CSCI 350
Ch. 9 – Caching and VM

Mark Redekopp

Michael Shindler & Ramesh Govindan

Examples of Caching Used

- What is caching?
  - Maintaining copies of information in locations that are faster to access than their primary home
- Examples
  - TLB
  - Data/instruction caches
  - Branch predictors
  - VM
  - Web browser
  - File I/O (disk cache)
  - Internet name resolutions

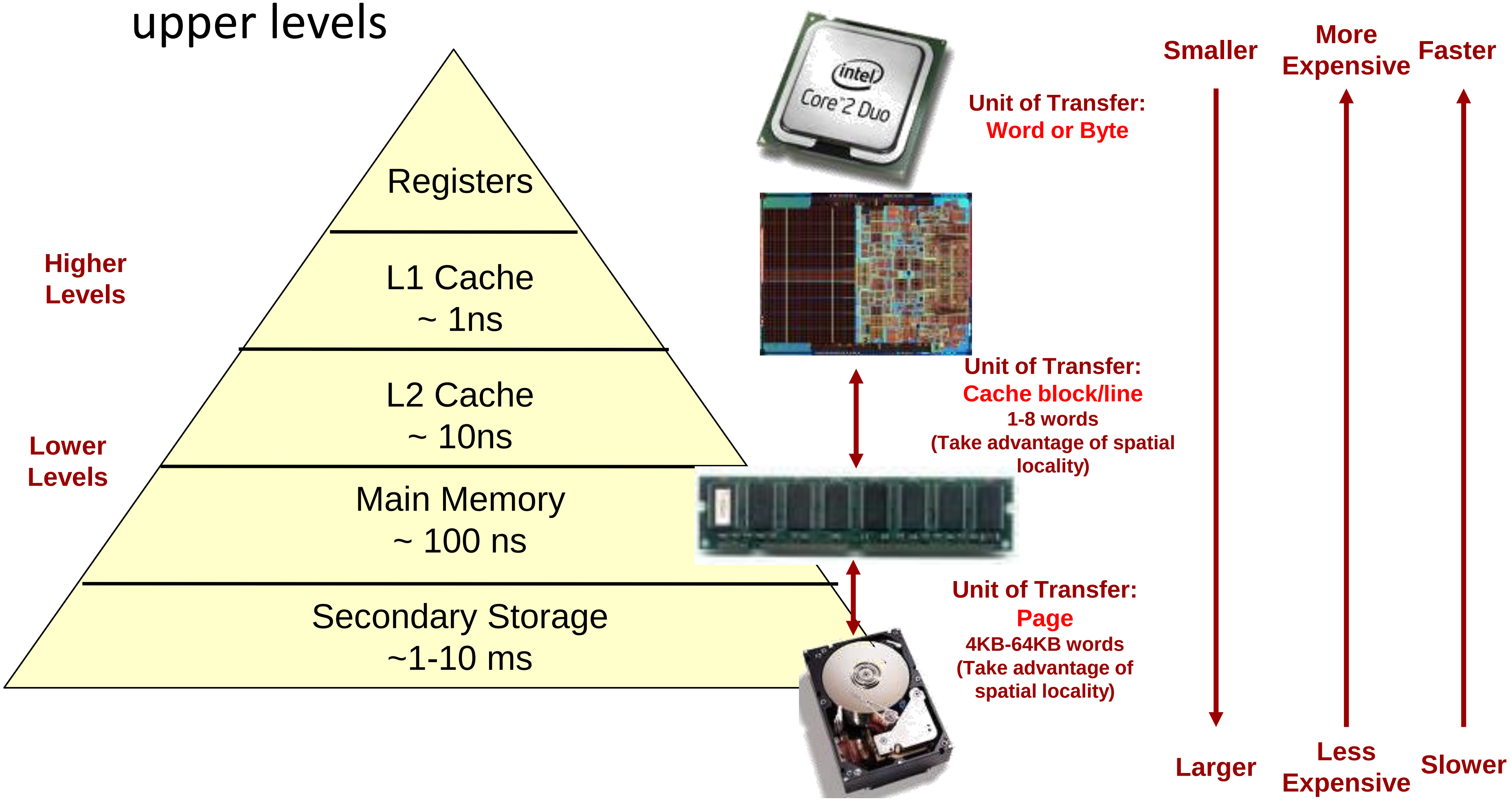
REVIEW OF DEFINITIONS & TERMS

What Makes a Cache Work

- What are the necessary conditions
  - – Locations used to store cached data must be faster to access than original locations
  - – Some reasonable amount of reuse
  - – Access patterns must be somewhat predictable

Memory Hierarchy & Caching

- Use several levels of faster and faster memory to hide delay of upper levels



Registers

L1 Cache
~ 1ns

L2 Cache
~ 10ns

Main Memory
~ 100 ns

Secondary Storage
~1-10 ms

Higher Levels

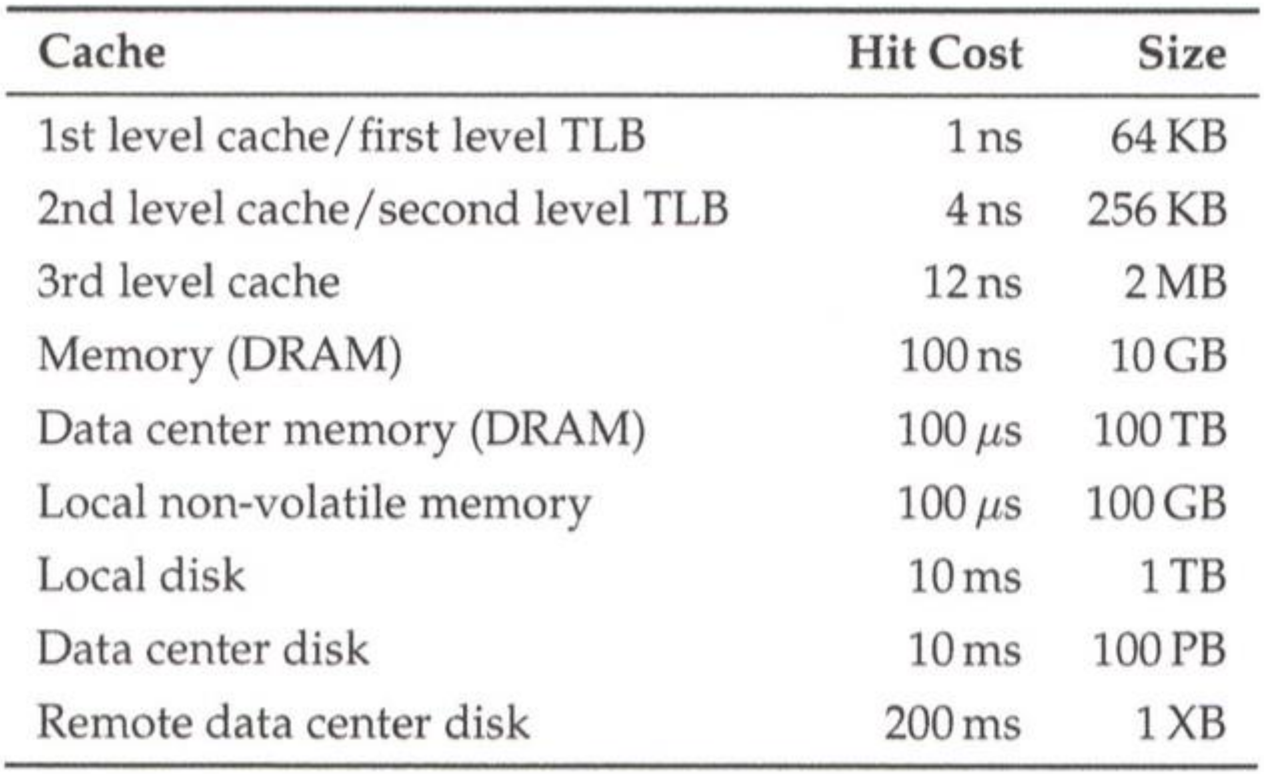
Lower Levels

Unit of Transfer:
Word or Byte

Unit of Transfer:
Cache block/line
1-8 words
(Take advantage of spatial locality)

Unit of Transfer:
Page
4KB-64KB words
(Take advantage of spatial locality)

Hierarchy Access Time & Sizes



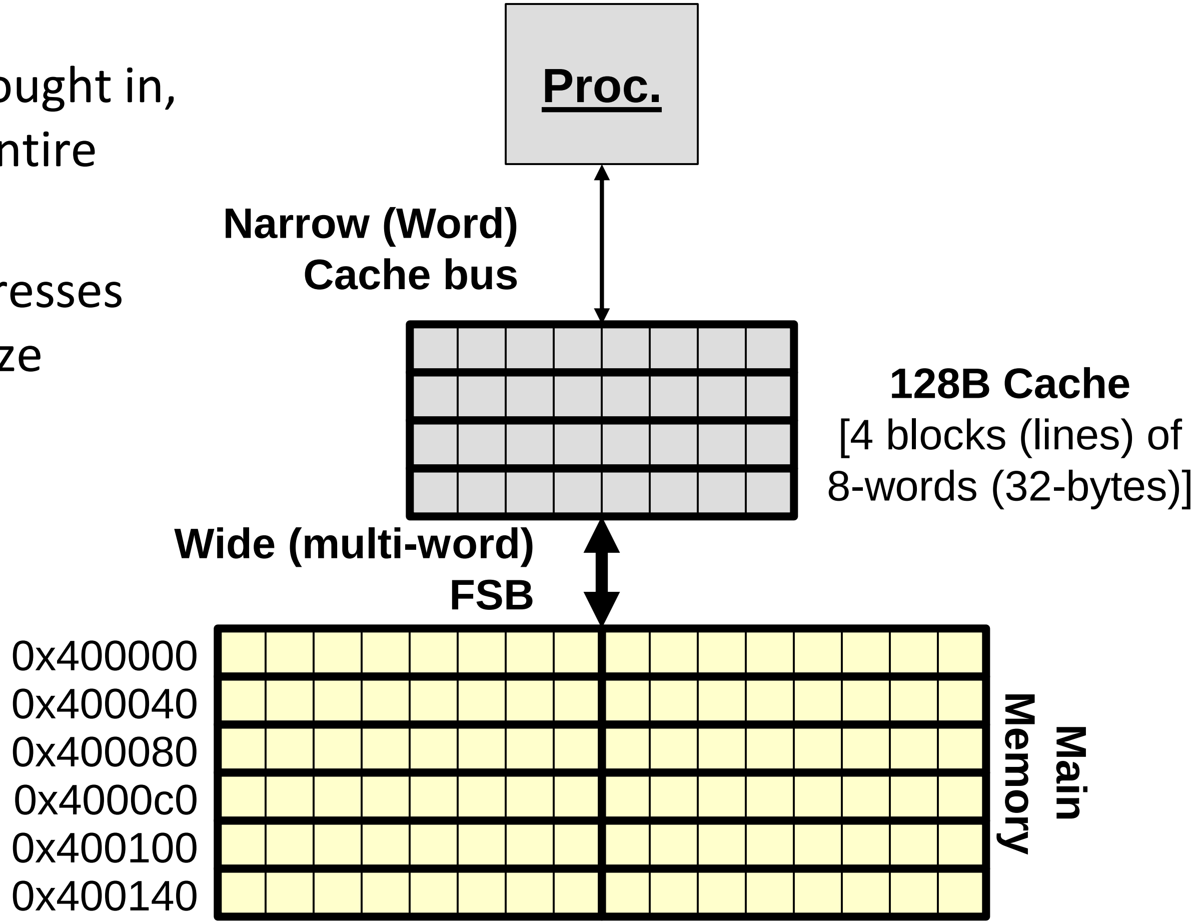
Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

Principle of Locality

- Caches exploit the Principle of Locality
  - Explains why caching with a hierarchy of memories yields improvement gain
- Works in two dimensions
  - **Temporal Locality**: If an item is referenced, it will tend to be referenced again soon
    - Examples: Loops, repeatedly called subroutines, setting a variable and then reusing it many times
  - **Spatial Locality**: If an item is referenced, items whose addresses are nearby will tend to be referenced soon
    - Examples: Arrays and program code

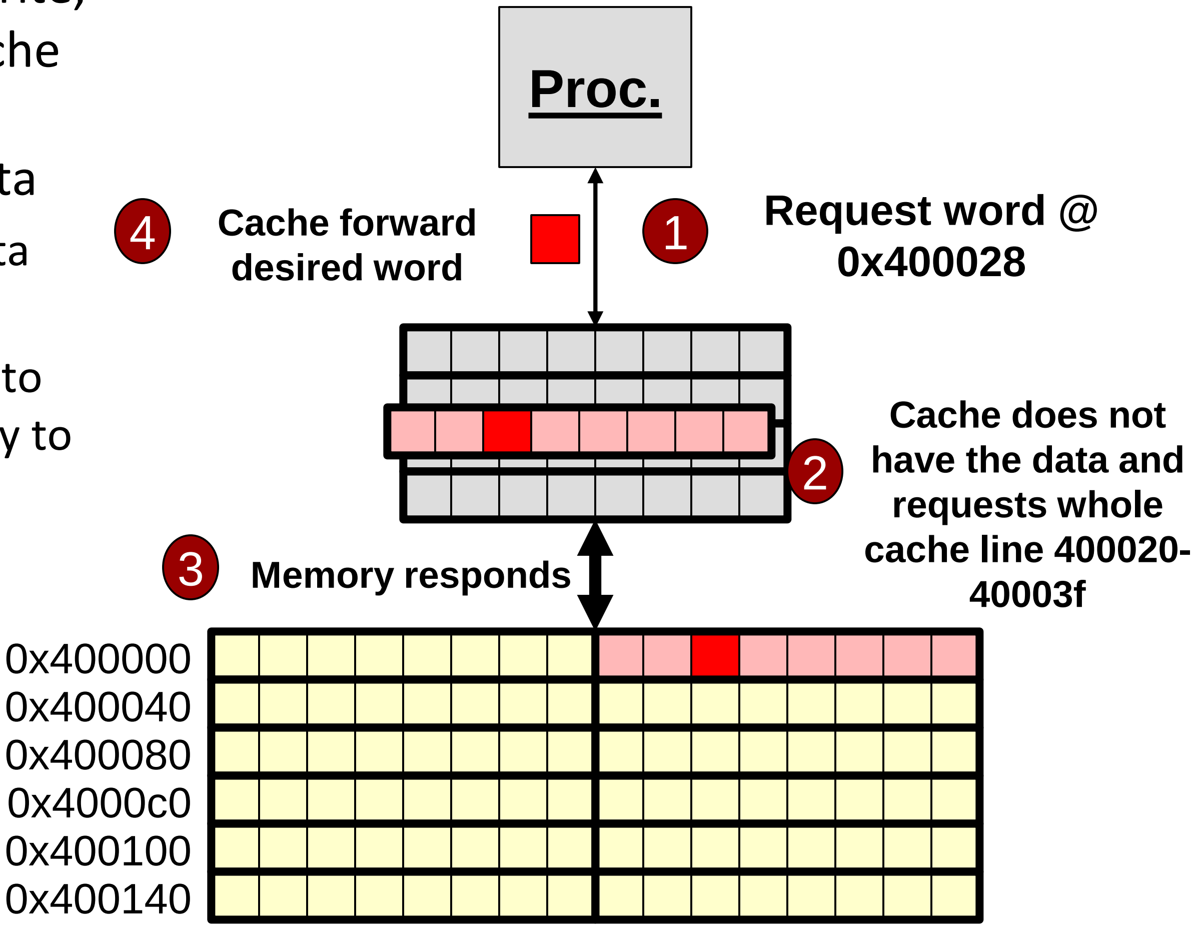
Cache Blocks/Lines

- Cache is broken into "blocks" or "lines"
  - Any time data is brought in, it will bring in the entire block of data
  - Blocks start on addresses multiples of their size



Cache Blocks/Lines

- Whenever the processor generates a read or a write, it will first check the cache memory to see if it contains the desired data
  - If so, it can get the data quickly from cache
  - Otherwise, it must go to the slow main memory to get the data

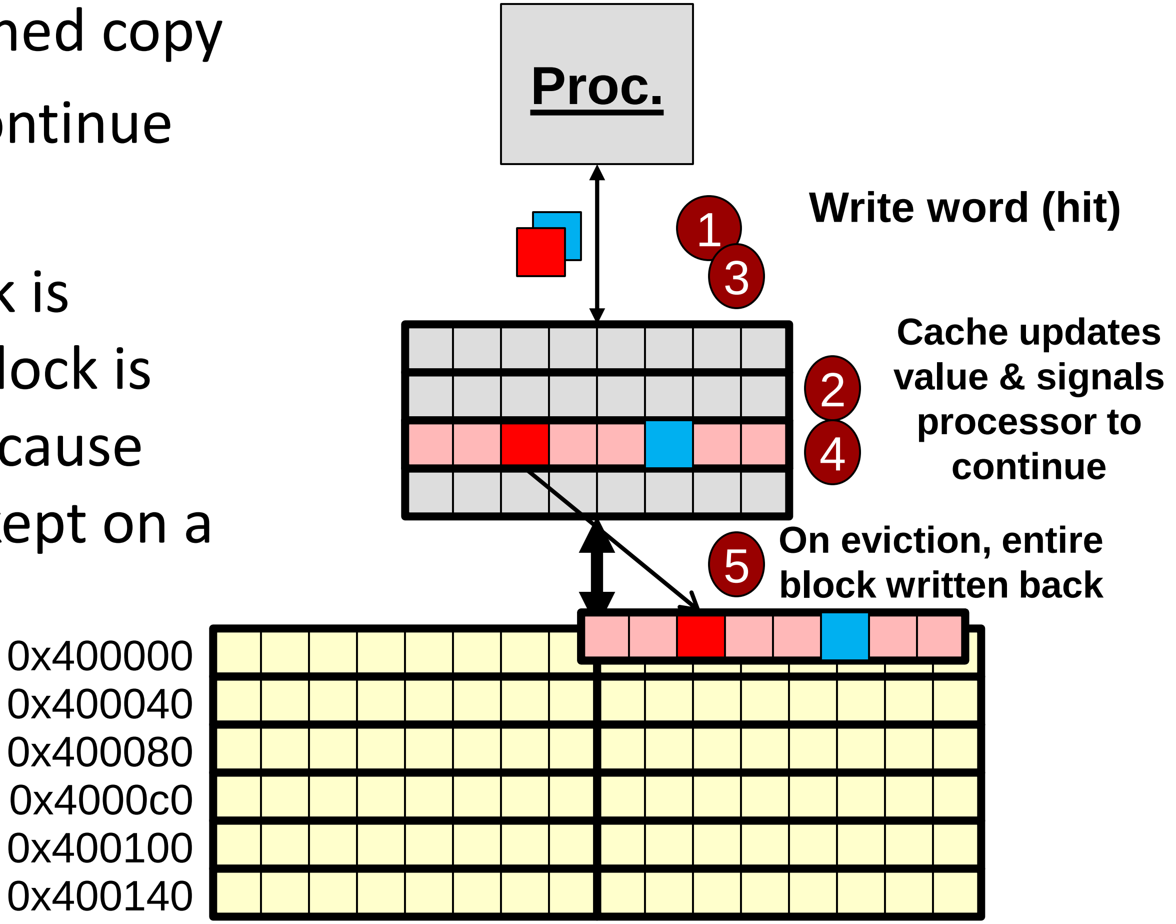

Proc.
4
Cache forward desired word
1
Request word @ 0x400028
2
Cache does not have the data and requests whole cache line 400020-40003f
3
Memory responds
0x400000
0x400040
0x400080
0x4000c0
0x400100
0x400140

Cache Definitions

- **Cache Hit** = Desired data is in current level of cache
- **Cache Miss** = Desired data is not present in current level
- When a cache miss occurs, the new block is brought from the lower level into cache
  - If cache is full a block must be evicted
- When CPU writes to cache, we may use one of two policies:
  - **Write Through (Store Through)**: Every write updates both current and next level of cache to keep them in sync. (i.e. coherent)
  - **Write Back**: Let the CPU keep writing to cache at fast rate, not updating the next level. Only copy the block back to the next level when it needs to be replaced or flushed

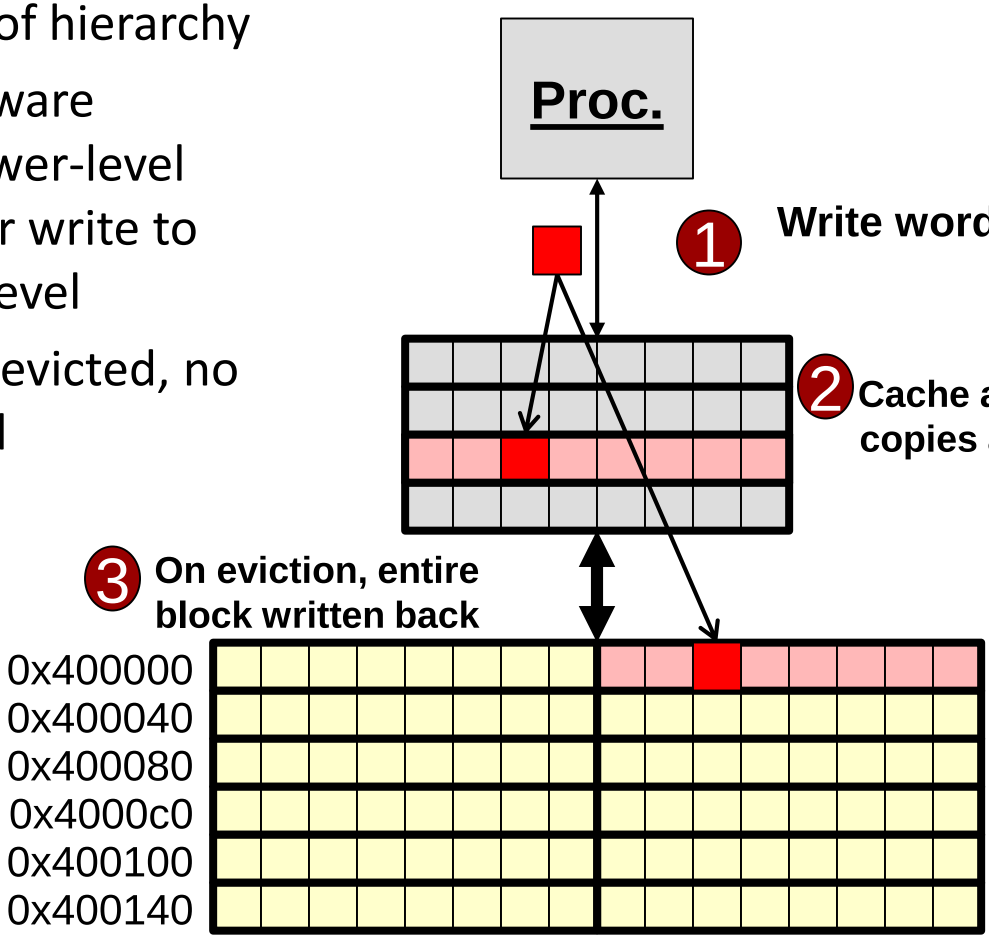
Write Back Cache

- On write-hit
  - Update only cached copy
  - Processor can continue quickly
  - Later when block is evicted, entire block is written back (because bookkeeping is kept on a per block basis)


Proc.
1
3
Write word (hit)
2
4
Cache updates value & signals processor to continue
5
On eviction, entire block written back
0x400000
0x400040
0x400080
0x4000c0
0x400100
0x400140

