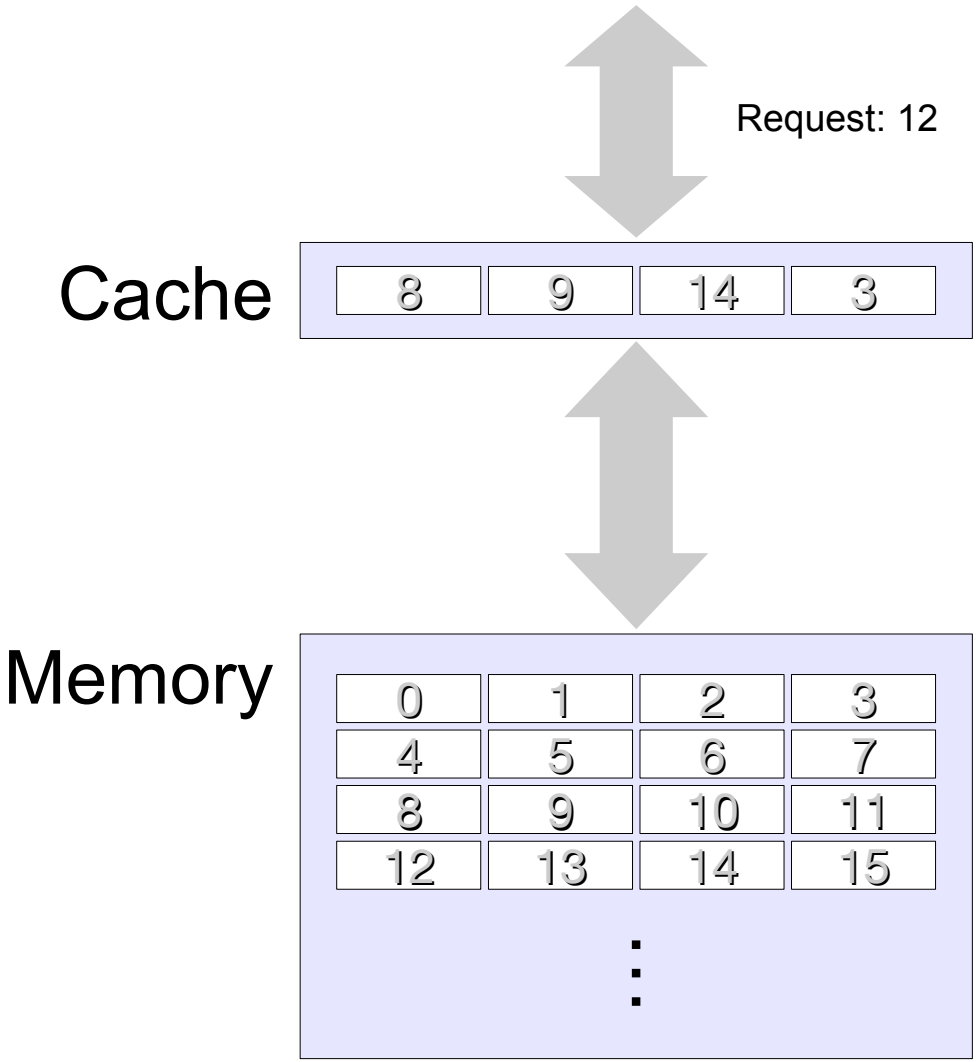
**Block *b* is in cache:**

***Hit!***

# Swing and a miss

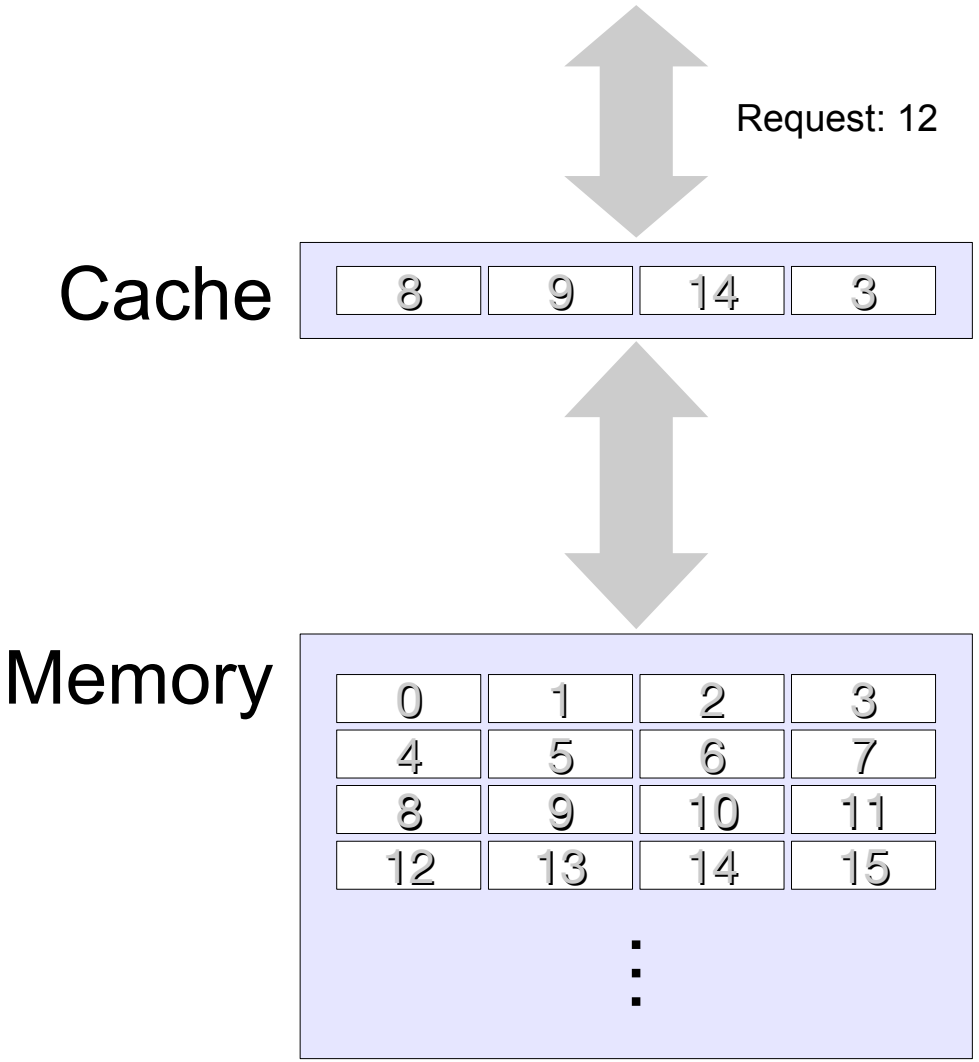


# Swing and a miss




**Data in block *b* is needed**

# Swing and a miss

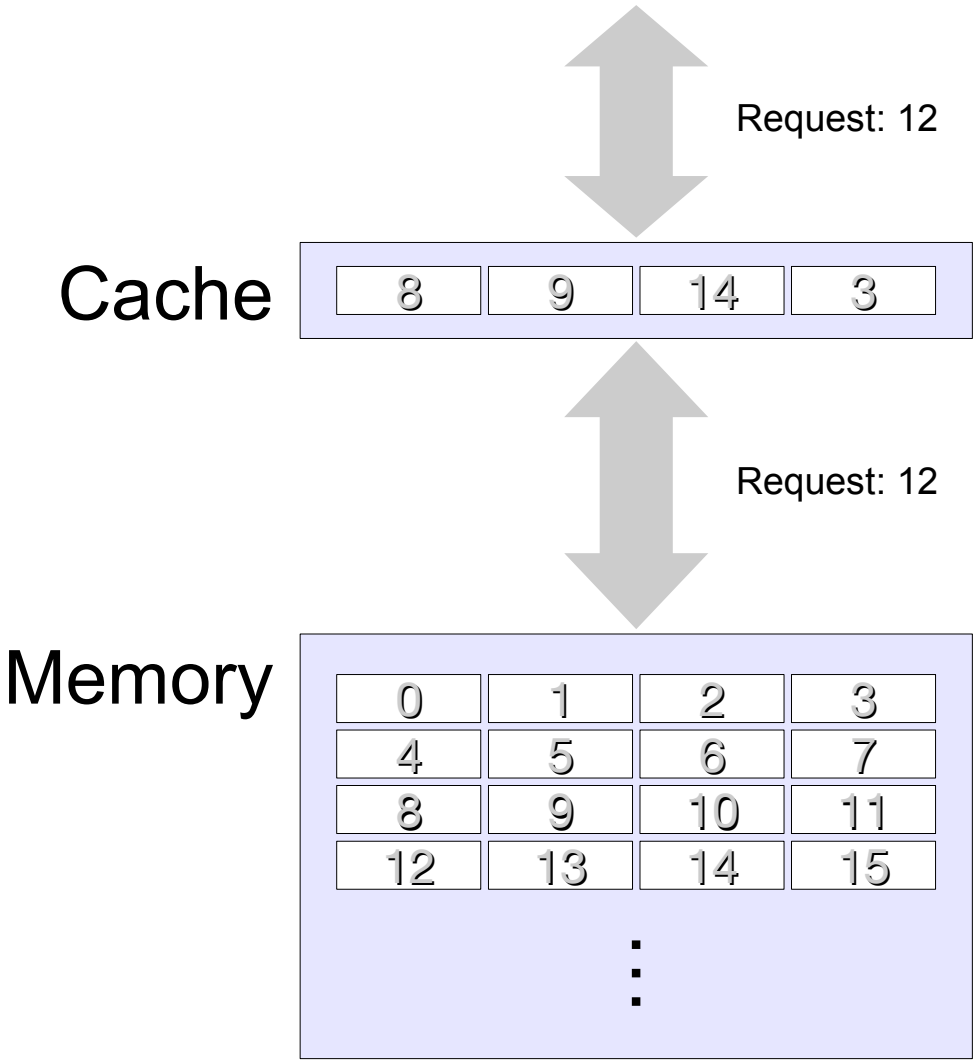



**Data in block *b* is needed**

**Block *b* is not in cache:**

***Miss!***

# Swing and a miss



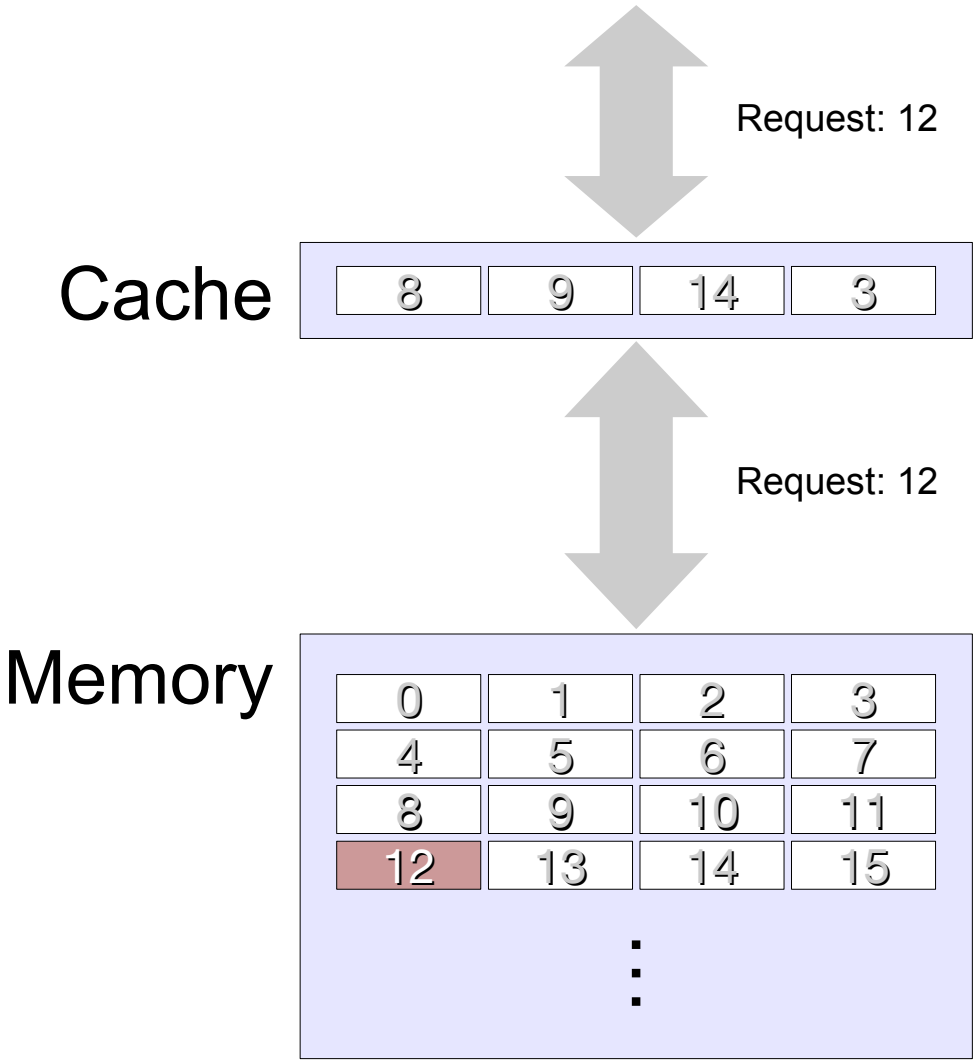