Write Through Cache

- On write-hit
  - Update both levels of hierarchy
  - Depending on hardware implementation, lower-level may have to wait for write to complete to lower level
  - Later when block is evicted, no writeback is needed

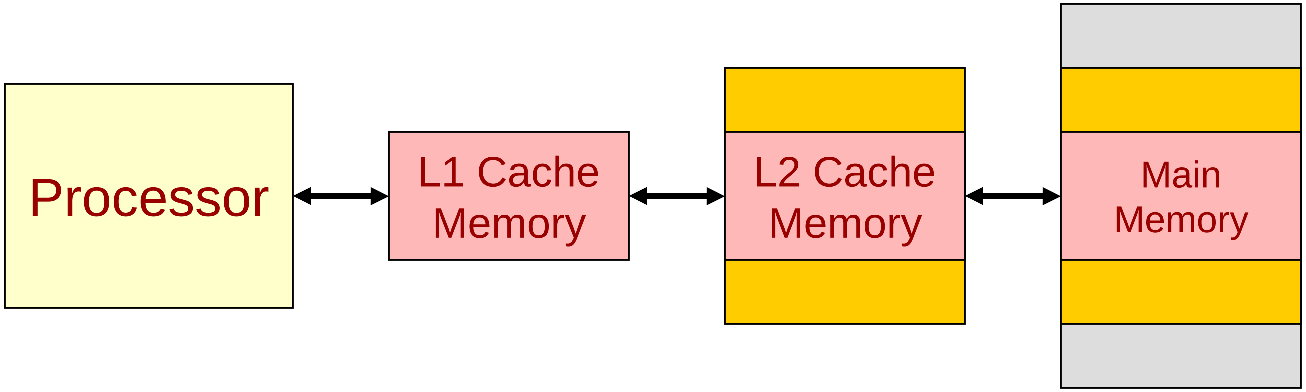

Proc.
1
Write word (hit)
2
Cache and memory copies are updated
3
On eviction, entire block written back
0x400000
0x400040
0x400080
0x4000c0
0x400100
0x400140

Write-through vs. Writeback

- Write-through
  - Pros
    - Avoid coherency issues between levels (need for eviction)
  - Cons
    - Poor performance if next level of hierarchy is slow (VM page fault to disk) or if many, repeated accesses
- Writeback
  - Pros
    - Fast if many repeated accesses
  - Cons
    - Coherency issues
    - Slow if few, isolated writes since entire block must be written back

Principle of Inclusion

- When the cache at level j misses on data that is store in level k (j < k), the data is brought into all levels i where j < i < k
- This implies that lower levels always contains a subset of higher levels
- Example:
  - L1 contains most recently used data
  - L2 contains that data + data used earlier
  - MM contains all data
- This make coherence far easier to maintain between levels



Average Access Time

- Define parameters
  - H_i = Hit Rate of Cache Level L_i
    (Note that $1-H_i$ = Miss rate)
  - T_i = Access time of level i
  - R_i = Burst rate per word of level i (after startup access time)
  - B = Block Size
- Let us find T_{AVE} = average access time

T_{ave} without L2 cache

- 2 possible cases:
  - Either we have a hit and pay only the L1 cache hit time
  - Or we have a miss and read in the whole block to L1 and then read from L1 to the processor
- $T_{ave} = T_1 + \underbrace{(1-H_1)\bullet[T_{MM} + B\bullet R_{MM}]}_{\text{(Miss Rate)*(Miss Penalty)}}$
- For T_1=10ns, H_1 = 0.9, B=8, T_{MM}=100ns, R_{MM}=25ns
  - T_{ave} = 10 + [(0.1) • (100+8•25)] = 40 ns

T_{ave} with L2 cache

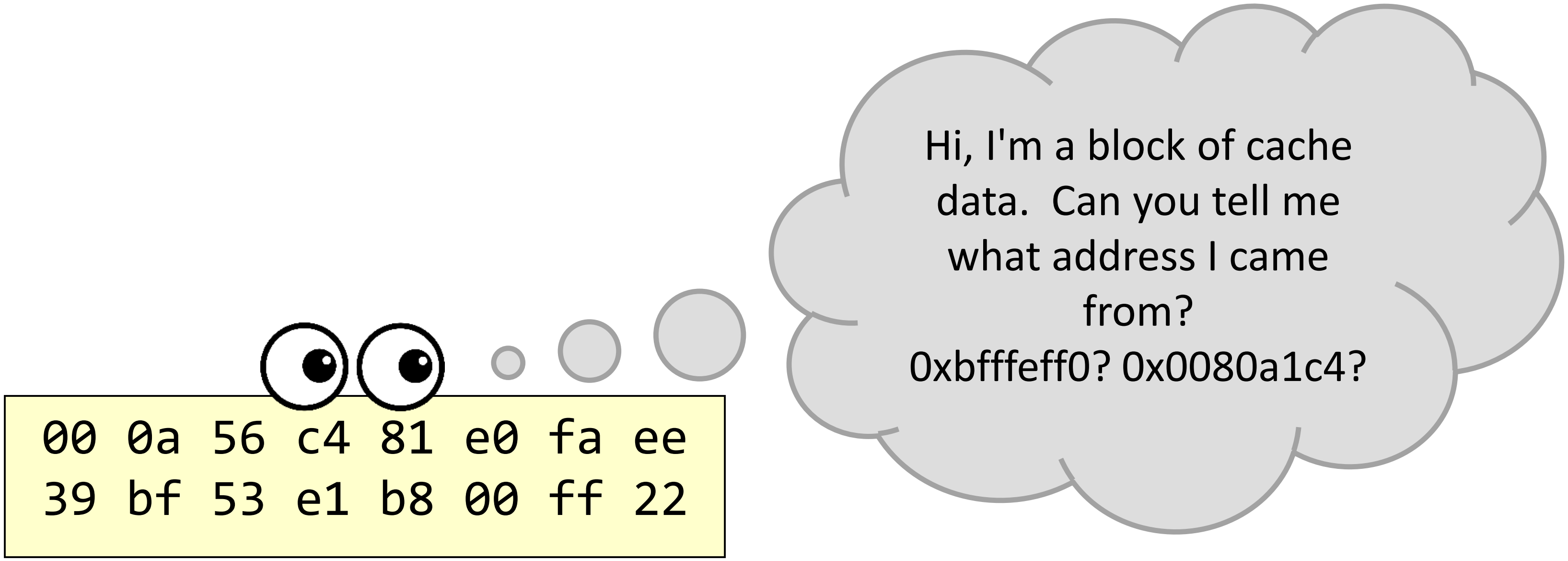
- 3 possible cases:
  - Either we have a hit and pay the L1 cache hit time
  - Or we miss L1 but hit L2 and read in the block from L2
  - Or we miss L1 and L2 and read in the block from MM
- $T_{ave} = T_1 + \underbrace{(1\text{-}H_1)\bullet H_2\bullet(T_2+B\bullet R_2)}_{\textbf{L1 miss / L2 Hit}} + \underbrace{(1\text{-}H_1)\bullet(1\text{-}H_2)\bullet(T_{MM}+B\bullet R_{MM})}_{\textbf{L1 miss / L2 Miss}}$
- For T_1 = 10ns, H_1 = 0.9, T_2 = 20ns, R_2 = 10ns, H_2 = 0.98, B=8, T_{MM}=100ns, R_{MM}=25 ns
- $T_{ave} = 10 + (0.1)\bullet(.98)\bullet(20+8\bullet10) + (0.1)\bullet(.02)\bullet(100+8\bullet25)$
  $= 10 + 9.8\text{ ns} + 0.6 = 20.4\text{ ns}$

Three Main Issues

- Finding cached data (hit/miss)
- Replacement algorithms
- Coherency (managing multiple versions)
  - Discussed in previous lectures

MAPPINGS

Cache Question



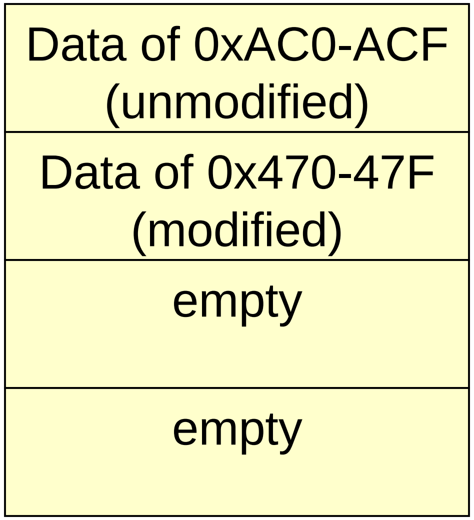
```
00 0a 56 c4 81 e0 fa ee
39 bf 53 e1 b8 00 ff 22
```

Hi, I'm a block of cache data. Can you tell me what address I came from?

0xbfffeff0? 0x0080a1c4?

Cache Implementation

- Assume a cache of 4 blocks of 16-bytes each
- Must store more than just data!
- What other bookkeeping and identification info is needed?
  - Has the block been modified
  - Is the block empty or full
  - Address range of the data: Where did I come from?



Data of 0xAC0-ACF (unmodified)

Data of 0x470-47F (modified)

empty

empty

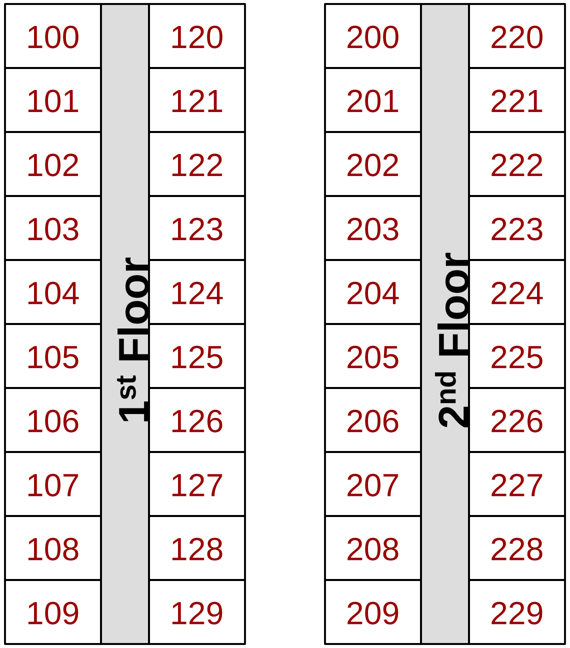
Cache with 4 data blocks

Implementation Terminology

- What bookkeeping values must be stored with the cache in addition to the block data?
- **Tag** – Portion of the block's address range used to identify the MM block residing in the cache from other MM blocks.
- **Valid bit** – Indicates the block is occupied with valid data (i.e. not empty or invalid)
- **Dirty bit** – Indicates the cache and MM copies are "inconsistent" (i.e. a write has been done to the cached copy but not the main memory copy)
  - Used for write-back caches

Identifying Blocks via Address Range

- Possible methods
  - Store start and end address (requires multiple comparisons)
  - Ensure block ranges sit on binary boundaries (upper address bits identify the block with a single value)
    - Analogy: Hotel room layout/addressing



100		120	200		220
101		121	201		221
102		122	202		222
103		123	203		223
104		124	204		224
105	1st Floor	125	205	2nd Floor	225
106		126	206		226
107		127	207		227
108		128	208		228
109		129	209		229

Analogy: Hotel Rooms

1st Digit = Floor
2nd Digit = Aisle
3rd Digit = Room w/in aisle

To refer to the range of rooms on the second floor, left aisle we would just say rooms **20x**

4 word (16-byte) blocks:

Addr. Range	Binary		
000-00f	0000	0000	0000 - 1111
010-01f	0000	0001	0000 - 1111

8 word (32-byte) blocks:

Addr. Range	Binary		
000-01f	0000	000	00000 - 11111
020-03f	0000	001	00000 - 11111

Cache Implementation

- Assume 12-bit addresses and 16-byte blocks
- Block addresses will range from xx0-xxF
  - Address can be broken down as follows
  - A[11:4] = identifies block range (i.e. xx0-xxF)
  - A[3:0] = byte offset within the cache block

A[11:4]	A[3:0]
Tag	**Byte**

Addr. = 0x124

Word 0x4 (1st) w/in block 120-12F

0001 0010	**0100**

Addr. = 0xACC

Word 0xC (3rd) w/in block AC0-

1010 1100	**1100**

Cache Implementation

- To identify which MM block resides in each cache block, the tags need to be stored along with the Dirty and Valid bits

Tag

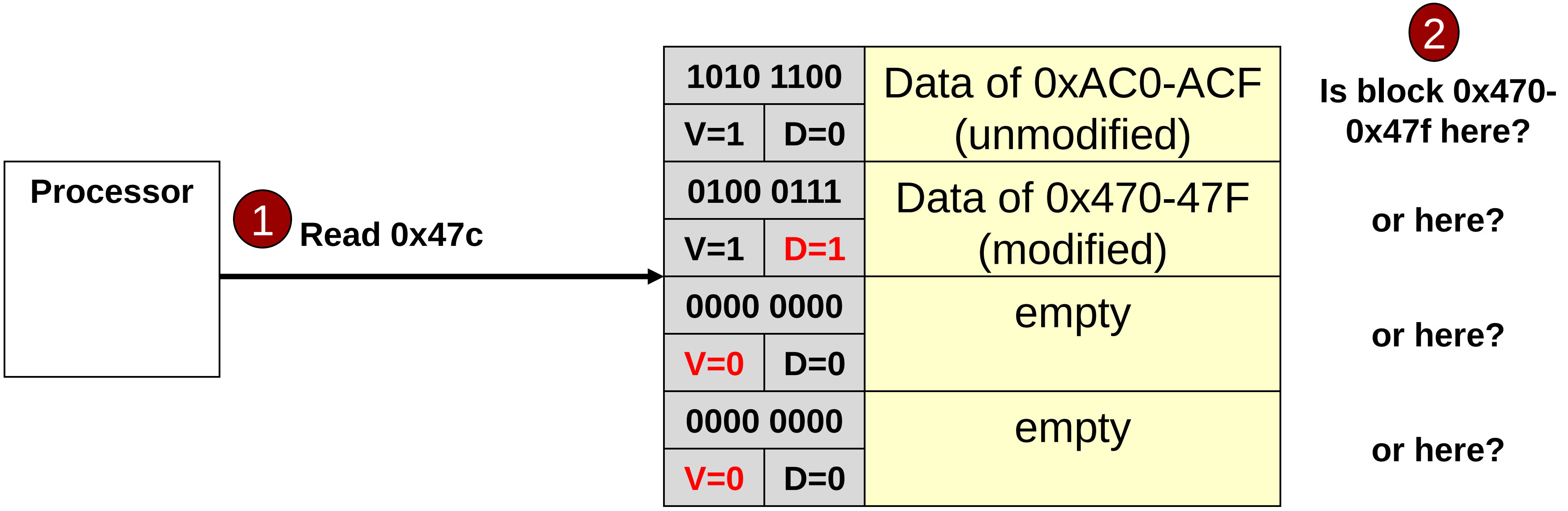
1010 1100		Data of 0xAC0-ACF (unmodified)
V=1	D=0	
0100 0111		Data of 0x470-47F (modified)
V=1	D=1	
0000 0000		empty
V=0	D=0	
0000 0000		empty
V=0	D=0	

Scenario

- You lost your keys
- You think back to where you have been lately
  - You've been the library, to class, to grab food at campus center, and the gym
  - Where do you have to look to find your keys?
- If you had been home all day and discovered your keys were missing, where would you have to look?
- **Key lesson**: If something can be anywhere you have to search ____________
  - By contrast, if we limit where things can be then our search need only look in those limited places

Content-Addressable Memory

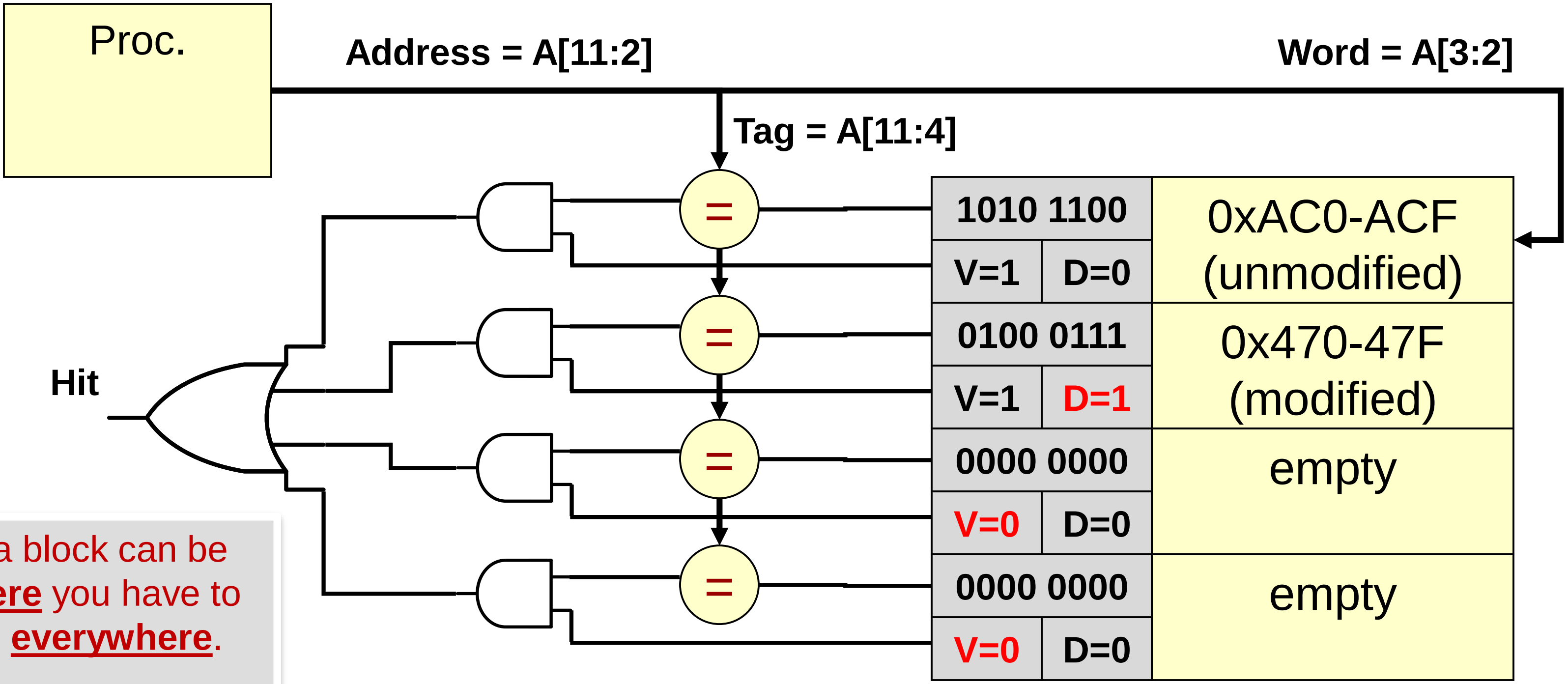
- Cache memory is one form of what is known as “content-addressable” memory
  - This means data can be in any location in memory and does not have one particular address
  - Additional information is saved with the data and is used to “address”/find the desired data (this is the “tag” in this case) via a search on each access
  - This search can be very time consuming!!



1010 1100		Data of 0xAC0-ACF (unmodified)
V=1	D=0	
0100 0111		Data of 0x470-47F (modified)
V=1	D=1	
0000 0000		empty
V=0	D=0	
0000 0000		empty
V=0	D=0	

Tag Comparison

- When caches have many blocks (> 16 or 32) it can be expensive (hardware-wise) to check all tags

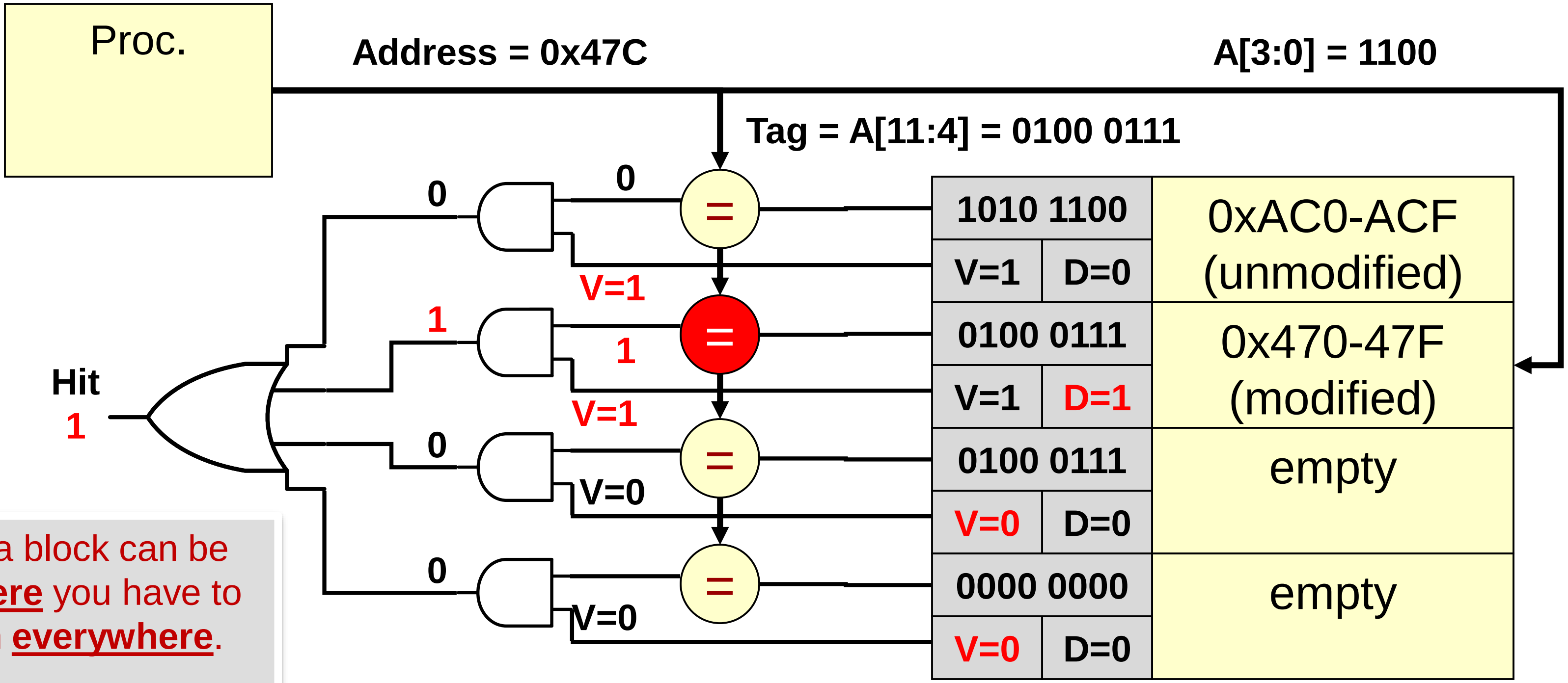


1010 1100		0xAC0-ACF (unmodified)	
V=1	D=0		
0100 0111		0x470-47F (modified)	
V=1	D=1		
0000 0000		empty	
V=0	D=0		
0000 0000		empty	
V=0	D=0		

When a block can be **anywhere** you have to search **everywhere**.

Tag Comparison Example

- Tag portion of desired address is check against all the tags and qualified with the valid bits to determine a hit



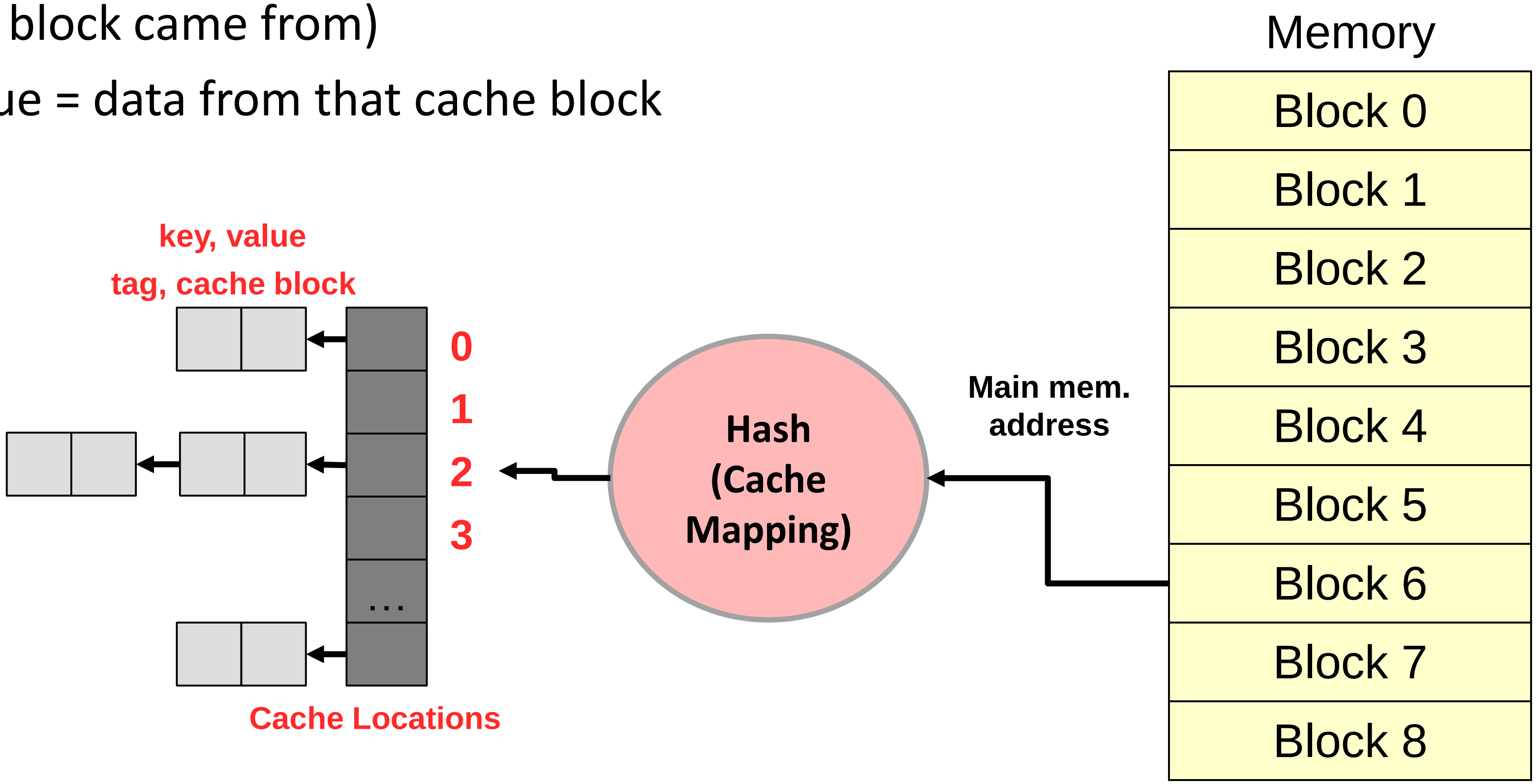
When a block can be **anywhere** you have to search **everywhere**.

Mapping Techniques

- Determines where blocks can be placed in the cache
- By reducing number of possible MM blocks that map to a cache block, hit logic (searches) can be done faster
- 3 Primary Methods
  - Direct Mapping
  - Fully Associative Mapping
  - Set-Associative Mapping

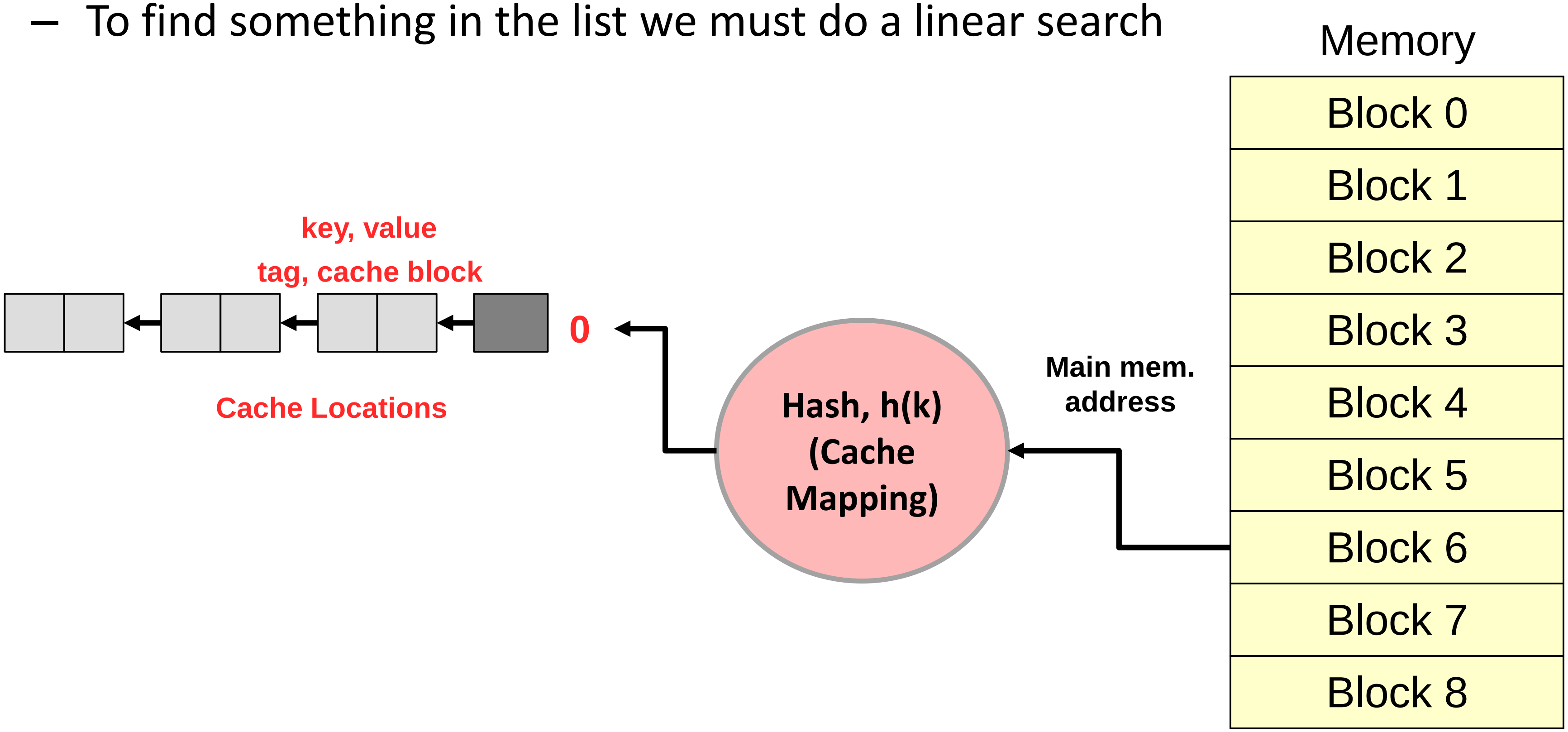
Cache Mapping Schemes

- Cache mappings are really just variations in hash table configurations
  - We hash the larger memory space to the smaller cache space
  - Key = originating memory address (e.g. main memory address where the block came from)
  - Value = data from that cache block



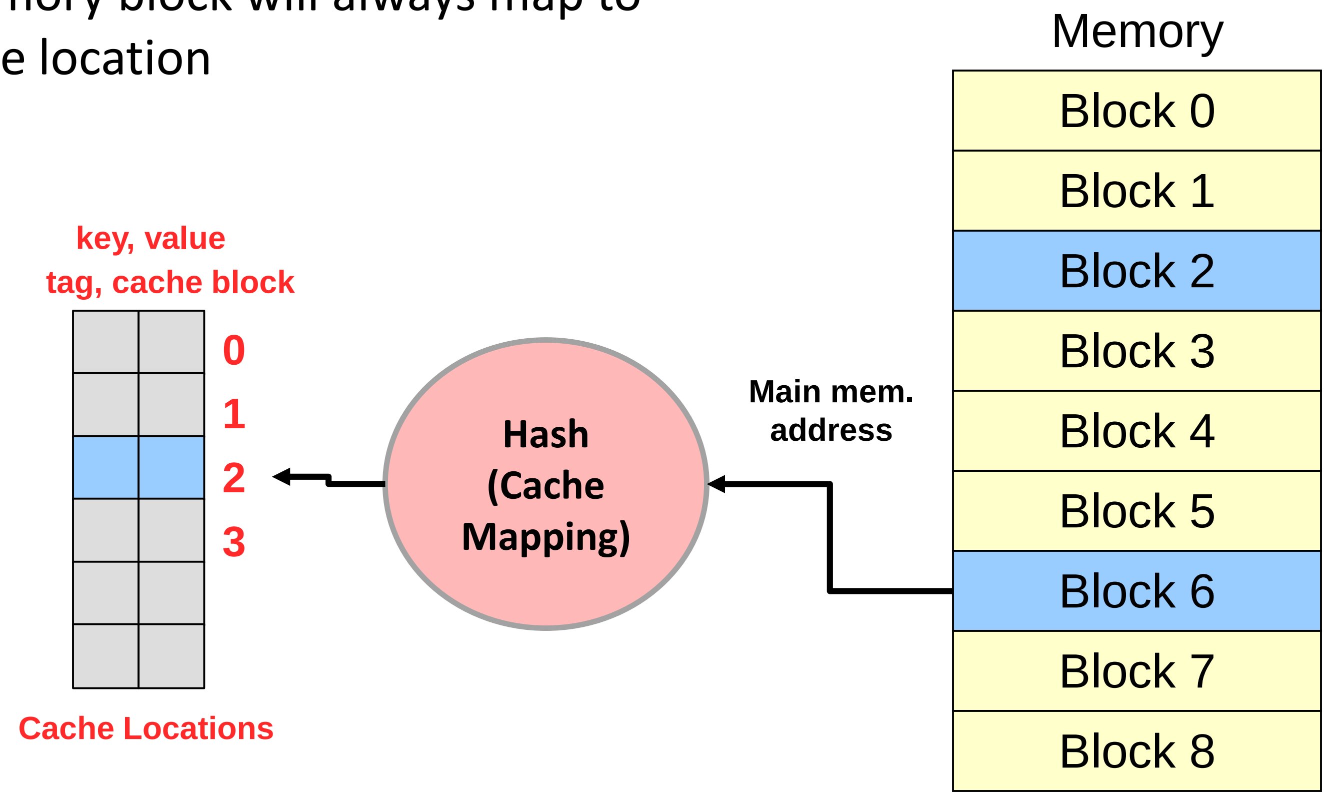
Fully Associative Cache Mapping

- Any memory block can go anywhere
- Like a hash table with 1 bucket/chain [h(k) = 0]
  - Turns into a linked list
  - To find something in the list we must do a linear search



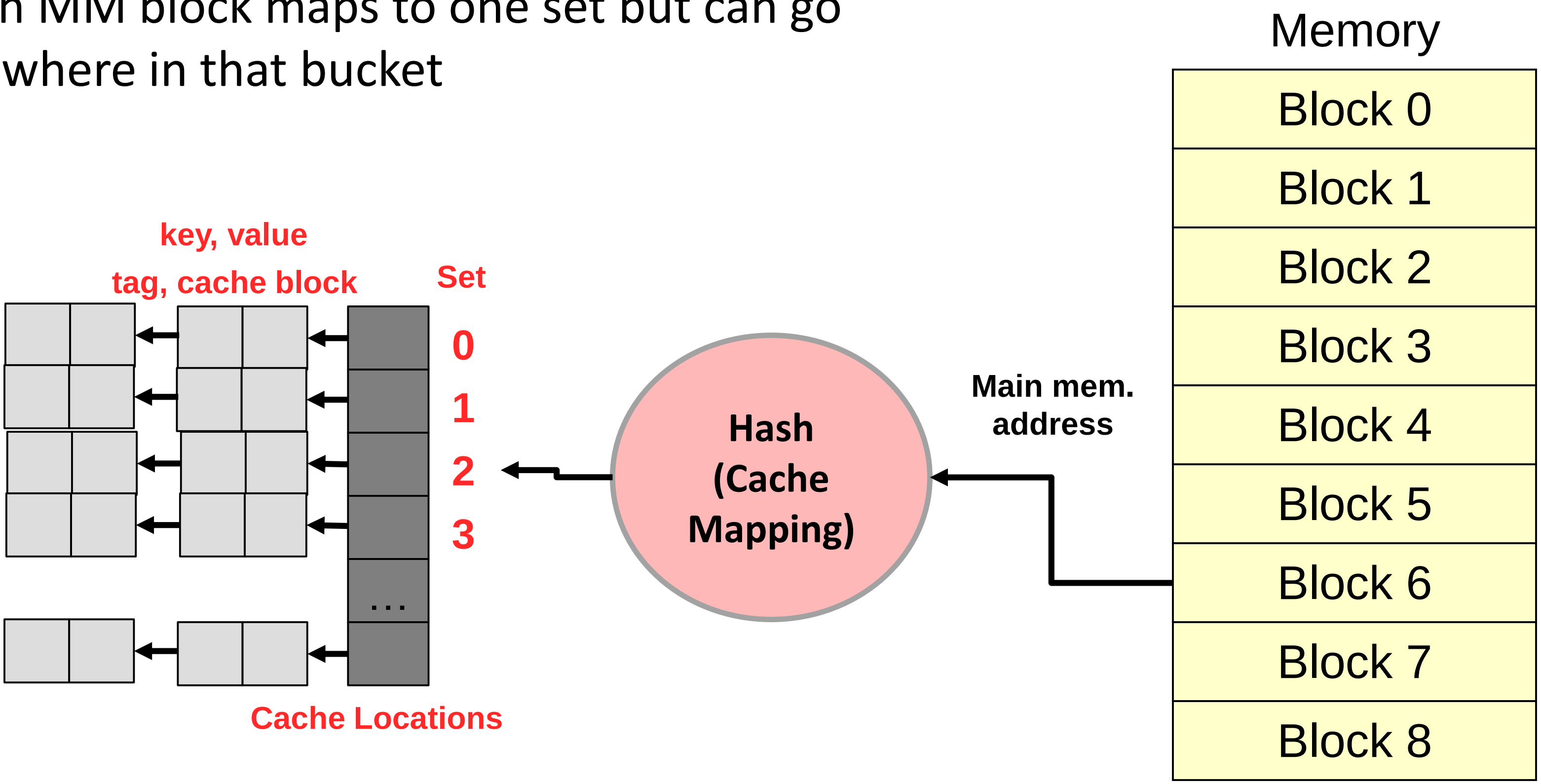
Direct Mapped Caches

- Cache is like a hash table without chaining (one slot per bucket)
  - Collisions yield to evictions
  - Each main memory block will always map to the same cache location



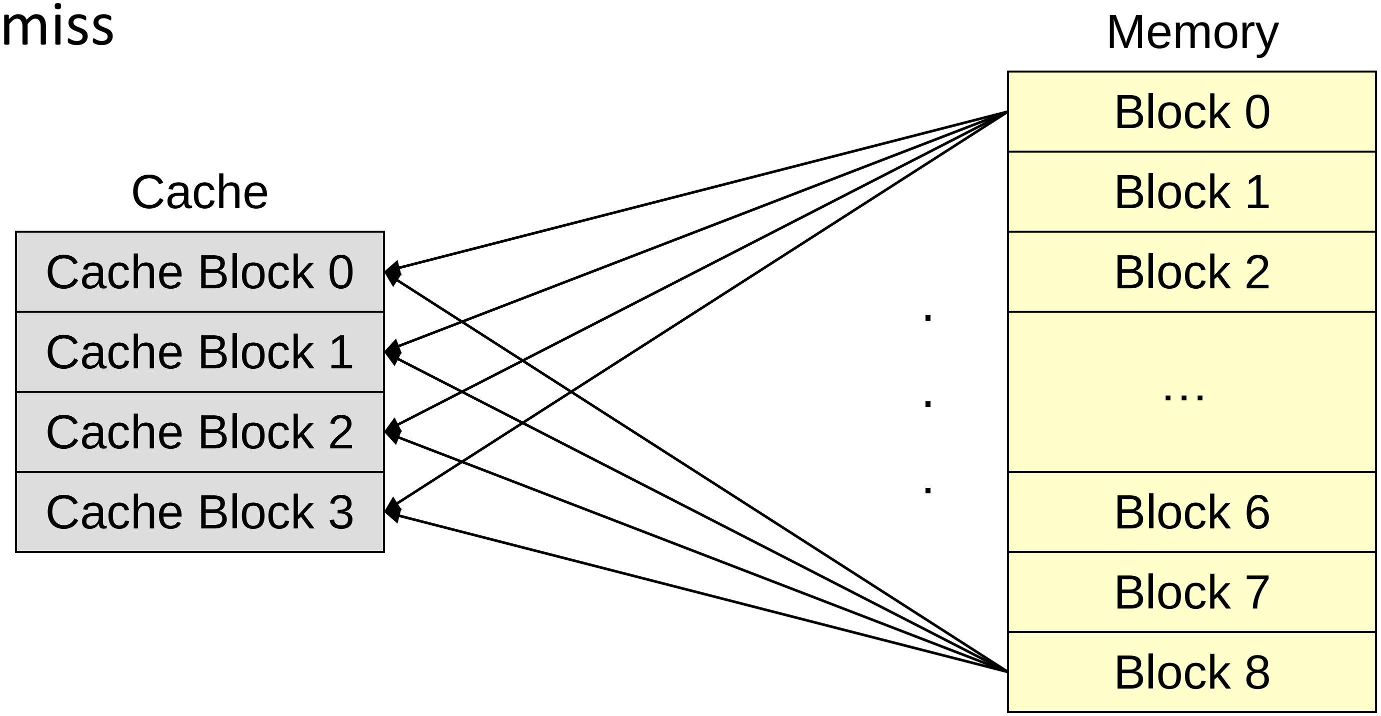
K-way Set Associative Mapping

- Buckets in the hash table are limited to size=k
  - Once a bucket is full, must evict blocks to make room for a new one
  - Each bucket is referred to as a set
  - Each MM block maps to one set but can go anywhere in that bucket



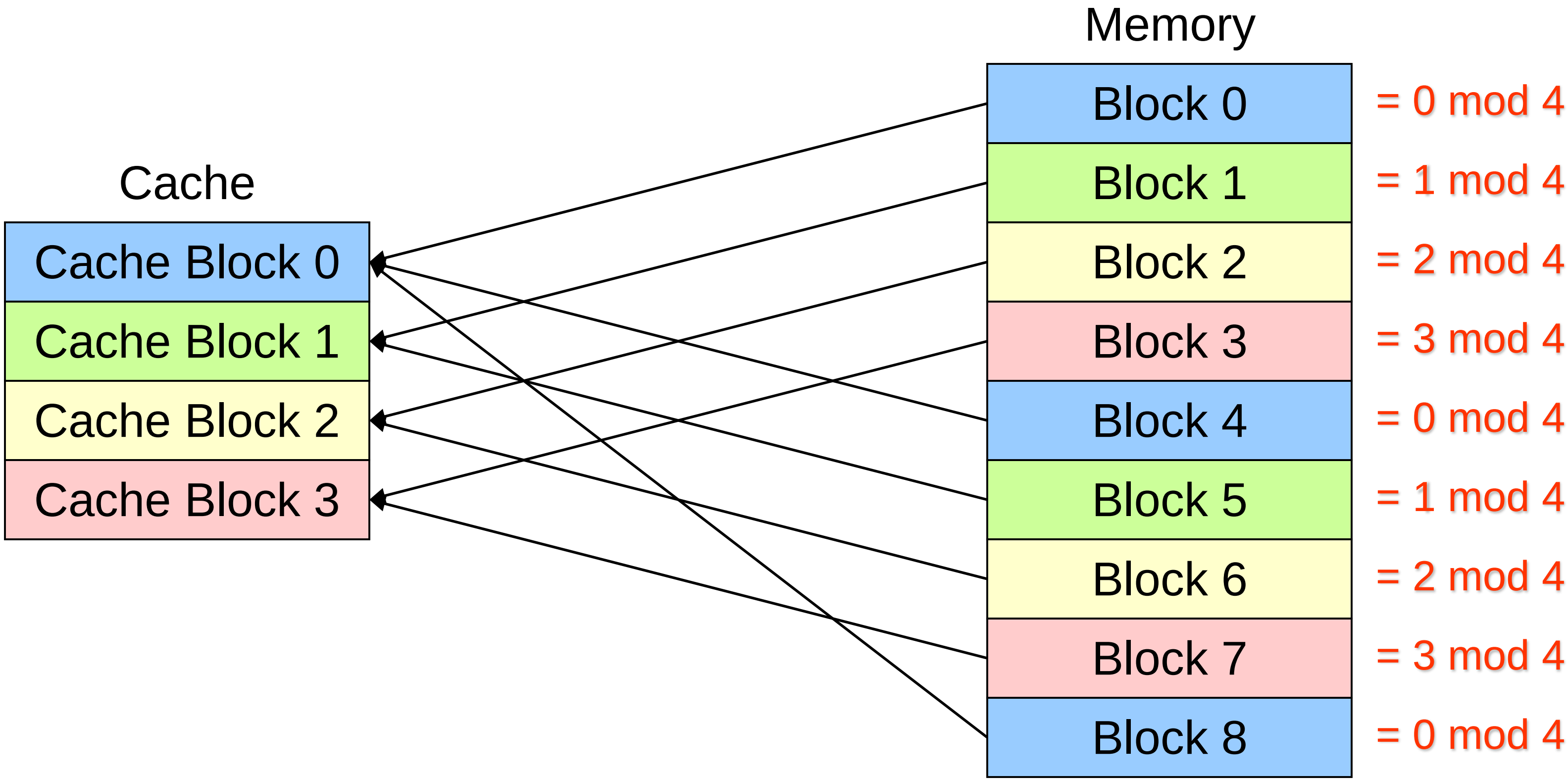
Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no restriction)
  - Implies we have to search everywhere to determine hit or miss