**Data in block *b* is needed**

**Block *b* is not in cache:**

***Miss!***

**Block *b* is fetched from memory.**

# Swing and a miss



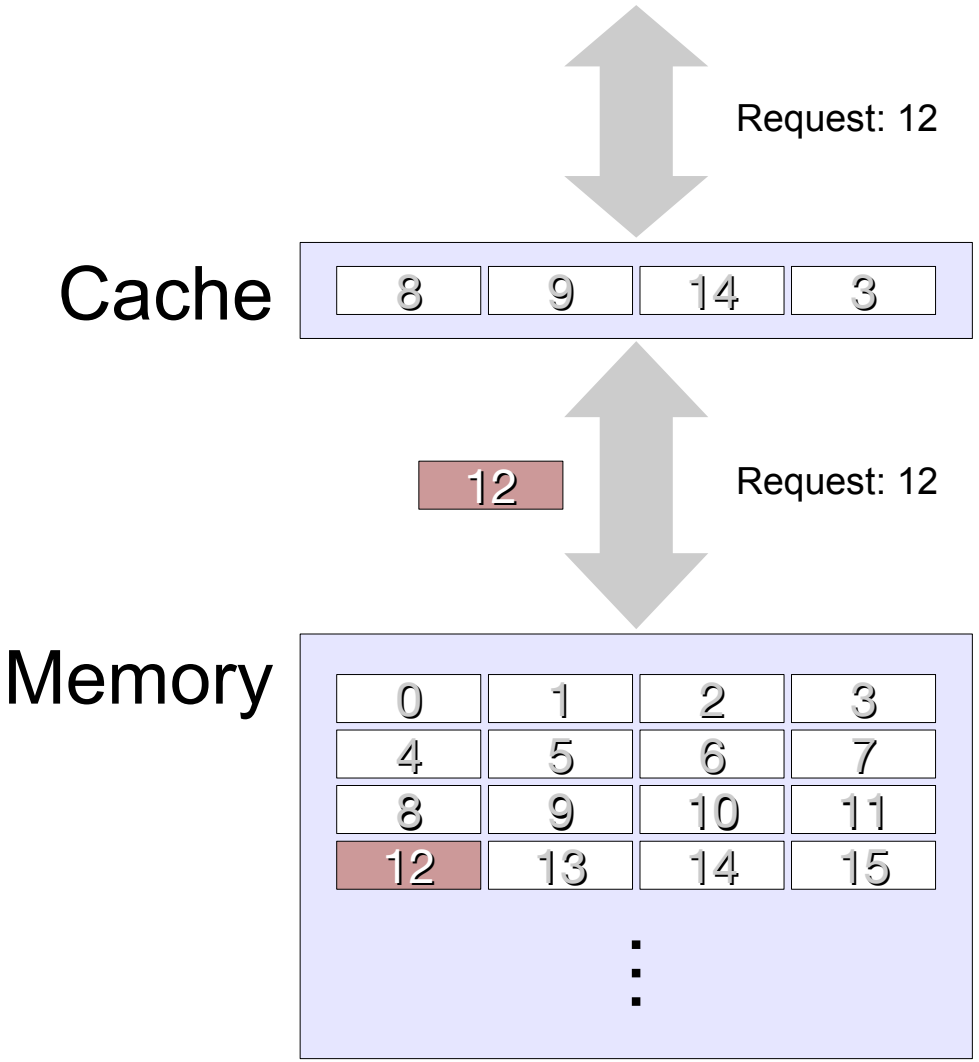

**Data in block *b* is needed**

**Block *b* is not in cache:**

***Miss!***

**Block *b* is fetched from memory.**

# Swing and a miss



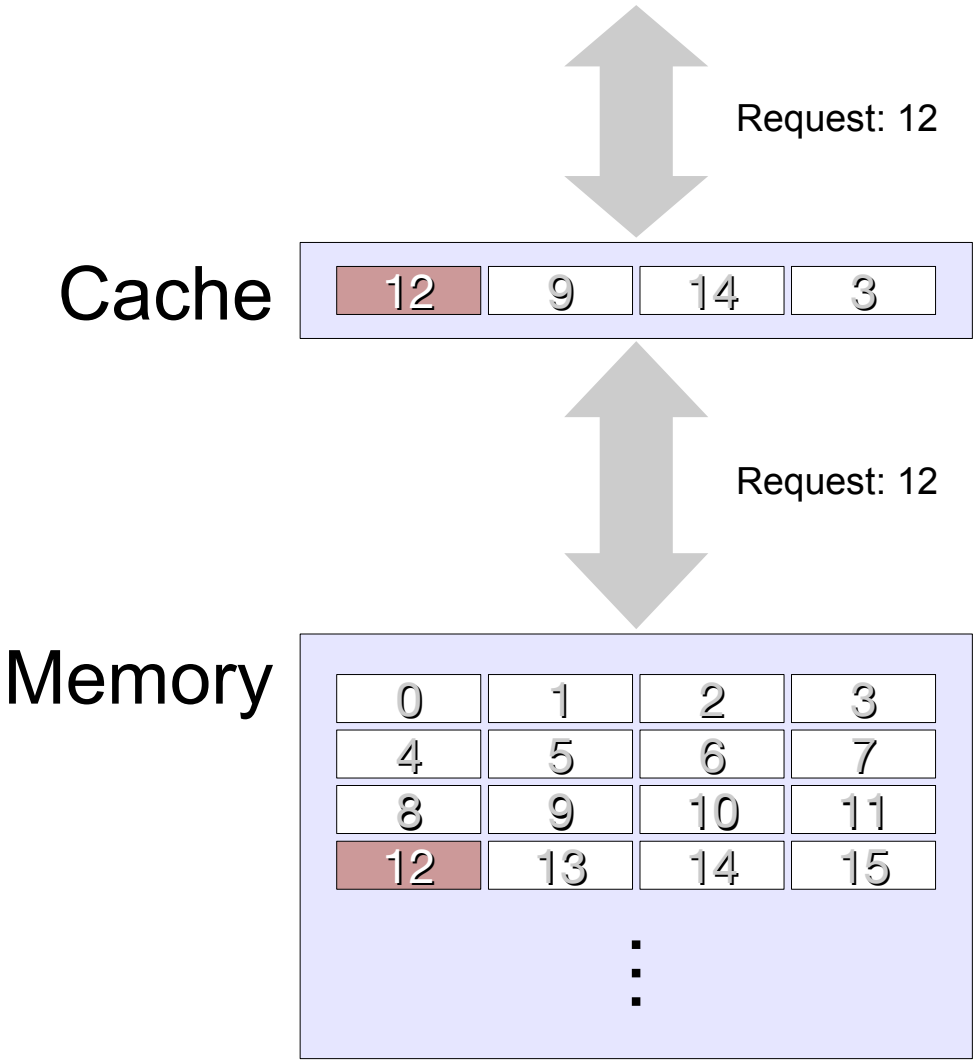