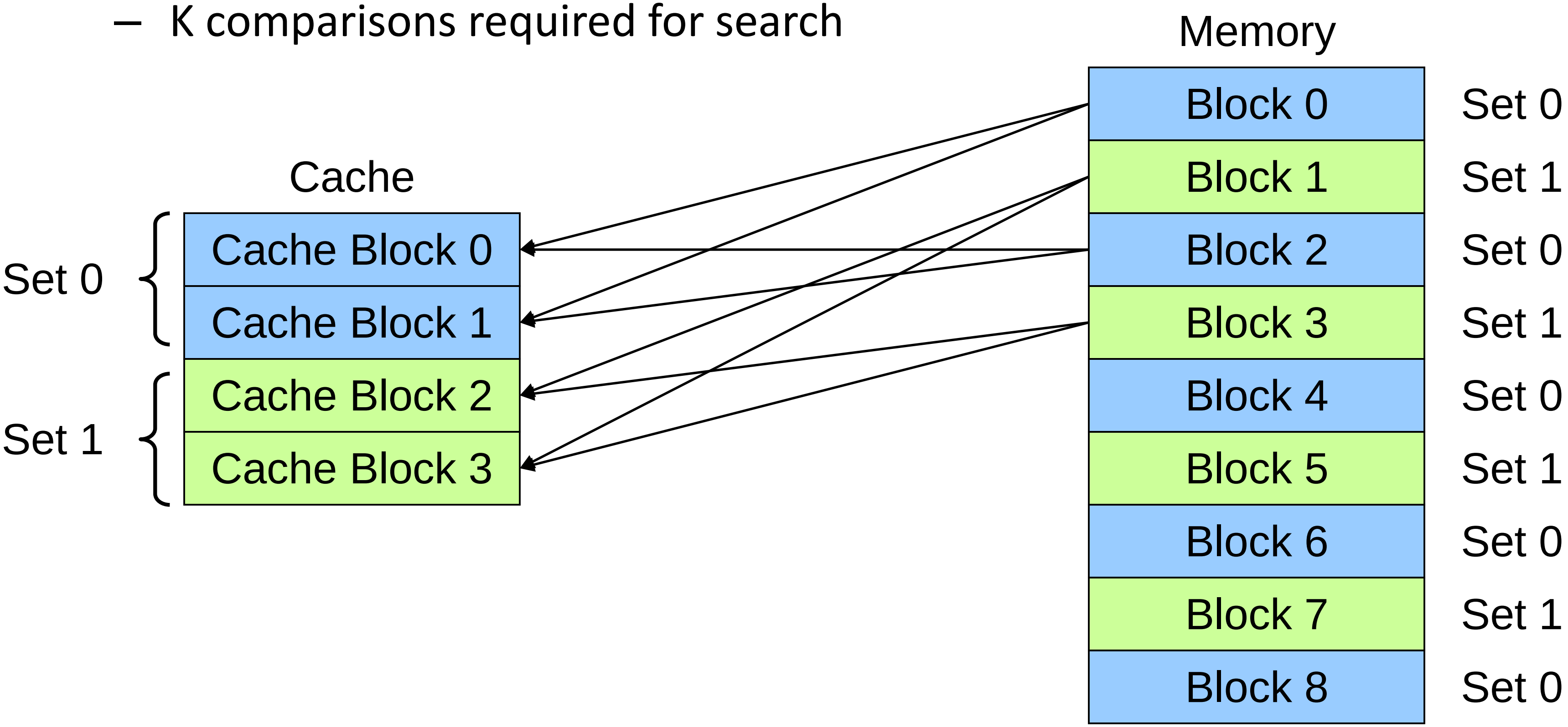
Direct Mapping

- Each block from memory can only be put in one location
- Given n cache blocks,
  MM block i maps to cache block i mod n



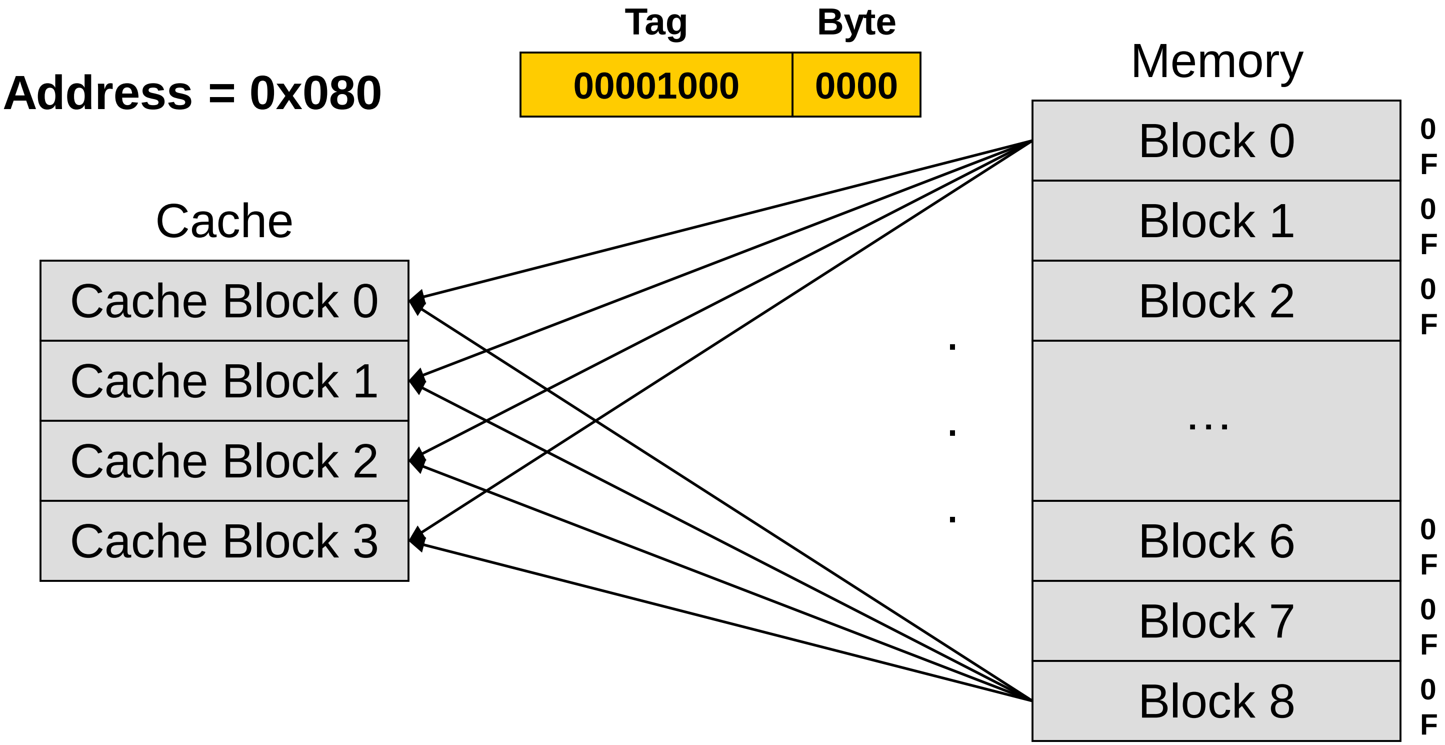
K-way Set-Associative Mapping

- Given, S sets, block i of MM maps to set i mod s
- Within the set, block can be put anywhere
- Let k = number of cache blocks per set = n/s
  - K comparisons required for search



Fully Associative Implementation

- 12-bit address:
  - 16 bytes per block => 4 LSB's used to determine the desired byte/word offset within the block
  - Tag = Block # = Upper bits used to identify the block in the cache

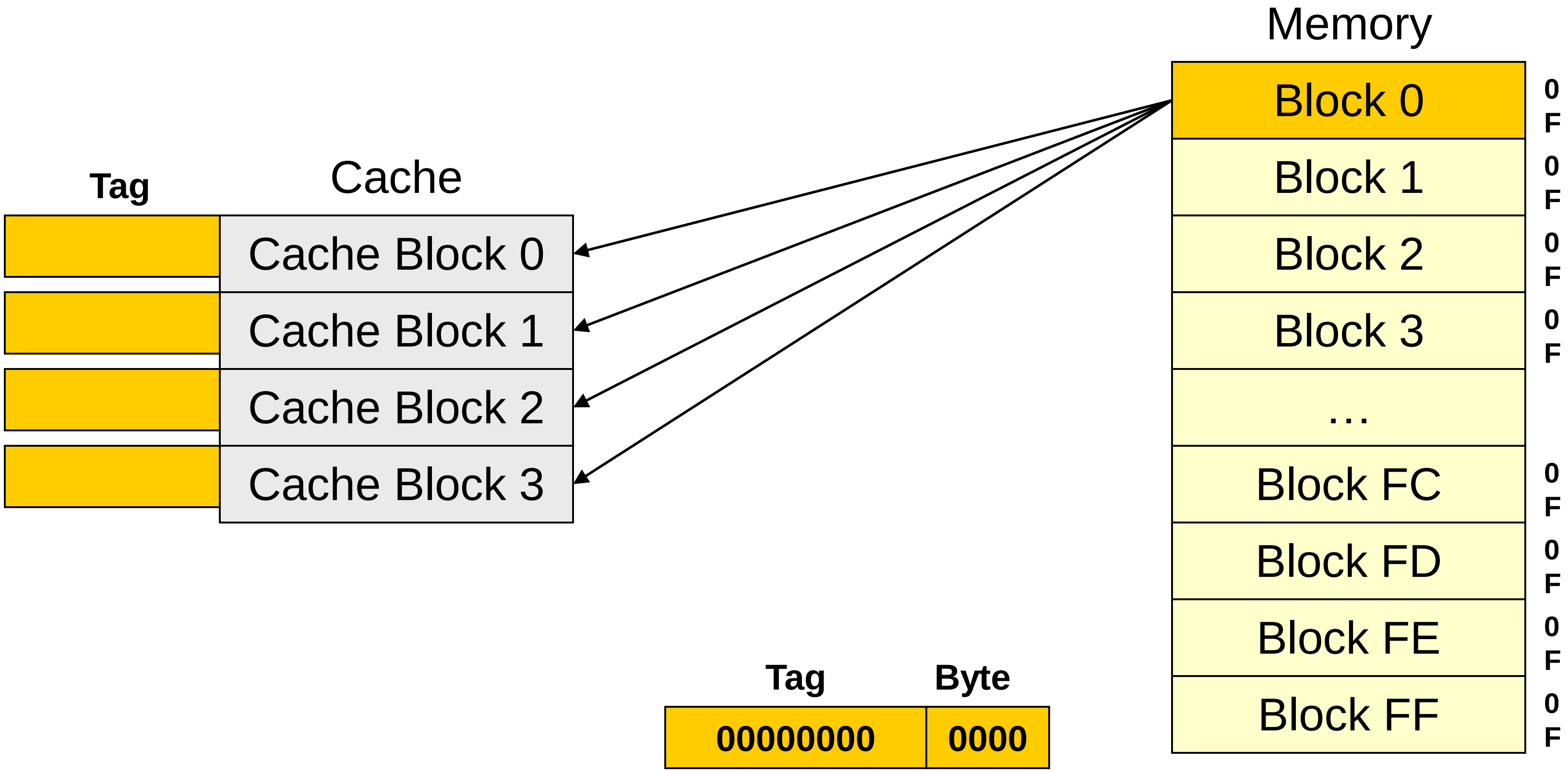


Fully Associative Address Scheme

- A[1:0] unused (word access only)
- Word bits = $\log_2 B$ bits (B=Block Size)
- Tag = Remaining bits

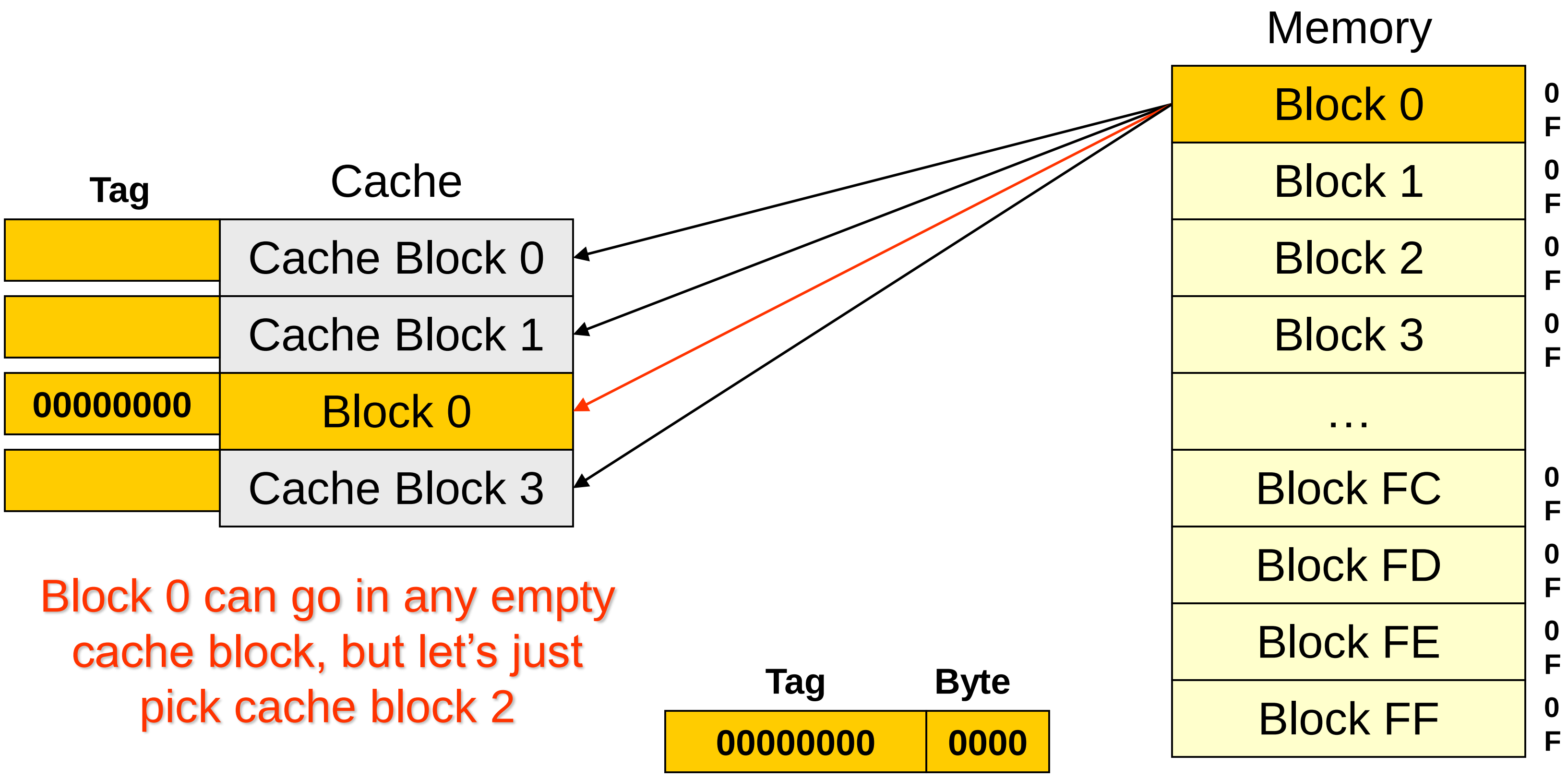
Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no mapping scheme)
- Completely flexible



Fully Associative Mapping

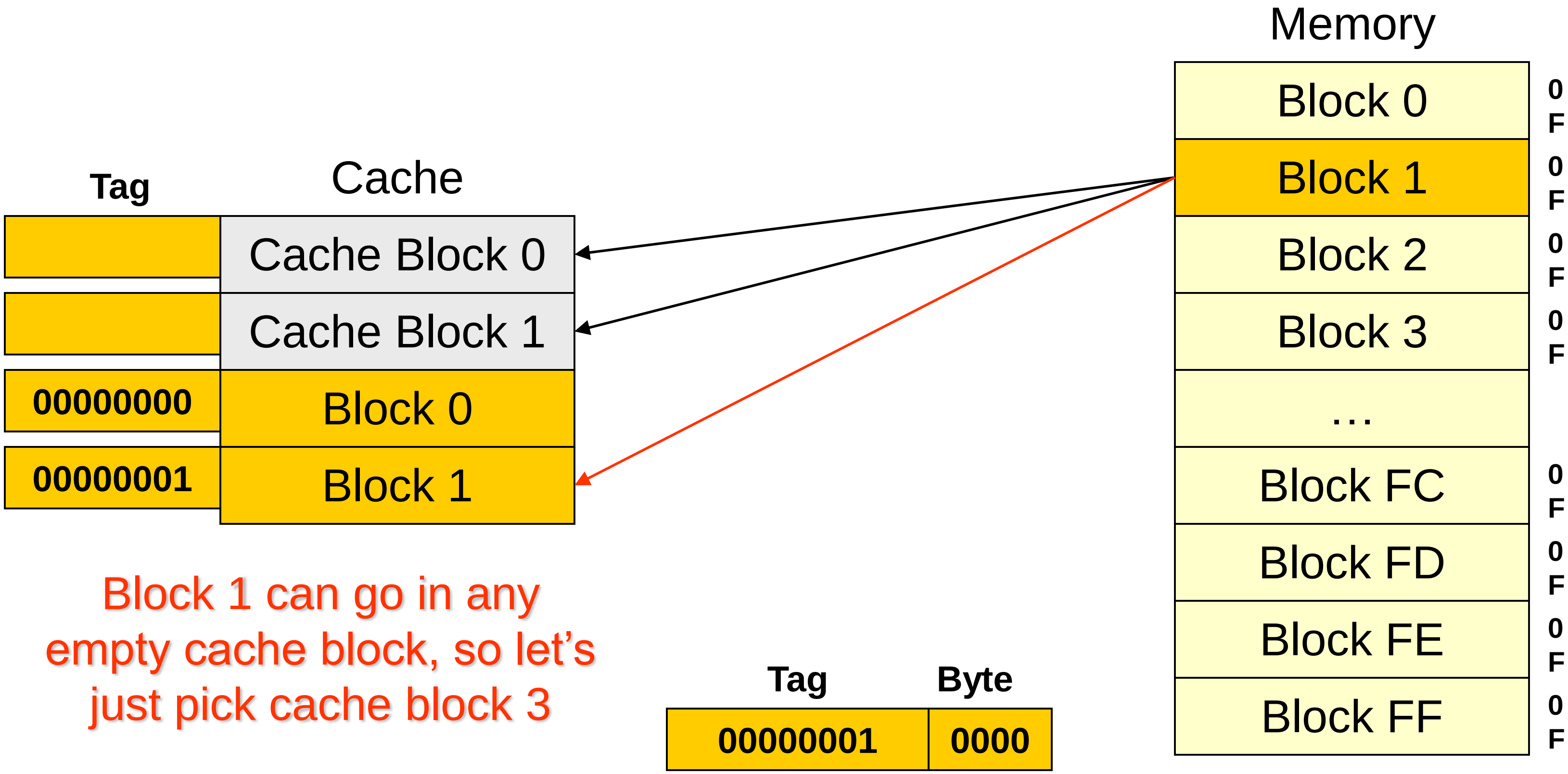
- Any block from memory can be put in any cache block (i.e. no mapping)



Block 0 can go in any empty cache block, but let's just pick cache block 2

Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no mapping)



Memory

Memory	
Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

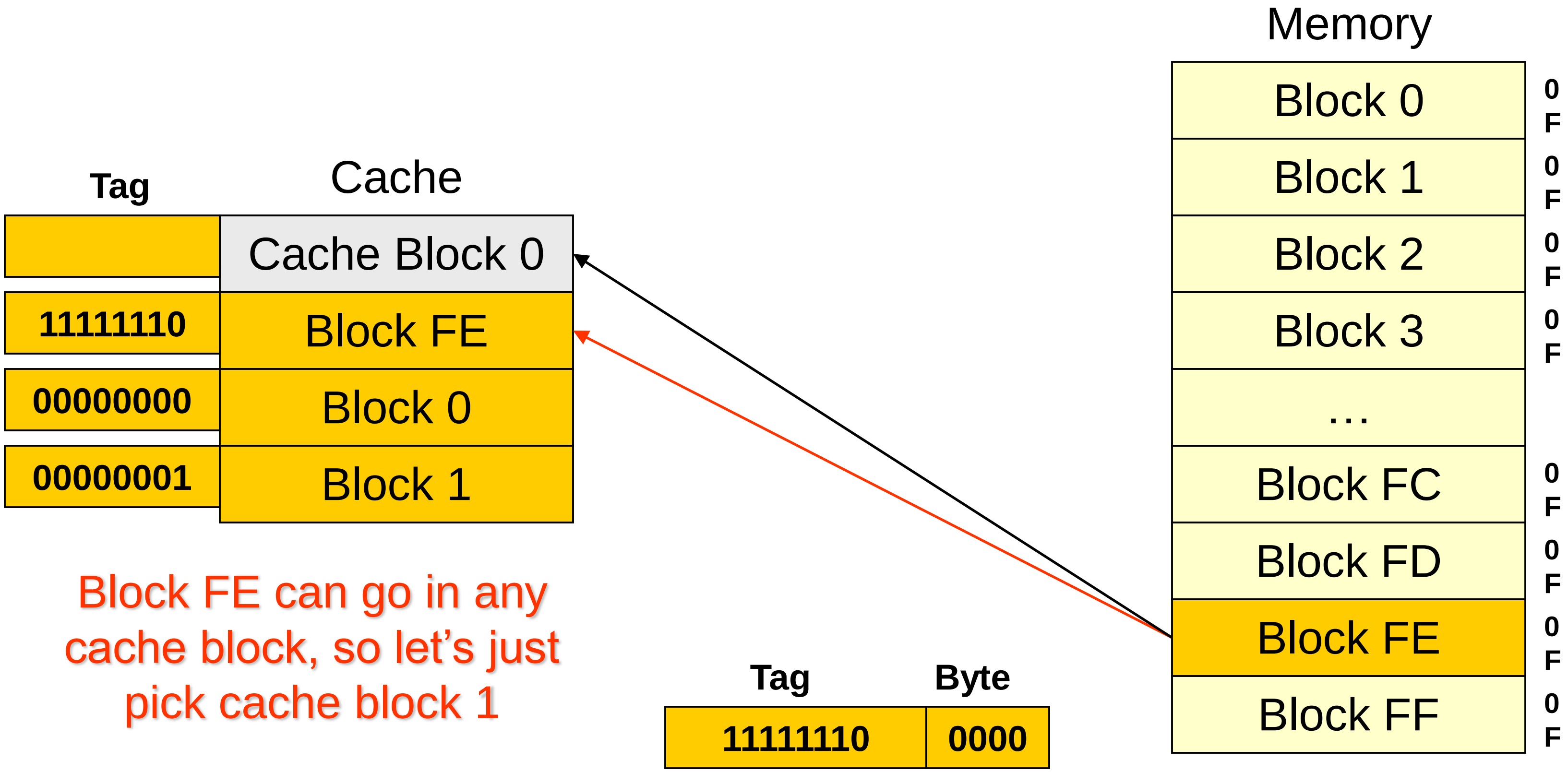
Tag	Cache
	Cache Block 0
	Cache Block 1
00000000	Block 0
00000001	Block 1

Block 1 can go in any empty cache block, so let's just pick cache block 3

Tag	Byte
00000001	0000

Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no mapping)



Memory

Block 0 0 F
Block 1 0 F
Block 2 0 F
Block 3 0 F
…
Block FC 0 F
Block FD 0 F
Block FE 0 F
Block FF 0 F

Cache

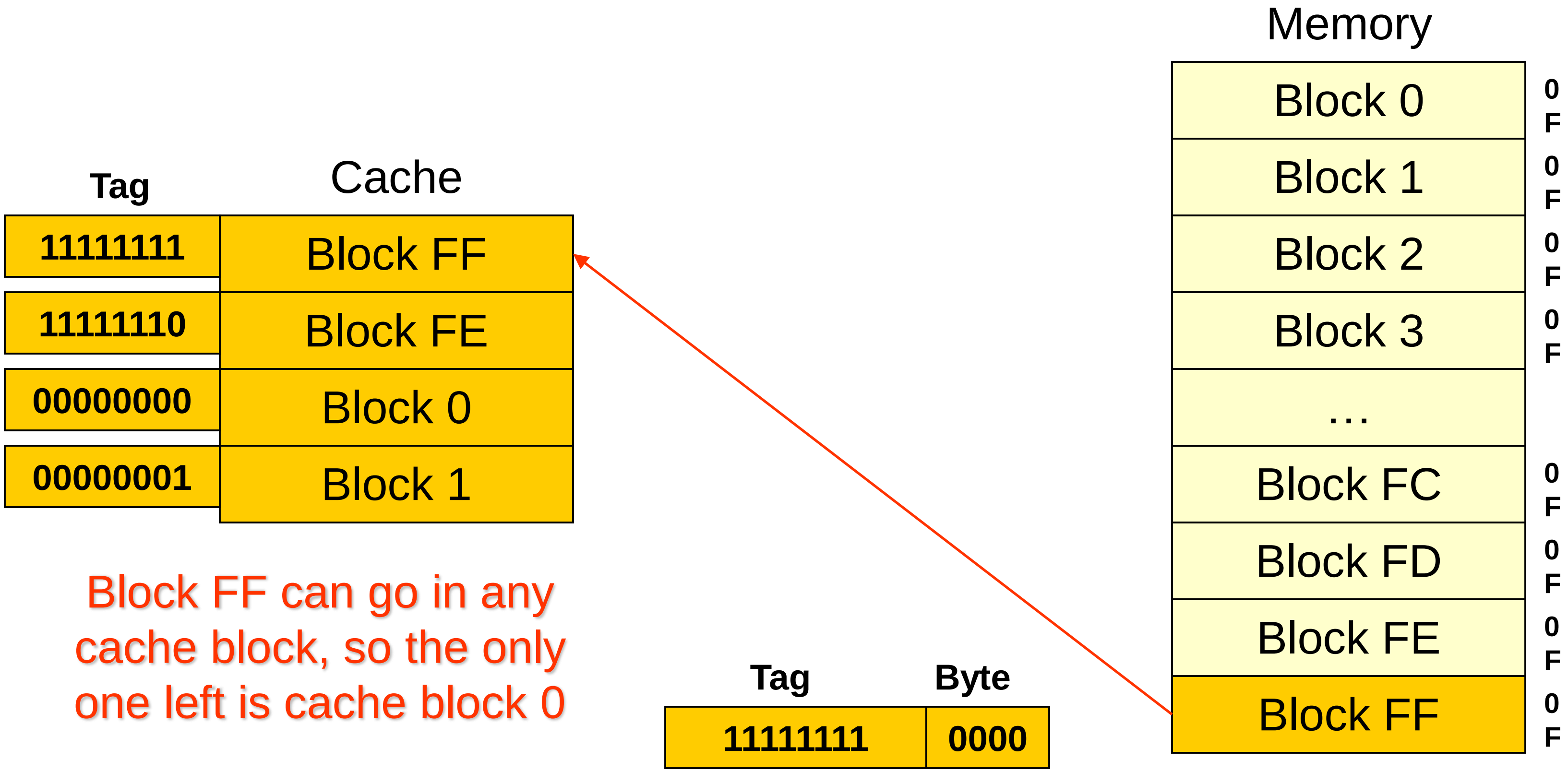
Tag	Cache
	Cache Block 0
11111110	Block FE
00000000	Block 0
00000001	Block 1

Block FE can go in any cache block, so let's just pick cache block 1

Tag	Byte
11111110	0000

Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no mapping)



Cache

Tag	
11111111	Block FF
11111110	Block FE
00000000	Block 0
00000001	Block 1

Memory

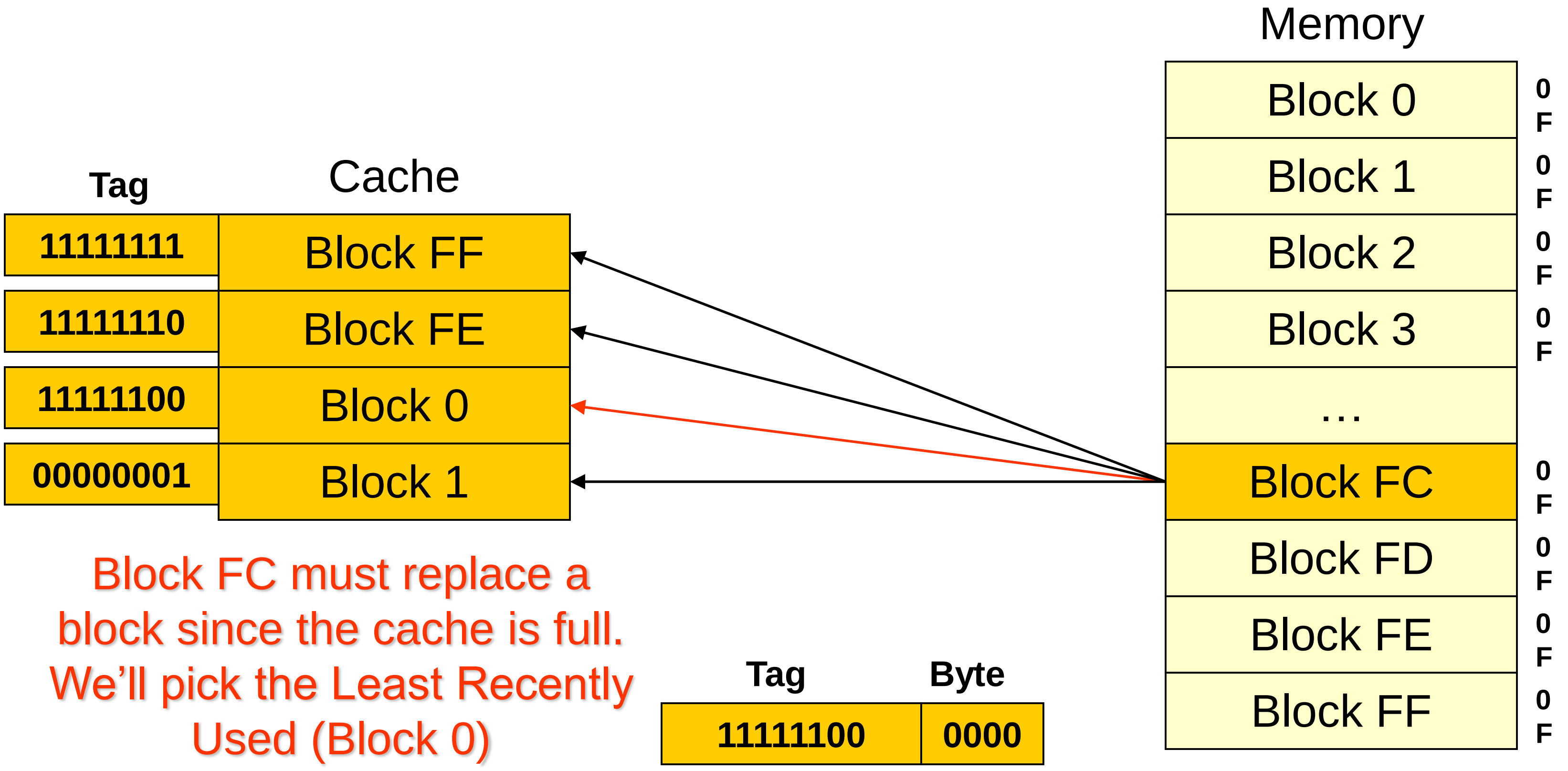
Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

Tag	Byte
11111111	0000

Block FF can go in any cache block, so the only one left is cache block 0

Fully Associative Mapping

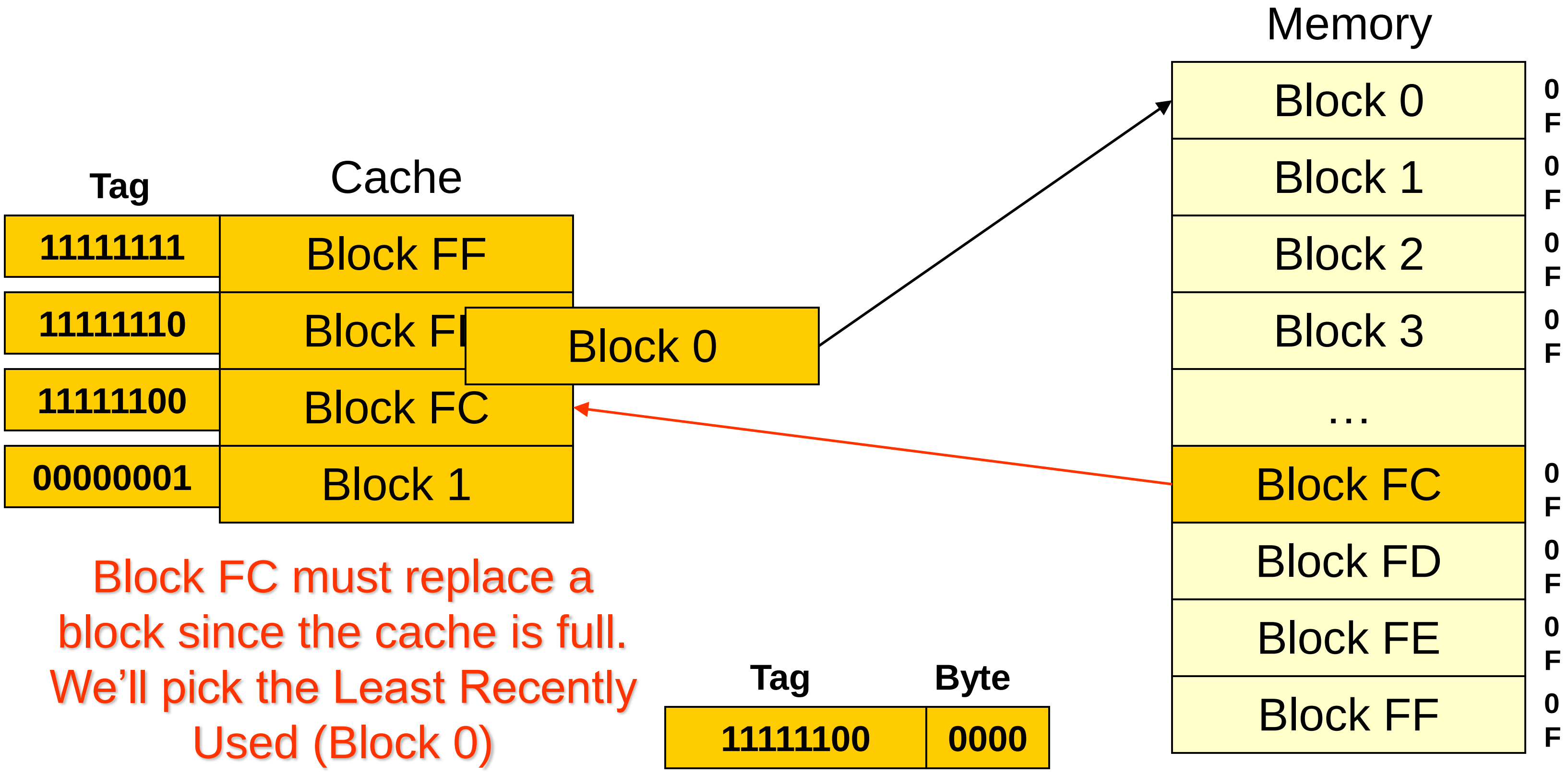
- Any block from memory can be put in any cache block (i.e. no mapping)



Block FC must replace a block since the cache is full. We'll pick the Least Recently Used (Block 0)

Fully Associative Mapping

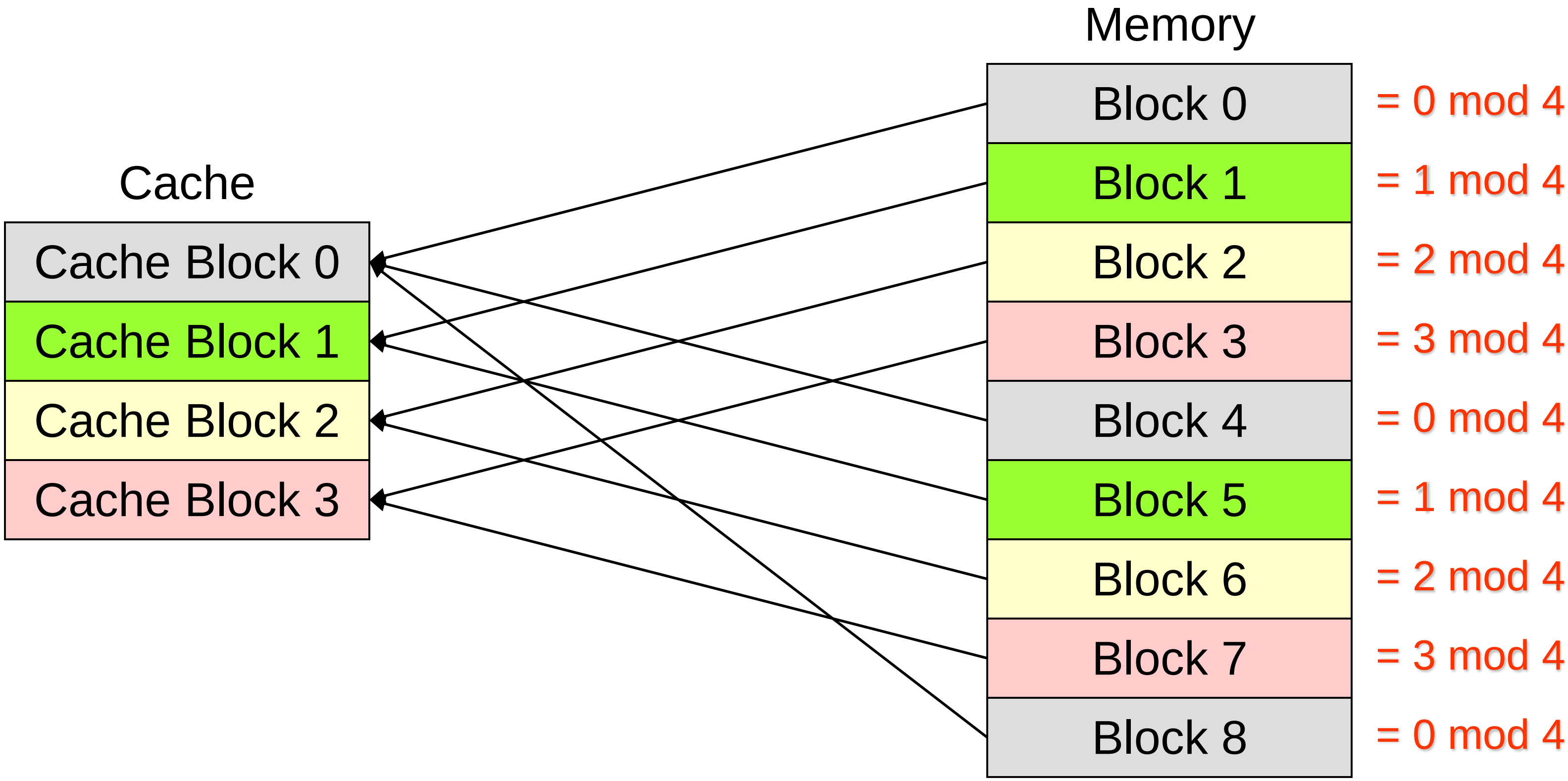
- Any block from memory can be put in any cache block (i.e. no mapping)



Block FC must replace a block since the cache is full. We'll pick the Least Recently Used (Block 0)

Direct Mapping

- Each block from memory can only be put in one location
- MM block i maps to cache block i mod n

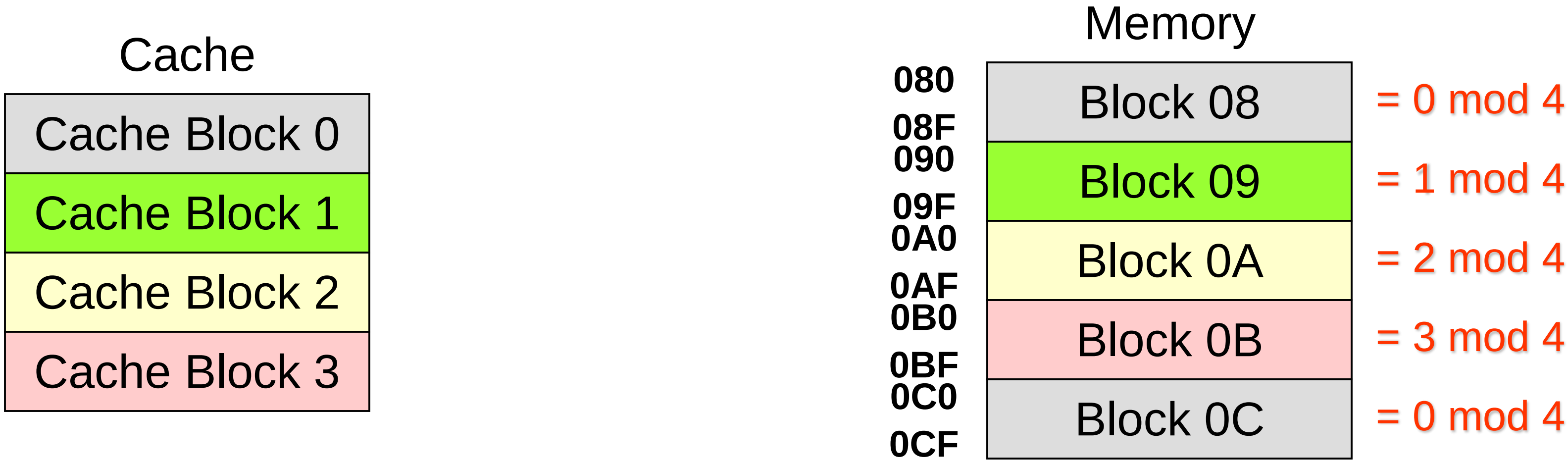


Direct Mapping Implementation

- 12-bit address:
  - 16 bytes per block => 4 LSB’s used to determine the desired byte/word offset within the block
  - 4 = 2^2 possible blocks => 2 bits to determine cache location (i.e. hash function => use these 2 bits of address)
  - Tag = Upper 6 bits used to identify the block in the cache (identifies between blocks that map to the same bucket (block 0, 4, 8, etc.)

Address = 080

Tag	Block	Byte
000010	00	0000



Cache

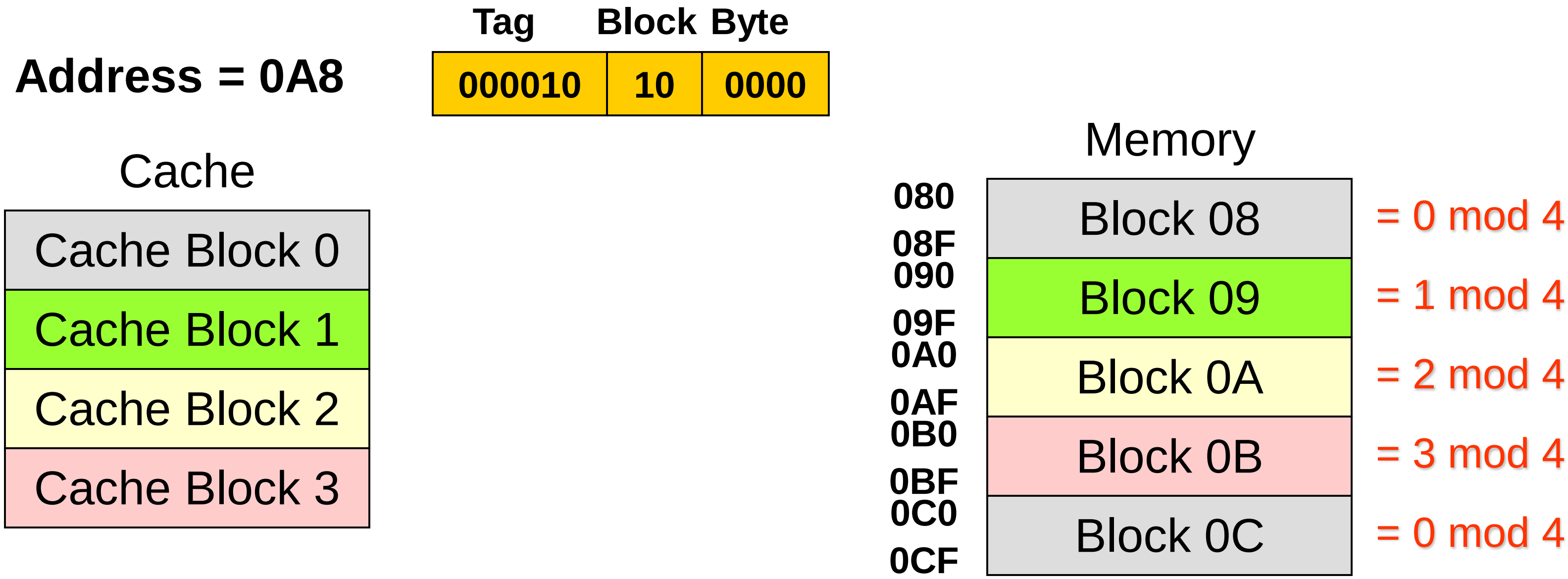
Cache Block 0
Cache Block 1
Cache Block 2
Cache Block 3

Memory

080 08F	Block 08	= 0 mod 4
090 09F	Block 09	= 1 mod 4
0A0 0AF	Block 0A	= 2 mod 4
0B0 0BF	Block 0B	= 3 mod 4
0C0 0CF	Block 0C	= 0 mod 4

Direct Mapping Implementation

- 12-bit address:
  - 16 bytes per block => 4 LSB’s used to determine the desired byte/word offset within the block
  - 4 = 2^2 possible blocks => 2 bits to determine cache location (i.e. hash function => use these 2 bits of address)
  - Tag = Upper 6 bits used to identify the block in the cache (identifies between blocks that map to the same bucket (block 0, 4, 8, etc.)