**Data in block *b* is needed**

**Block *b* is not in cache:**

***Miss!***

**Block *b* is fetched from memory.**

# Swing and a miss




**Data in block *b* is needed**

**Block *b* is not in cache:**

***Miss!***

**Block *b* is fetched from memory.**

**Block *b* is stored in cache**

Placement policy:
determines where *b* goes

Replacement policy:
determines which "victim" block gets evicted

# Reasons for cache misses

- Cold (compulsory) miss
  - Occurs because the cache is empty
- Cache is "cold" when it has just been turned on or flushed
- Cache is "hot" when it is full and likely to hit

# Reasons for cache misses

- Capacity miss
  - Occurs when the cache is not big enough to hold the working set
  - Working set: the blocks which are currently being accessed frequently

# Reasons for cache misses

- Conflict miss
  - Occurs when the cache is large enough, but the block has been evicted because of policy
- Most caches limit blocks from level *k*+1 to a subset (or just one) of the positions at level *k*.
  - Why?
  - E.g., block *i* in level *k*+1 must be placed at (*i* mod 4) in level *k*.
  - In this case, referencing blocks 0, 8, 0, 8, … would miss every time.

# Cache miss example

Level *k*

| Offset | Data |
|---|---|
| 100 | X |
| 101 | - |
| 110 | K |

Level *k*+1

| Offset (mod 8) | Data | | |
|---|---|---|---|
| 100 | X | Y | ... |
| 101 | A | B | ... |
| 110 | G | K | ... |

# Cache miss example

Level *k*

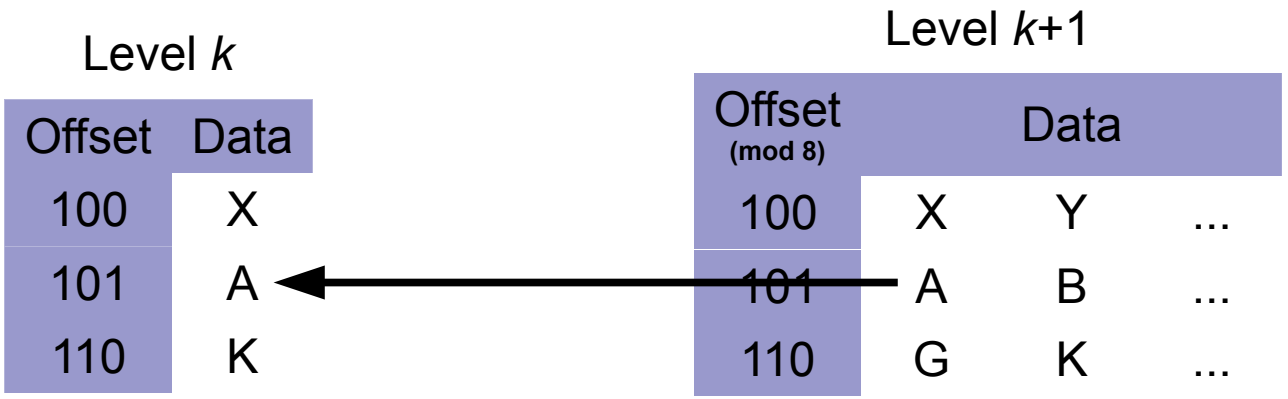
| Offset | Data |
|---|---|
| 100 | X |
| 101 | - |
| 110 | K |

Level *k*+1

| Offset (mod 8) | Data | | |
|---|---|---|---|
| 100 | X | Y | ... |
| 101 | A | B | ... |
| 110 | G | K | ... |