Address = 0A8

Tag	Block	Byte
000010	10	0000

Cache

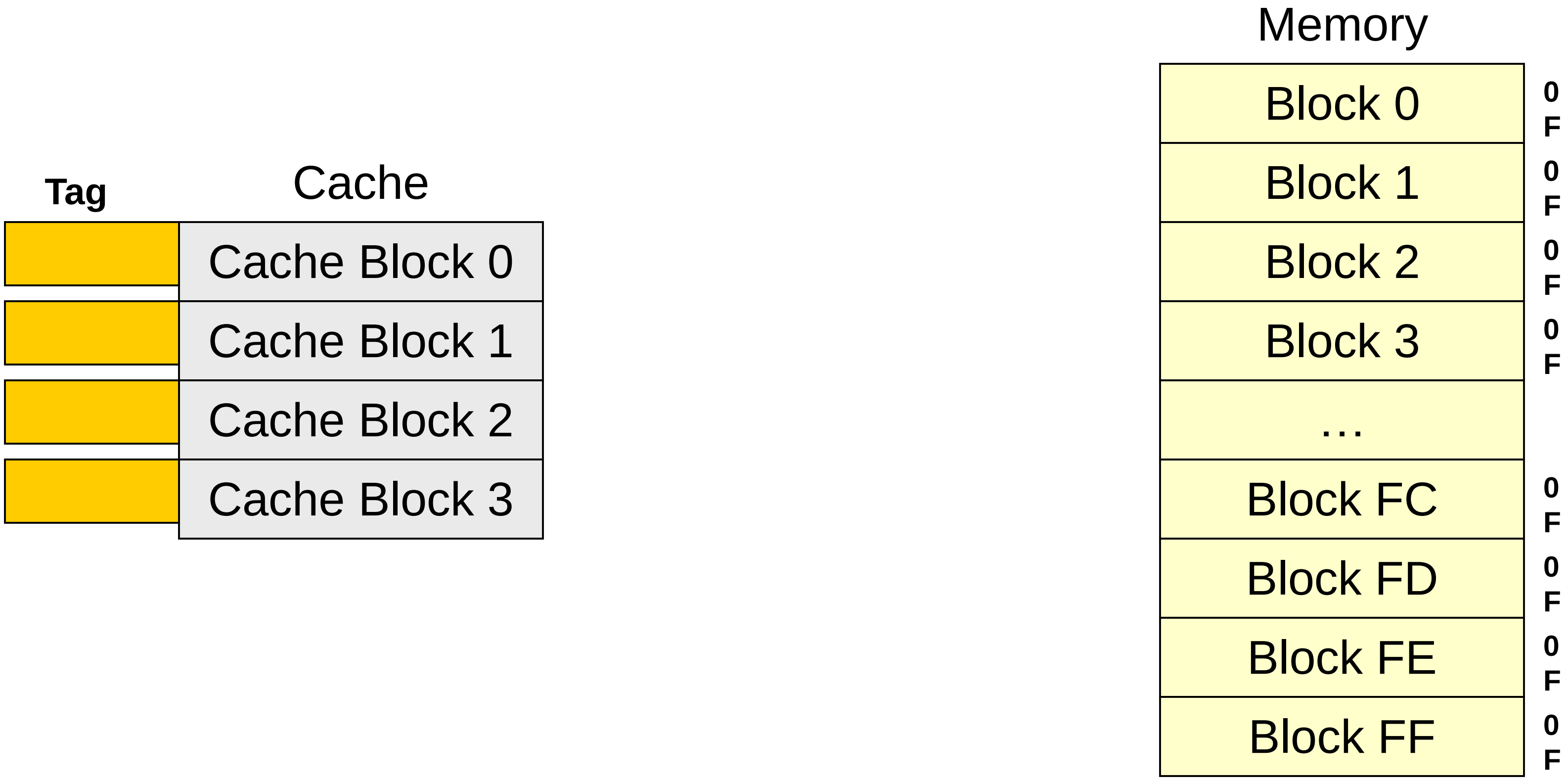
Cache
Cache Block 0
Cache Block 1
Cache Block 2
Cache Block 3

Memory

Address	Memory	
080 08F	Block 08	= 0 mod 4
090 09F	Block 09	= 1 mod 4
0A0 0AF	Block 0A	= 2 mod 4
0B0 0BF	Block 0B	= 3 mod 4
0C0 0CF	Block 0C	= 0 mod 4

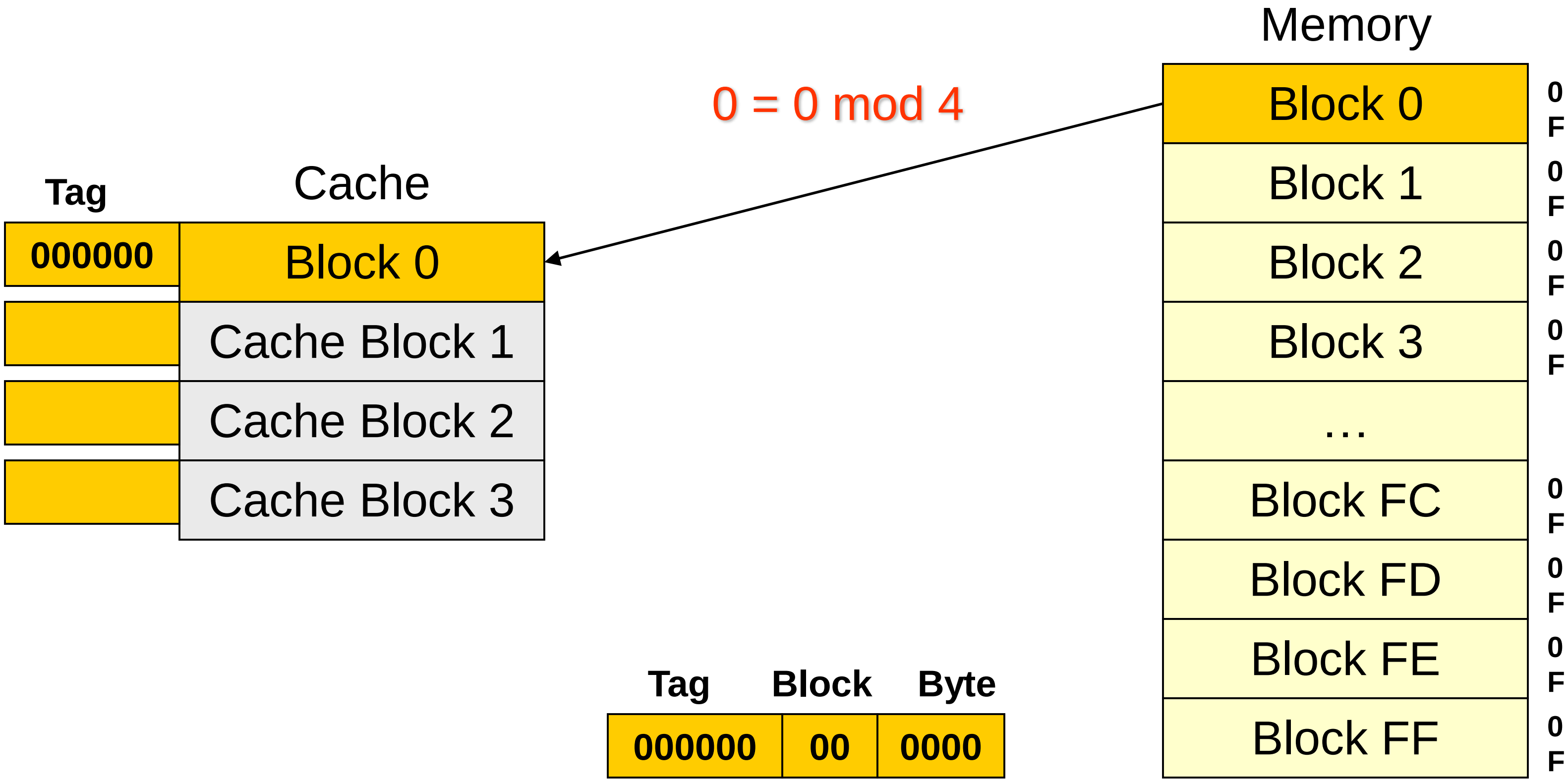
Direct Mapping

- Each block from memory can only be put in one location
- MM block i maps to cache block i mod n



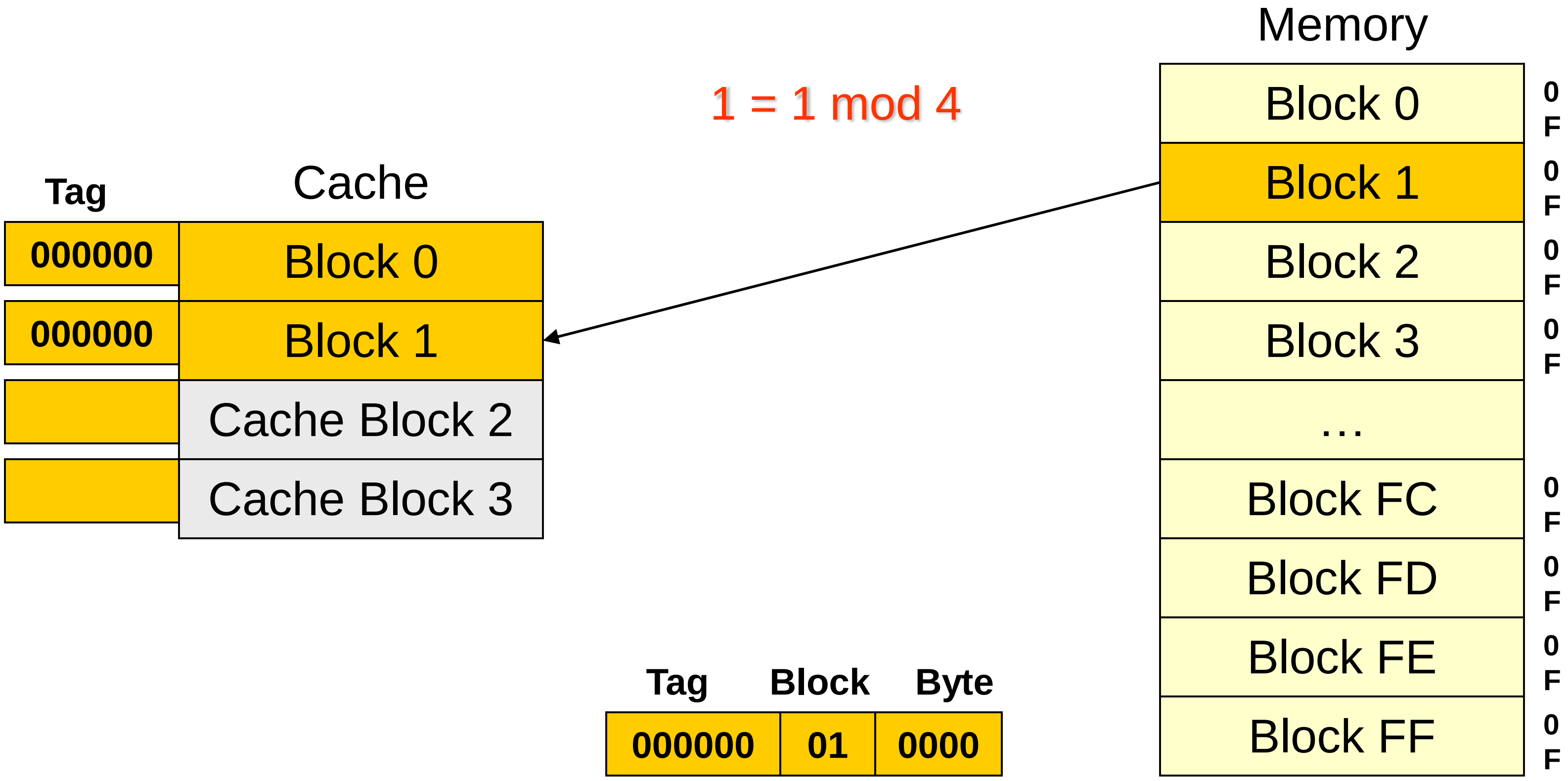
Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



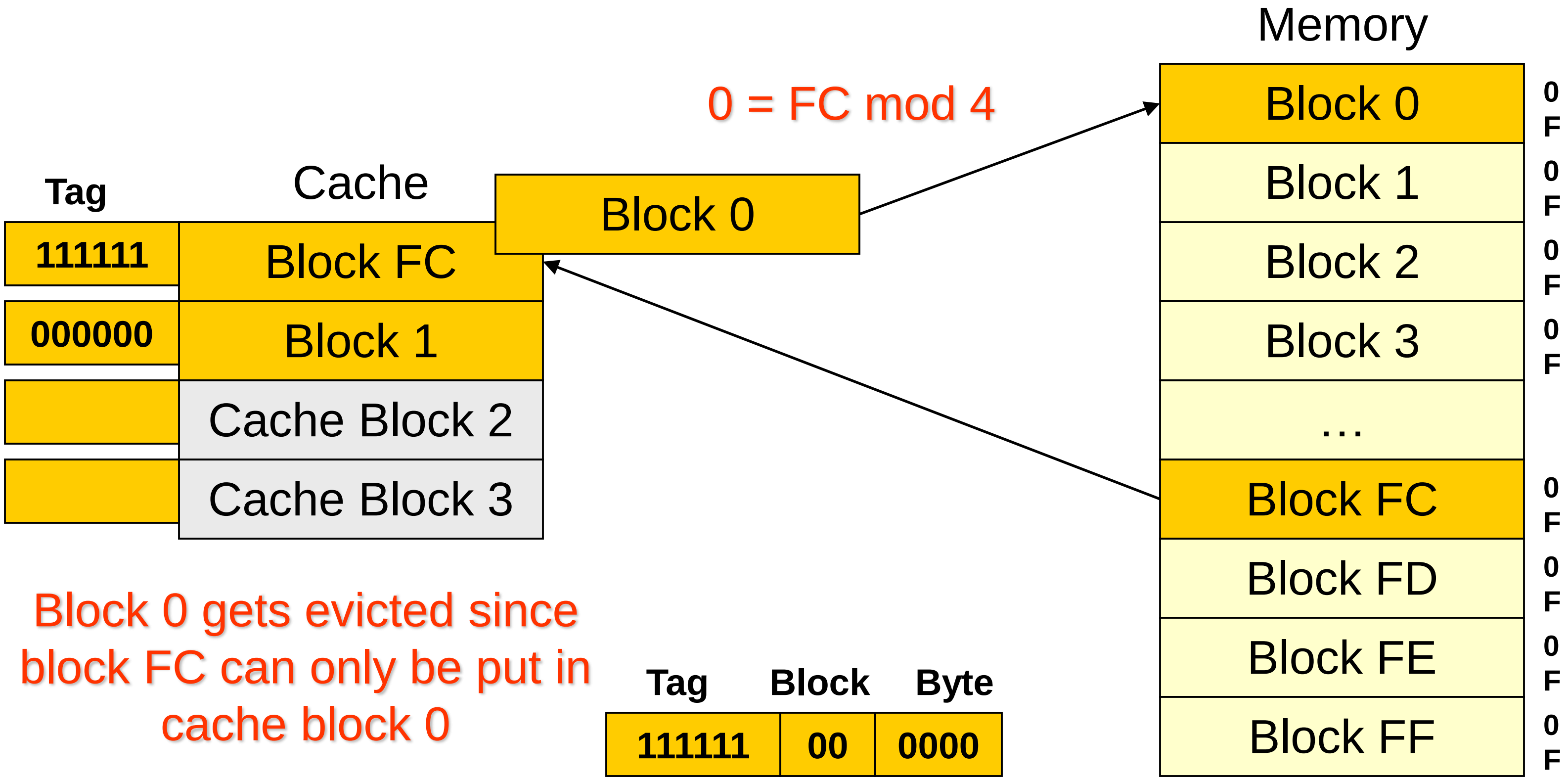
Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



Memory

0 = FC mod 4

Tag	Cache
111111	Block FC
000000	Block 1
	Cache Block 2
	Cache Block 3

Block 0

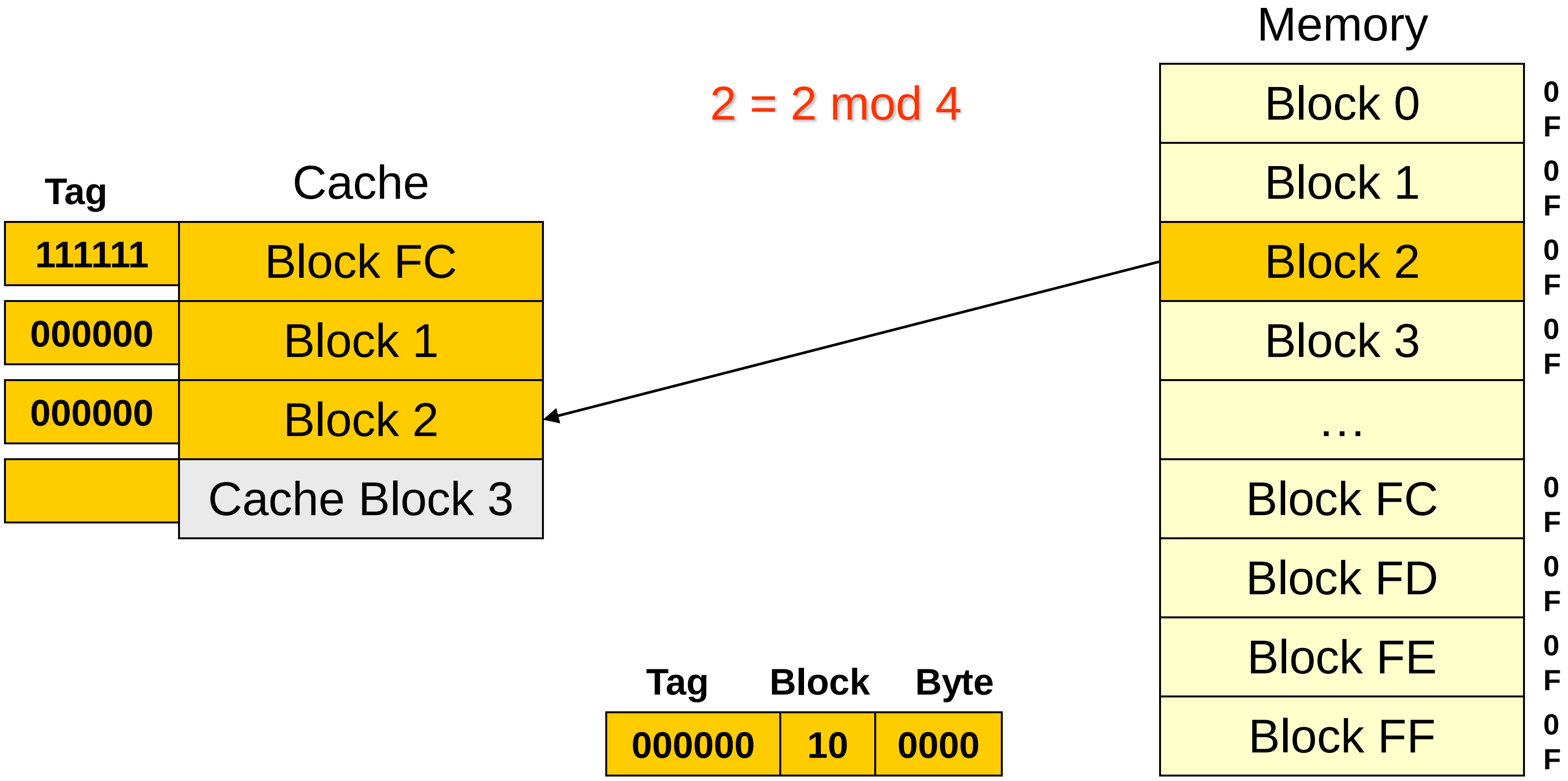
Memory	
Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

Block 0 gets evicted since block FC can only be put in cache block 0

Tag	Block	Byte
111111	00	0000

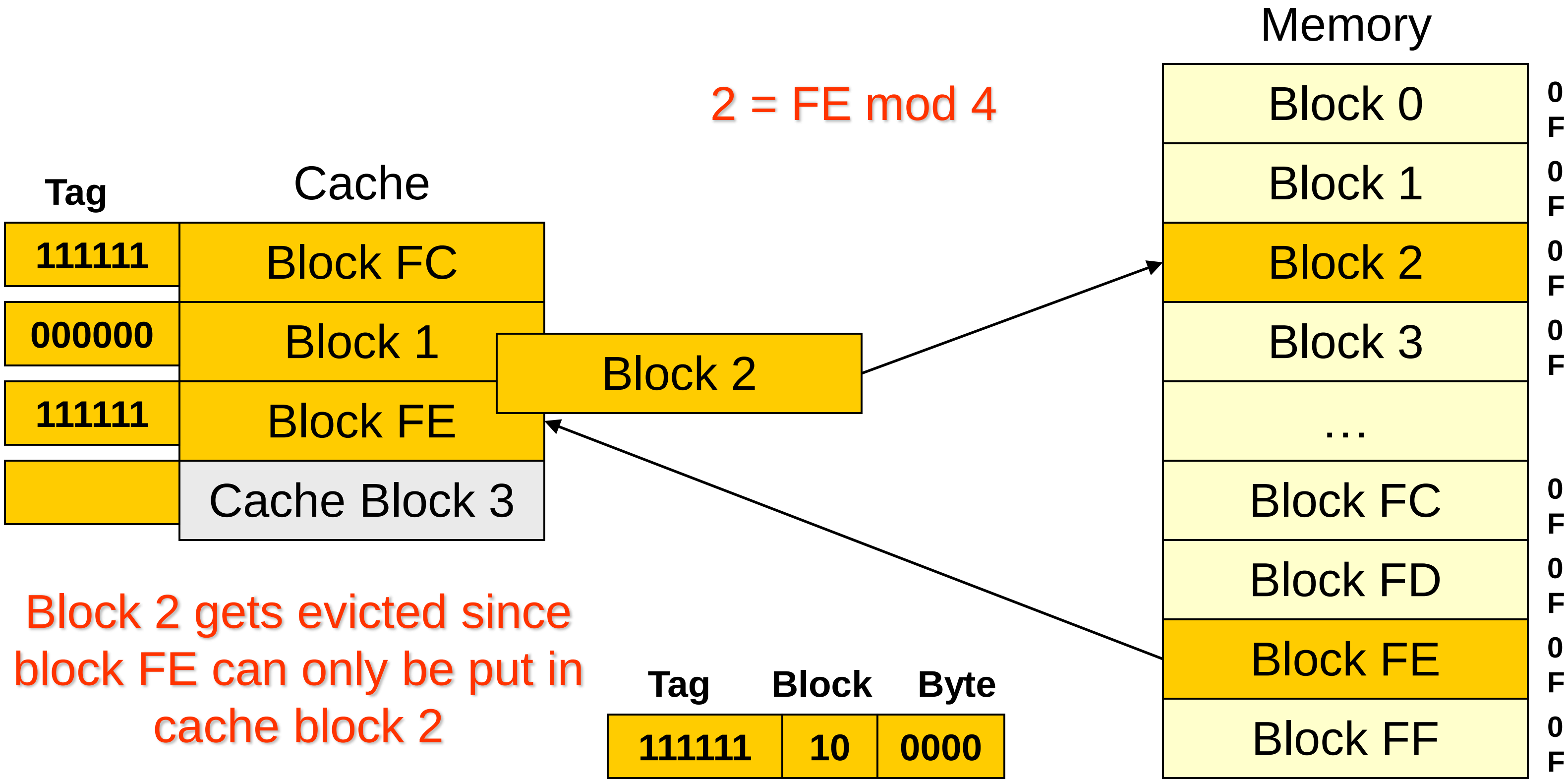
Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



Direct Mapping

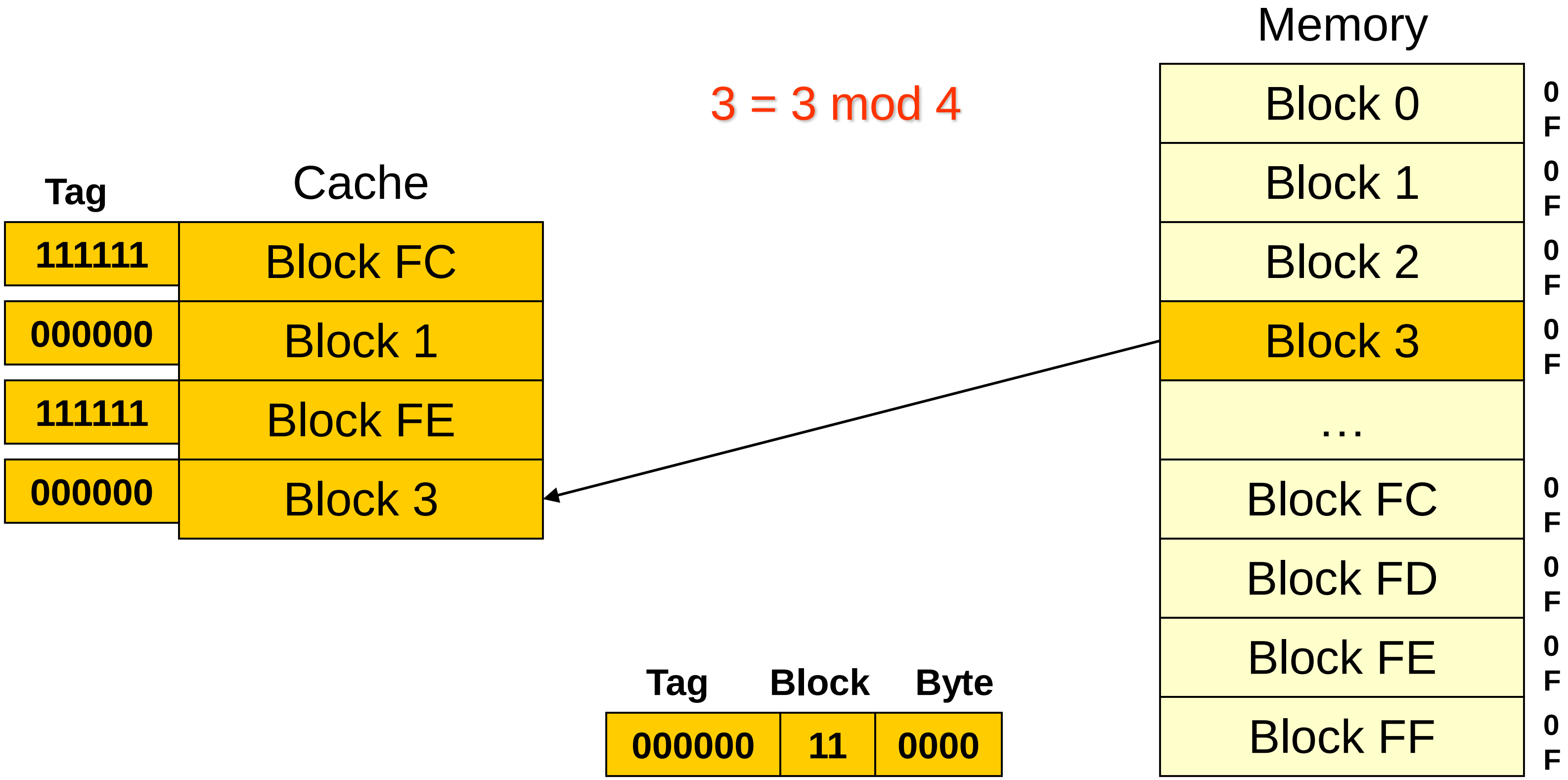
- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



Block 2 gets evicted since block FE can only be put in cache block 2

Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



3 = 3 mod 4

Cache

Tag	
111111	Block FC
000000	Block 1
111111	Block FE
000000	Block 3

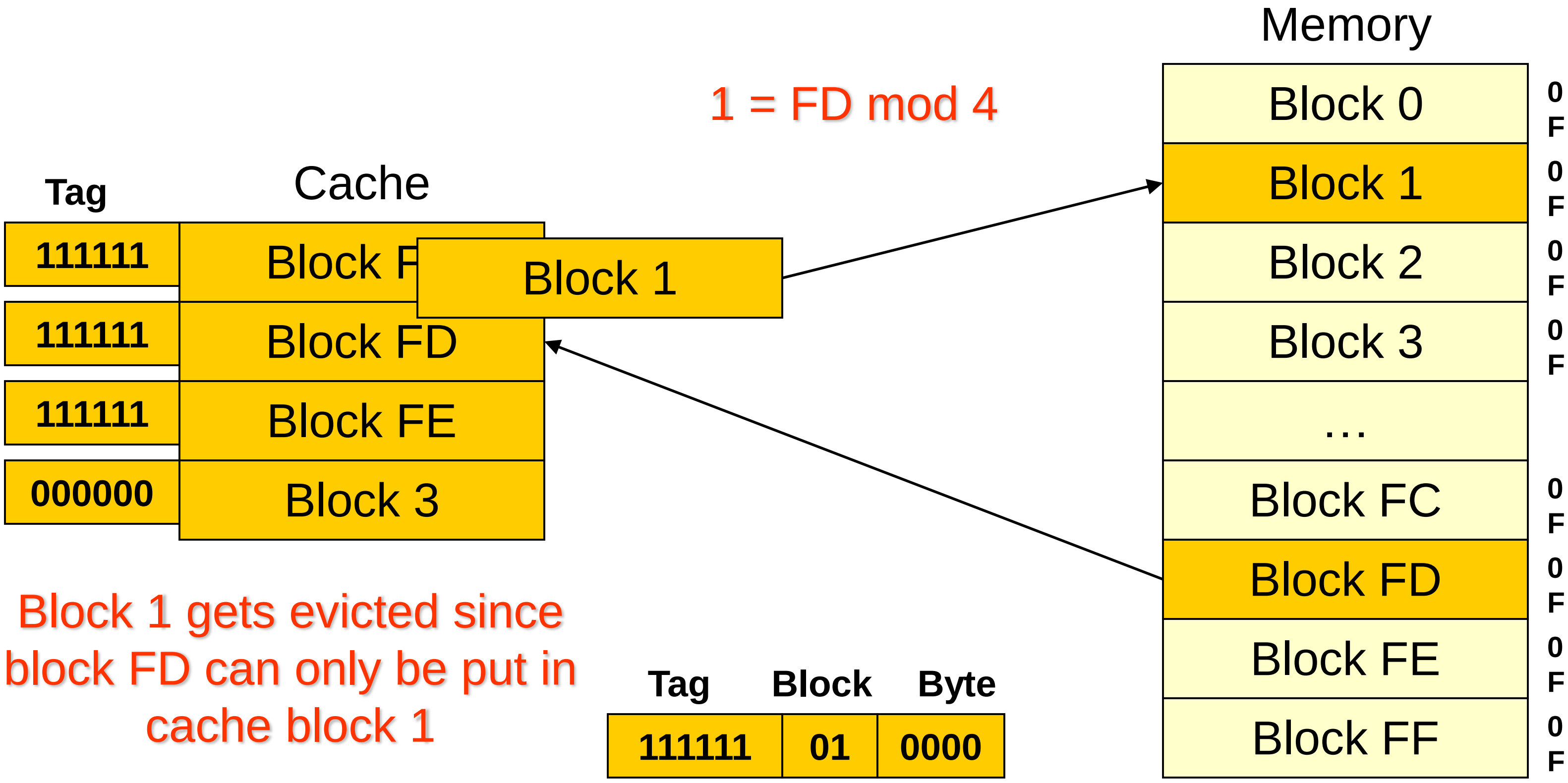
Memory

Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

Tag	Block	Byte
000000	11	0000

Direct Mapping

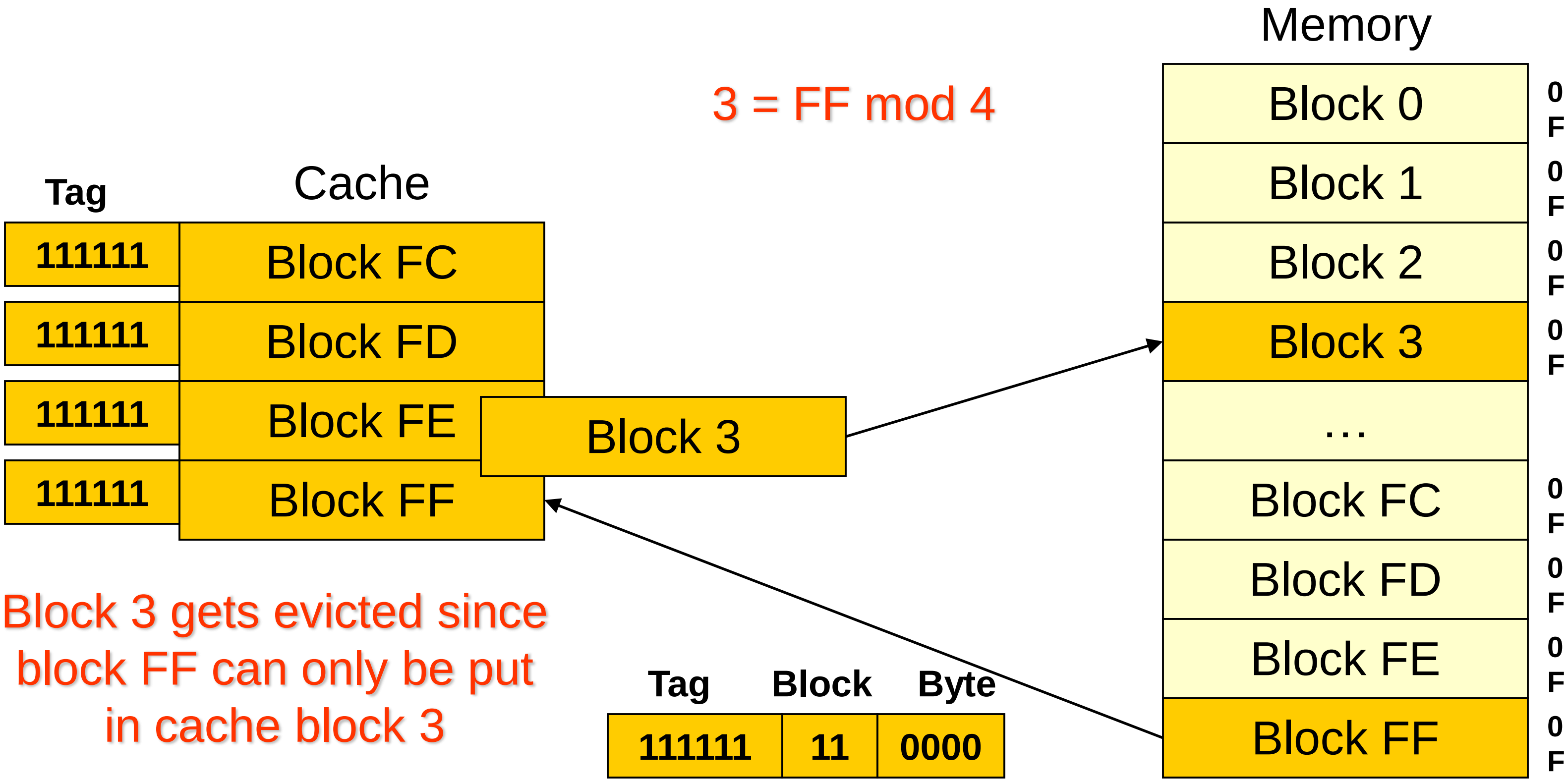
- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



Block 1 gets evicted since block FD can only be put in cache block 1

Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



3 = FF mod 4

Tag

Cache

Tag	Cache
111111	Block FC
111111	Block FD
111111	Block FE
111111	Block FF

Block 3

Memory

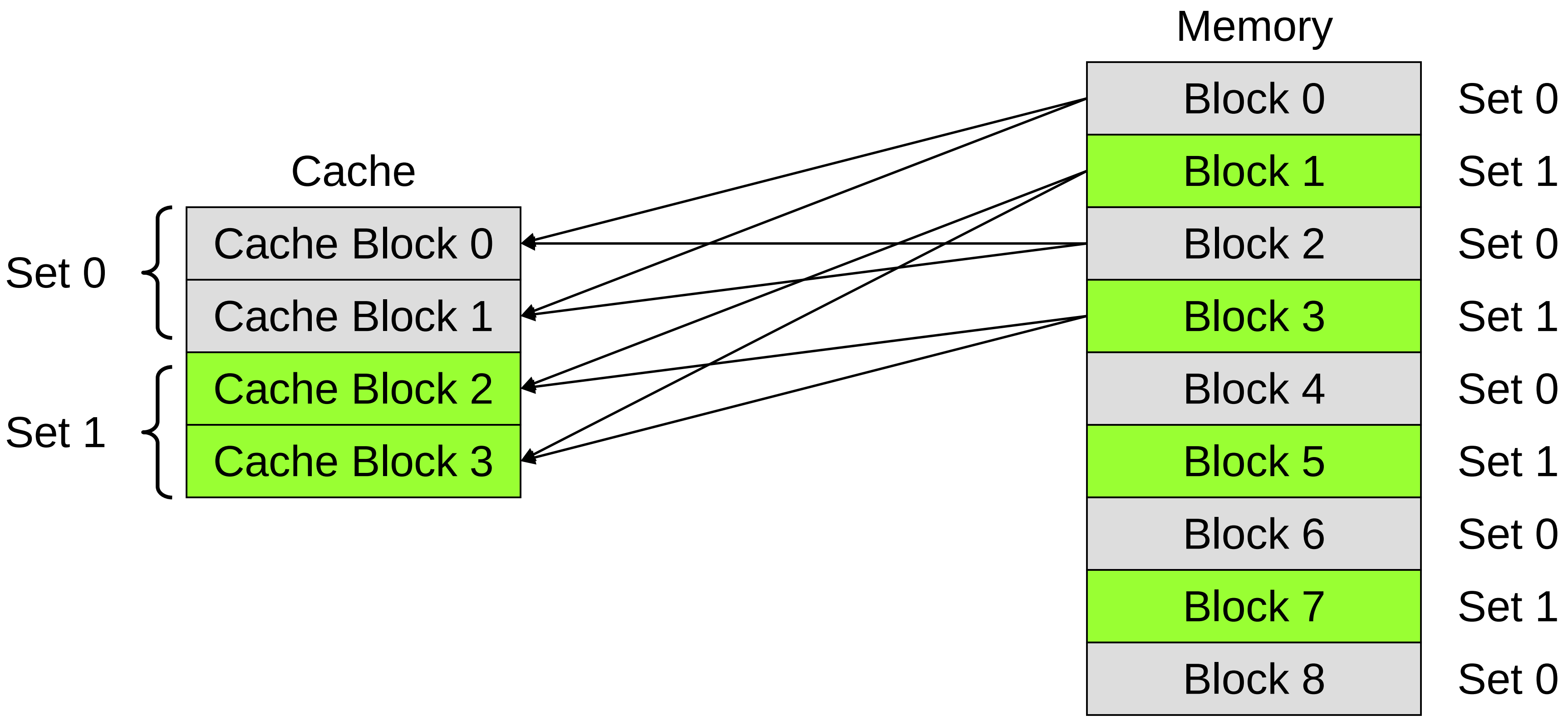
Memory	
Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

Block 3 gets evicted since block FF can only be put in cache block 3

Tag	Block	Byte
111111	11	0000

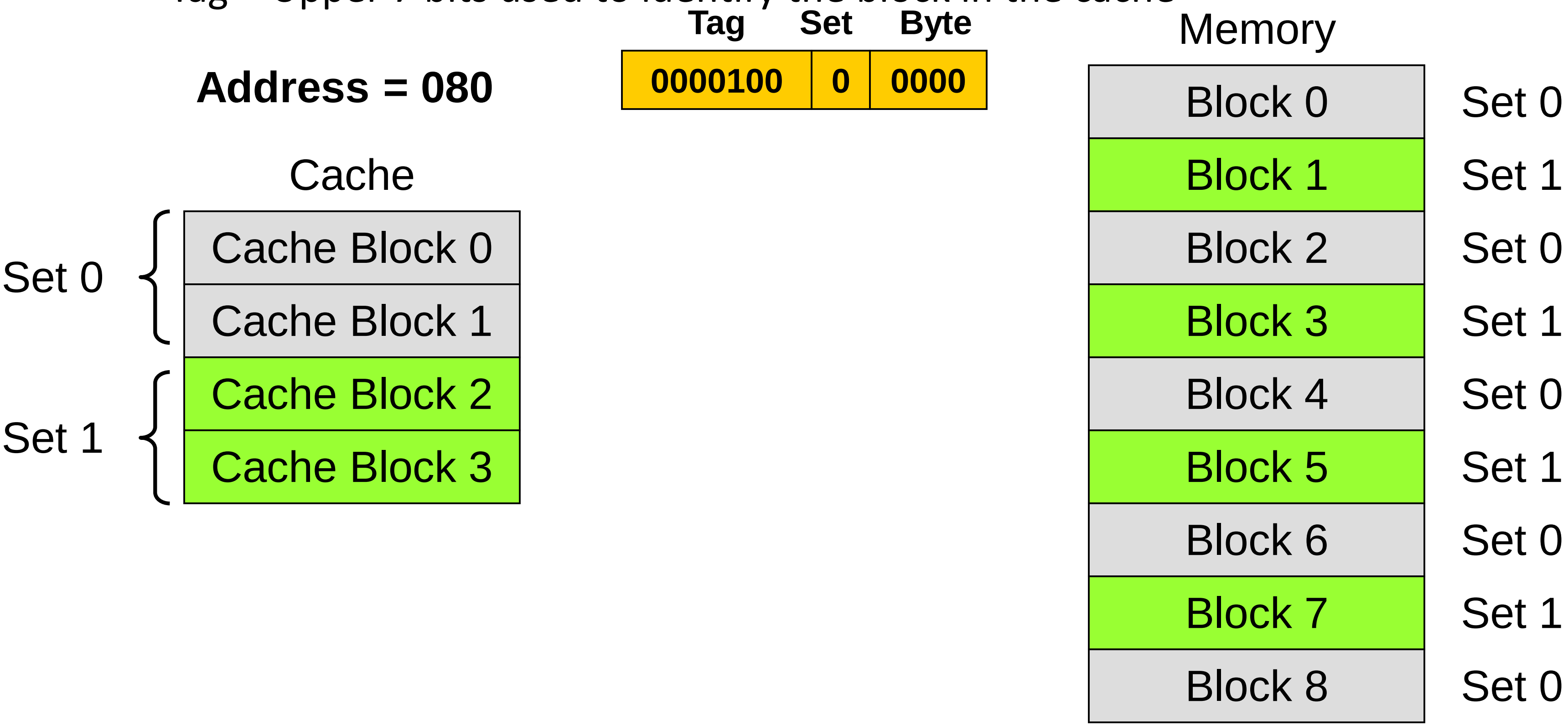
Set-Associative Mapping

- Blocks from set i can map into any cache block from set i



Set-Associative Mapping

- 12-bit address:
  - 16 bytes per block => 4 LSB's used to determine the desired byte/word offset within the block
  - 2 = 2^1 possible sets => 1 bits to determine cache set (i.e. hash function => use this 1-bit of address)
  - Tag = Upper 7 bits used to identify the block in the cache



Address = 080

Tag	Set	Byte
0000100	**0**	**0000**

Cache

Set 0: Cache Block 0, Cache Block 1

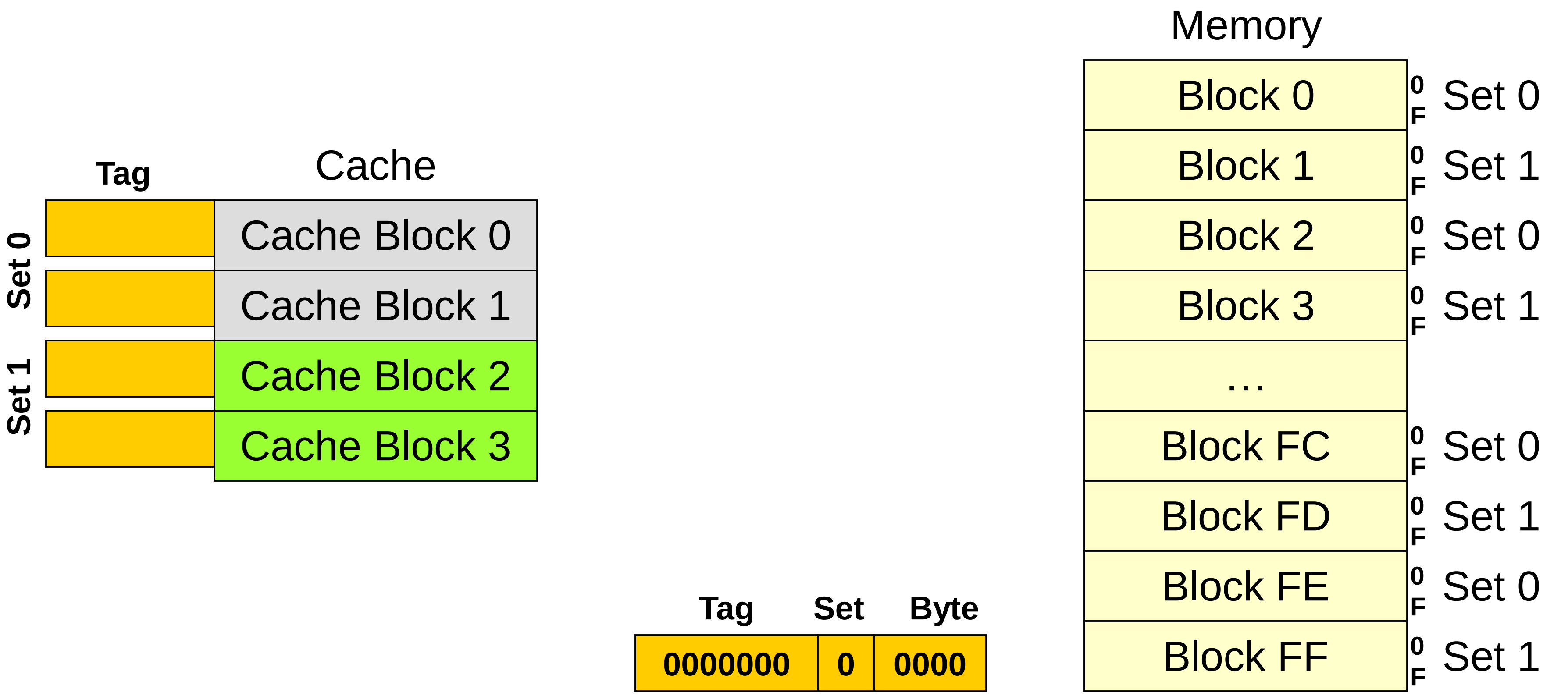
Set 1: Cache Block 2, Cache Block 3

Memory

Memory	
Block 0	Set 0
Block 1	Set 1
Block 2	Set 0
Block 3	Set 1
Block 4	Set 0
Block 5	Set 1
Block 6	Set 0
Block 7	Set 1
Block 8	Set 0