Request: block A

# Cache miss example




Request: block A
Cold miss, fetched from level *k*+1

# Cache miss example

Level *k*

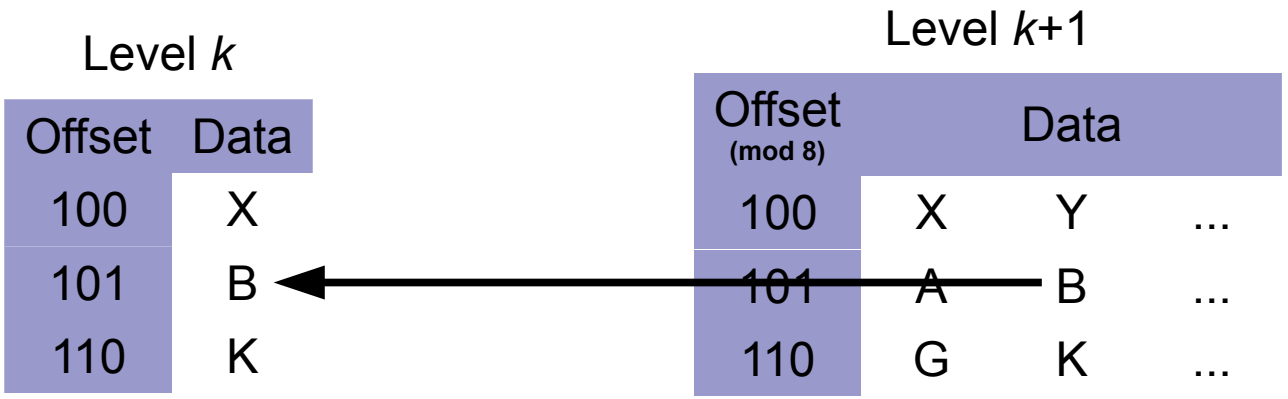
| Offset | Data |
|---|---|
| 100 | X |
| 101 | A |
| 110 | K |

Level *k*+1

| Offset (mod 8) | Data | | |
|---|---|---|---|
| 100 | X | Y | ... |
| 101 | A | B | ... |
| 110 | G | K | ... |

Request: block B

# Cache miss example




Request: block B
Conflict miss, fetched from level *k*+1

# Cache replacement

- When your cache is full and you acquire a new value, you must evict a previously stored value
  - Performance of cache is affected by how appropriate the cache replacement policy is
  - Popular policies
    - Least recently used (LRU) – evict the value that has been in the cache the longest without being accessed
    - Least frequently used (LFU) – evict the value that has been access the fewest number of times
    - First in, first out (FIFO) – evict the values in the same order they came in
  - Efficiency of a policy is measured by the hit ratio (number of cache hits vs. total memory accesses) and measured costs
    - Determined by working set and workload

# Caching in the hierarchy

| Cache Type | What is Cached? | Where is it Cached? | Latency (cycles) | Managed By |
|---|---|---|---|---|
| Registers | 4-8 byte words | CPU core | ~0 | Compiler |
| TLB | Address translations | On-chip TLB | ~0 | Hardware |
| L1 cache | 64-byte blocks | On-chip L1 | 1 | Hardware |
| L2 cache | 64-byte blocks | On/off-chip L2 | 10 | Hardware |
| Virtual memory | 4kiB pages | Main memory | 100 | Hardware + OS |
| Buffer cache | Parts of files | Main memory | 100 | OS |
| Disk cache | Disk sectors | Disk controller | 100,000 | Disk firmware |
| Network buffer cache | Parts of files | Local disk | 10,000,000 | Network FS client |
| Browser cache | Web pages | Local disk | 10,000,000 | Web browser |
| Web cache | Web pages | Remote server disks | 1,000,000,000 | Web proxy server |

# Next lecture

- Unix Essentials – getting started with using a Unix-like system
  - Basics of the command line
  - Using a traditional and powerful CLI editor
  - Writing simple scripts
- Hands-on!  Bring your laptop (with VM) if you have one.