Set-Associative Mapping

- Blocks from set i can map into any cache block from set i

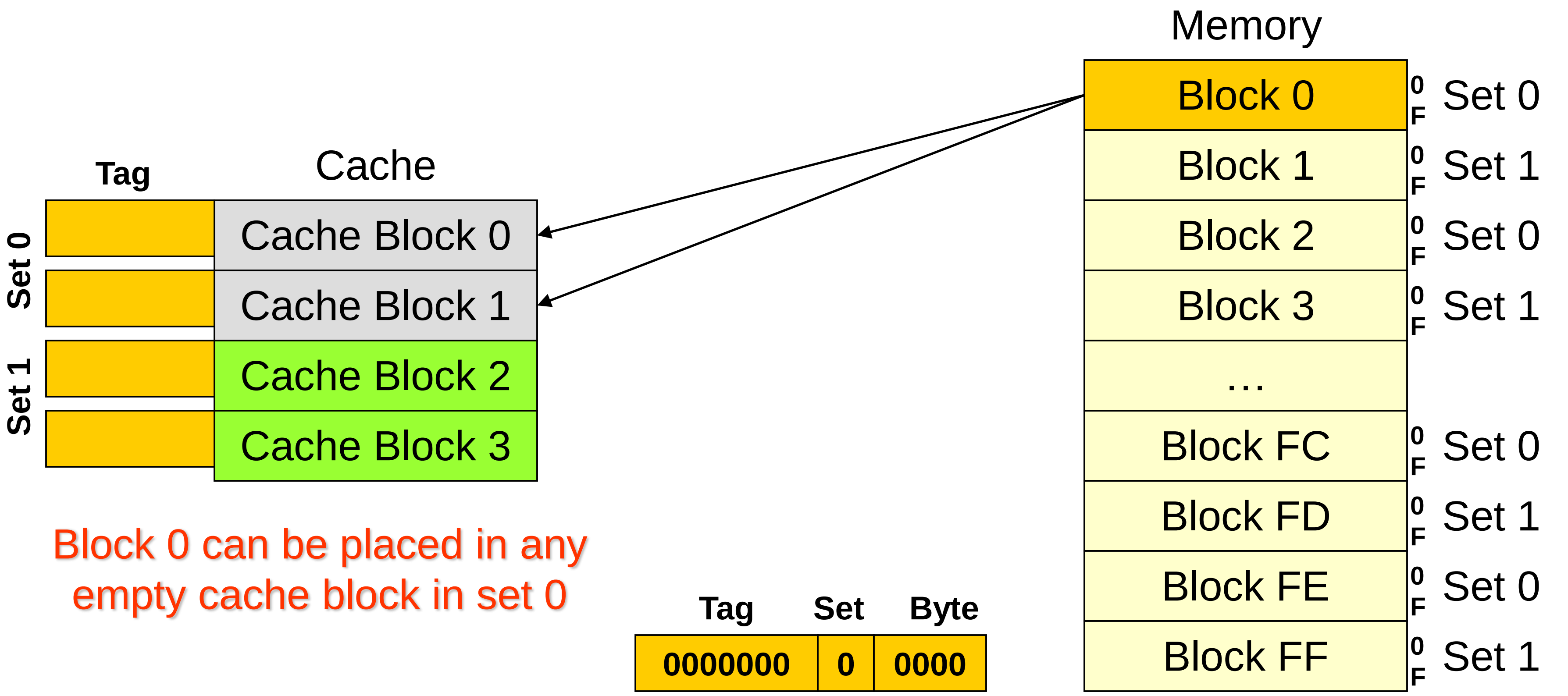


Memory		
Block 0	0 F	Set 0
Block 1	0 F	Set 1
Block 2	0 F	Set 0
Block 3	0 F	Set 1
…		
Block FC	0 F	Set 0
Block FD	0 F	Set 1
Block FE	0 F	Set 0
Block FF	0 F	Set 1

Tag	Set	Byte
0000000	0	0000

Set-Associative Mapping

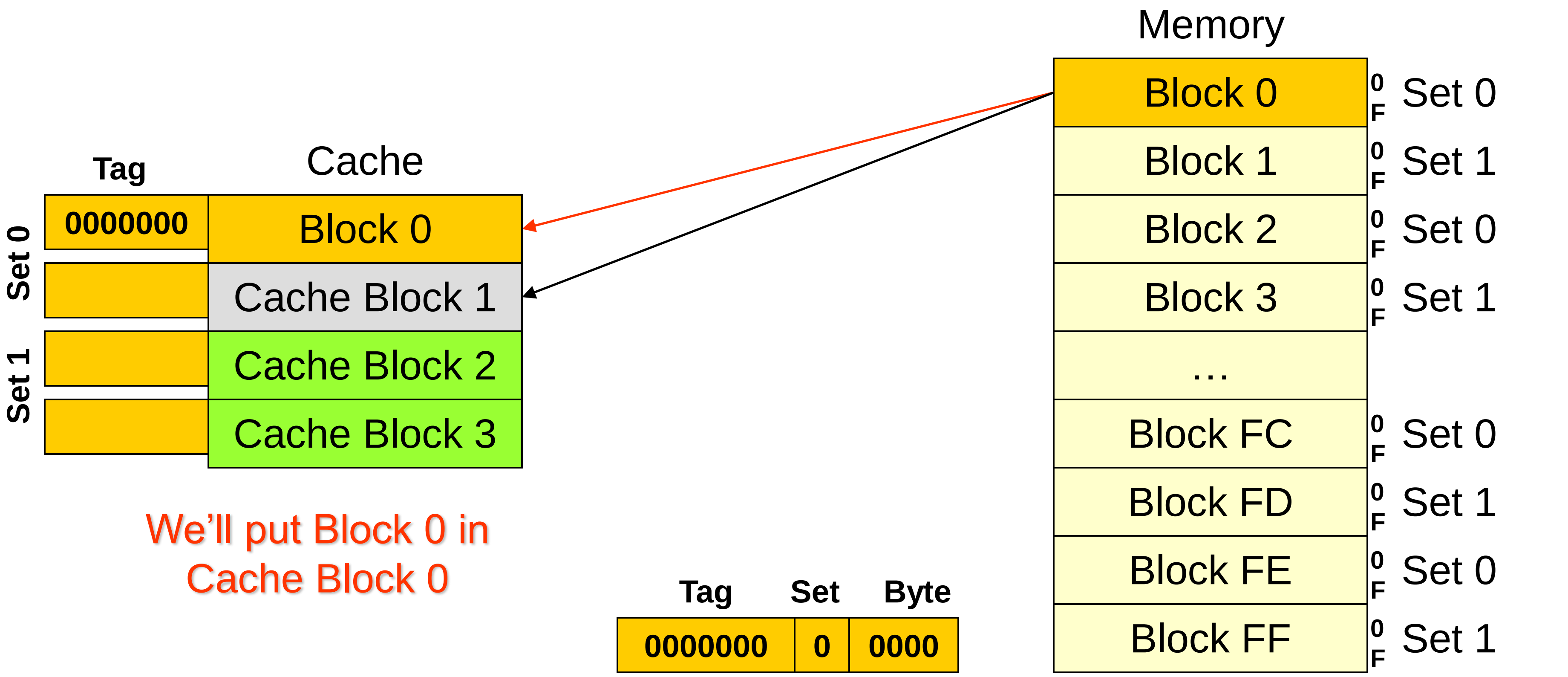
- Blocks from set i can map into any cache block from set i



Block 0 can be placed in any empty cache block in set 0

Set-Associative Mapping

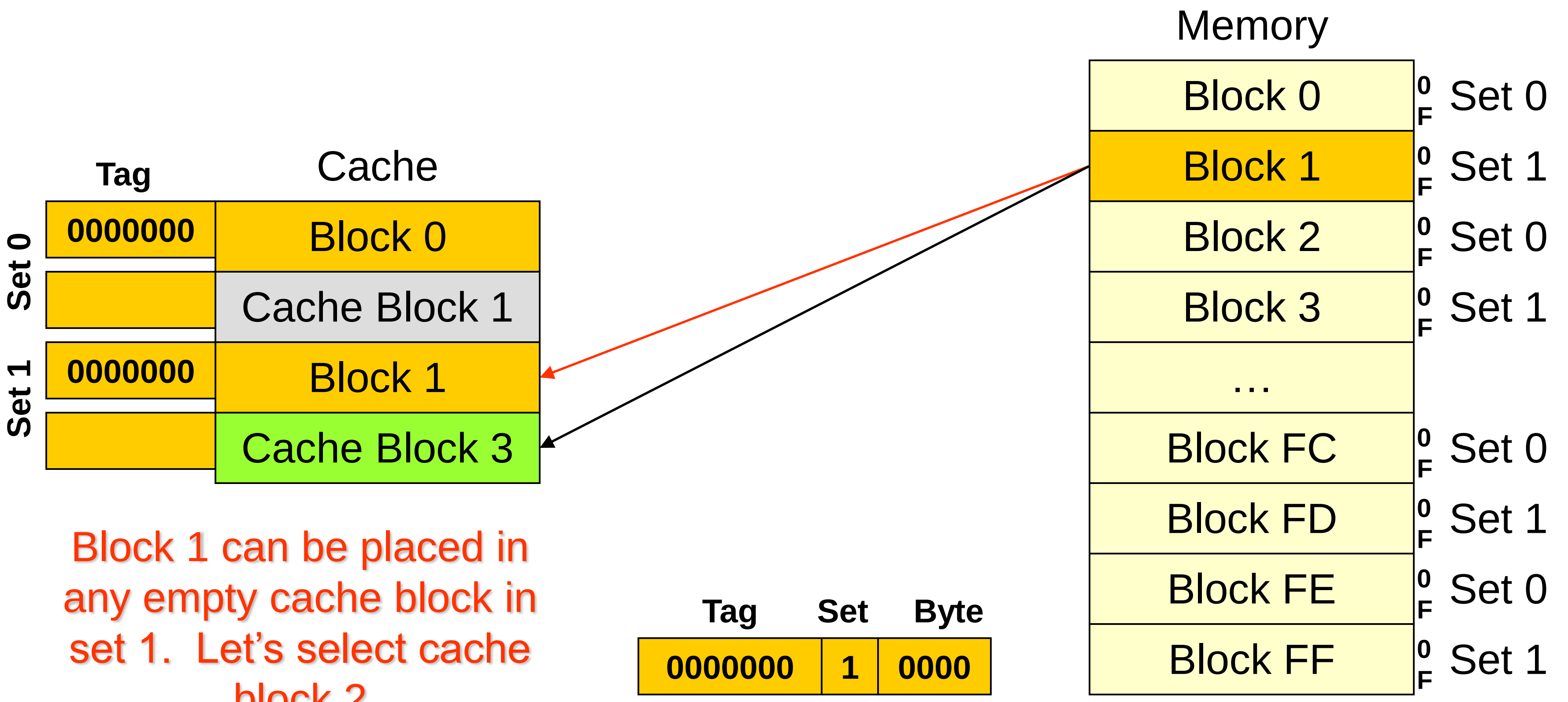
- Blocks from set i can map into any cache block from set i



We'll put Block 0 in Cache Block 0

Set-Associative Mapping

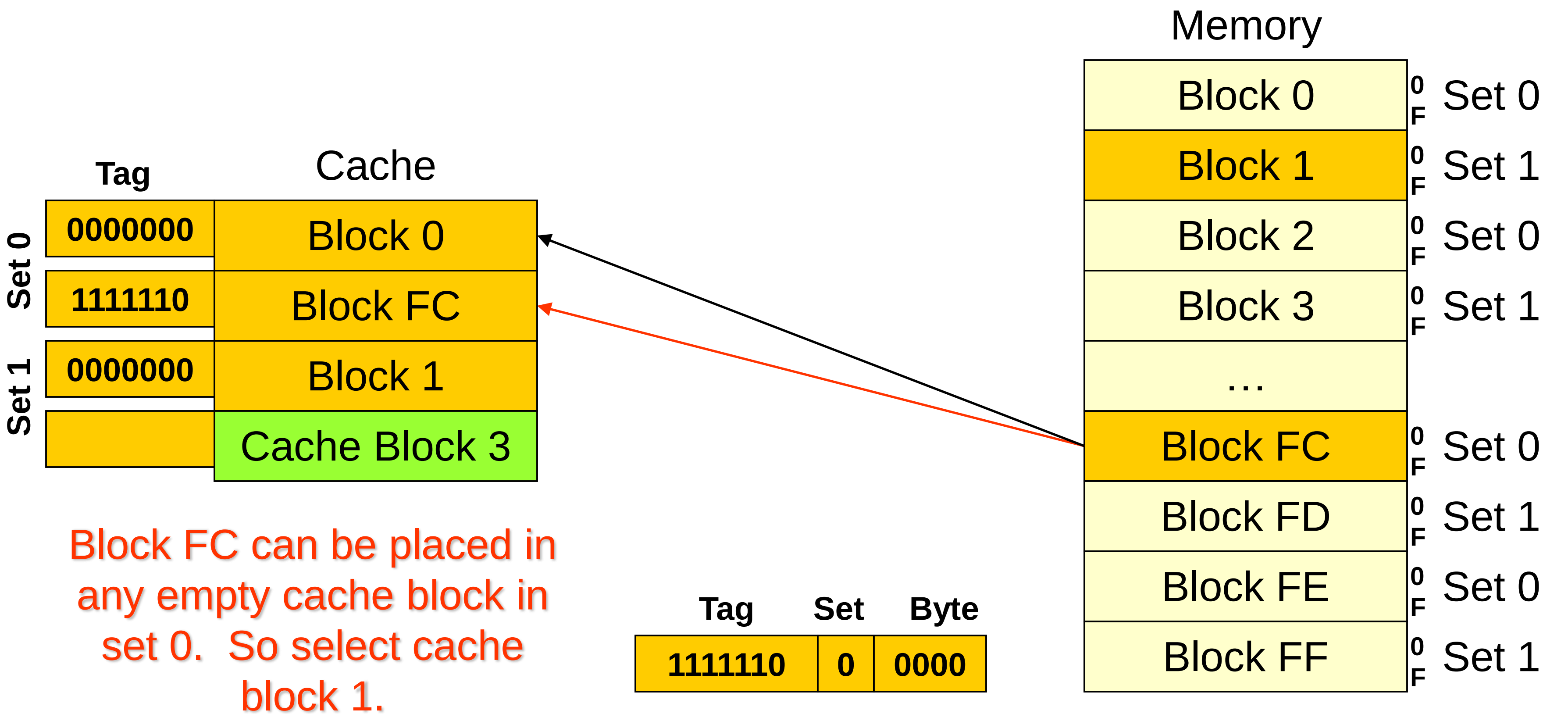
- Blocks from set i can map into any cache block from set i



Block 1 can be placed in any empty cache block in set 1. Let's select cache block 2

Set-Associative Mapping

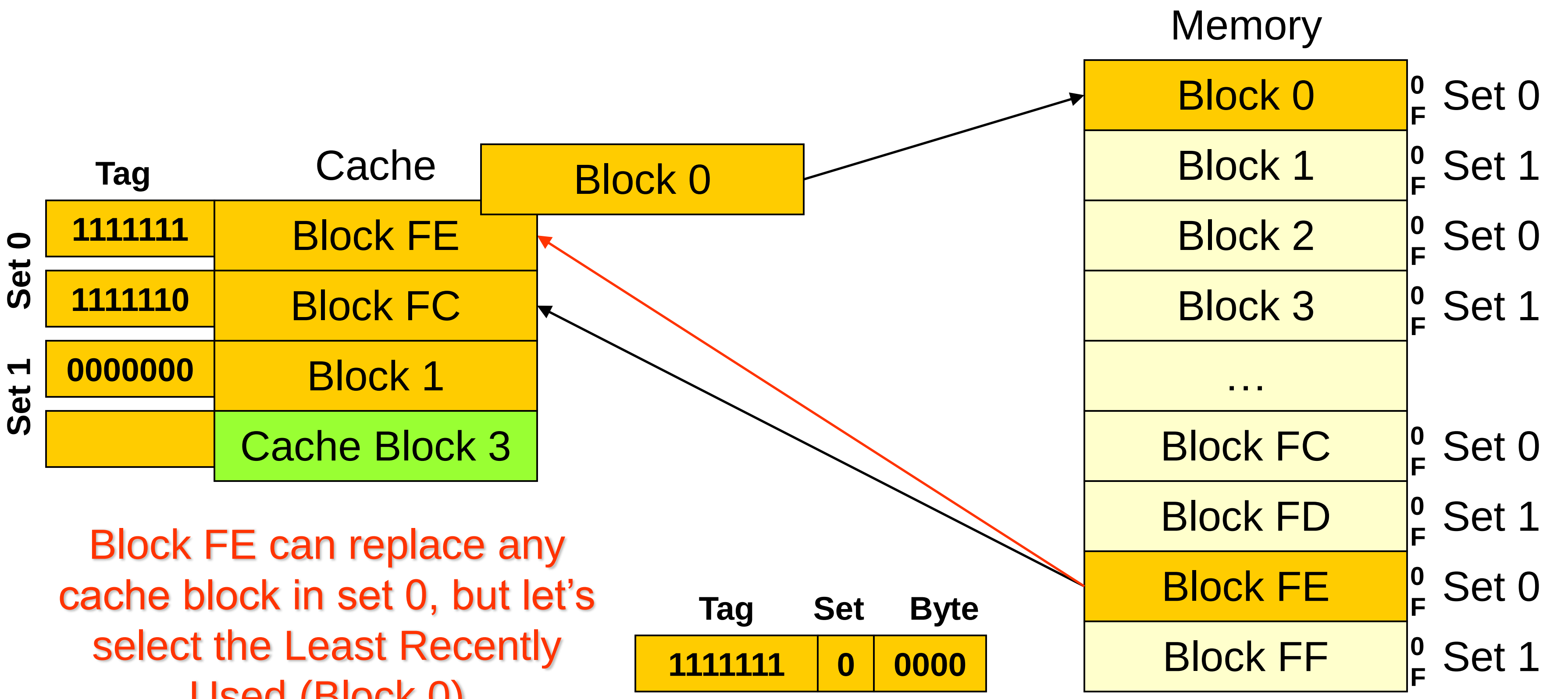
- Blocks from set i can map into any cache block from set i



Block FC can be placed in any empty cache block in set 0. So select cache block 1.

Set-Associative Mapping

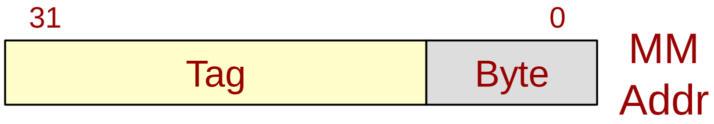
- Blocks from set i can map into any cache block from set i



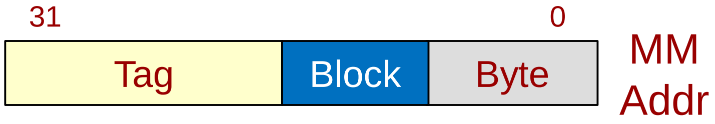
Block FE can replace any cache block in set 0, but let's select the Least Recently Used (Block 0)

Summary of Mapping Schemes

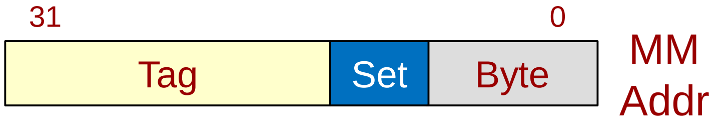
- Fully associative
  - – Most flexible (less evictions)
  - – Longest search time O(N)
- Direct-mapped cache
  - – Least flexible (more evictions)
  - – Shortest search time O(1)
- K-way Set Associative mapping
  - – Compromise
    - 1-way set associative = _________
    - N-way set associative = _________
  - – Search time is O(k) [usually small enough to be done in parallel => O(1)]



Fully Associative
No hashing…can be placed anywhere in cache. Must search N locations.

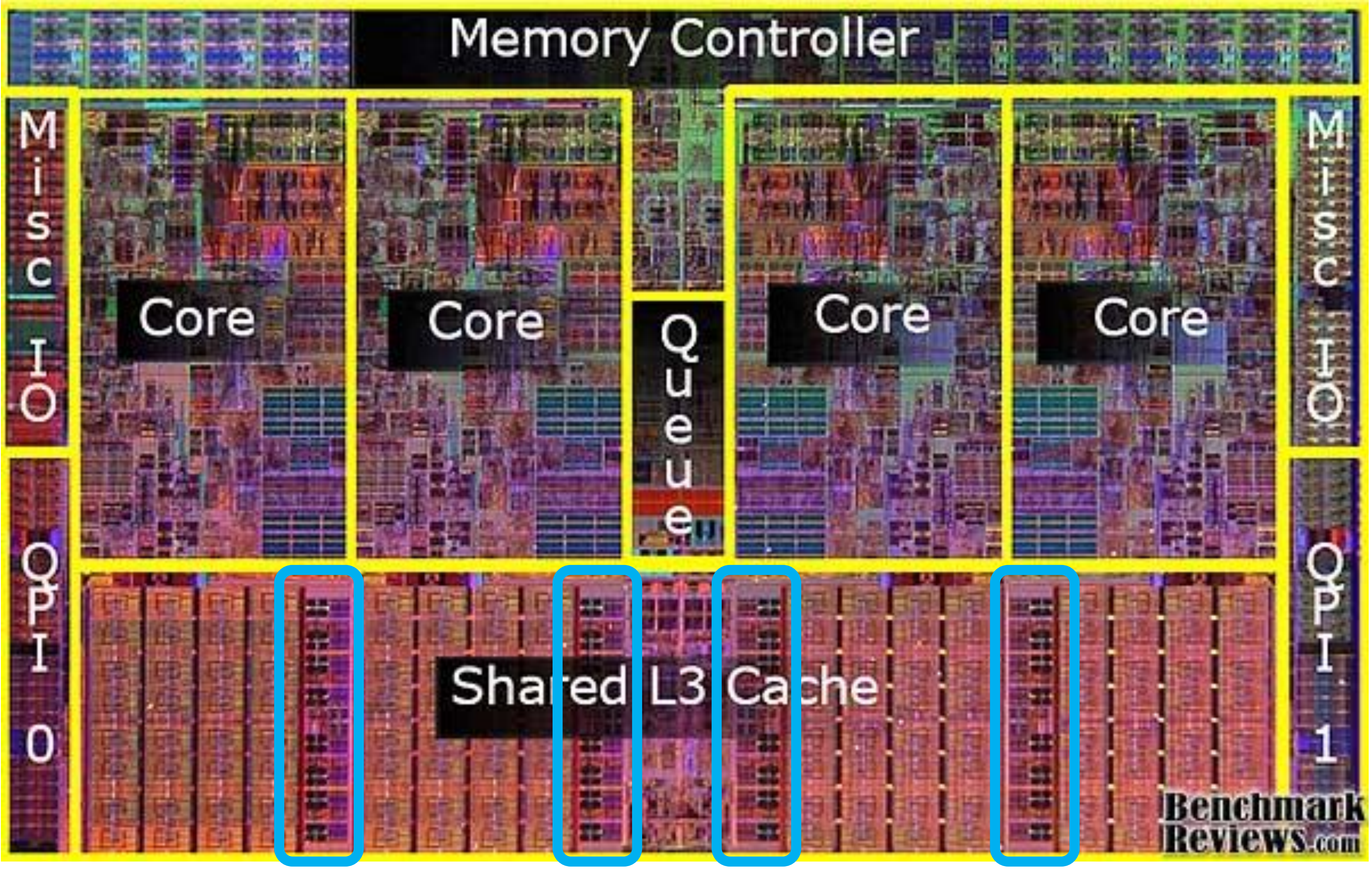


Direct Mapped Cache
h(a) = block field
Only search 1 location.



K-way Set Associative Mapping
h(a) = set field
Only search k locations

Intel Nehalem Quad Core



Cache Configurations

	AMD Opteron	Intel P4	PPC 7447a
Clock rate (2004)	2.0 GHz	3.2 GHz	1.5 – 2 GHz
Instruction Cache	64KB, 2-way SA	96 KB	32 KB, 8-way SA
Latency (clocks)	3	4	1
Data cache	64 KB, 2-way SA	8 KB, 4-way SA	32 KB, 8-way SA
Latency (clocks)	3	2	1
L1 Write Policy	Write-back	Write-through	Programmable
On-chip L2	1 MB, 16-way SA	512 KB, 8-way SA	512 KB, 8-way SA
L2 Latency	6	5	9
Block size (L1/L2)	64	64/128	32/64
L2 Write-Policy	Write-back	Write-back	Programmable

Sources: H&P, "CO&D", 3rd ed., Freescale.com,

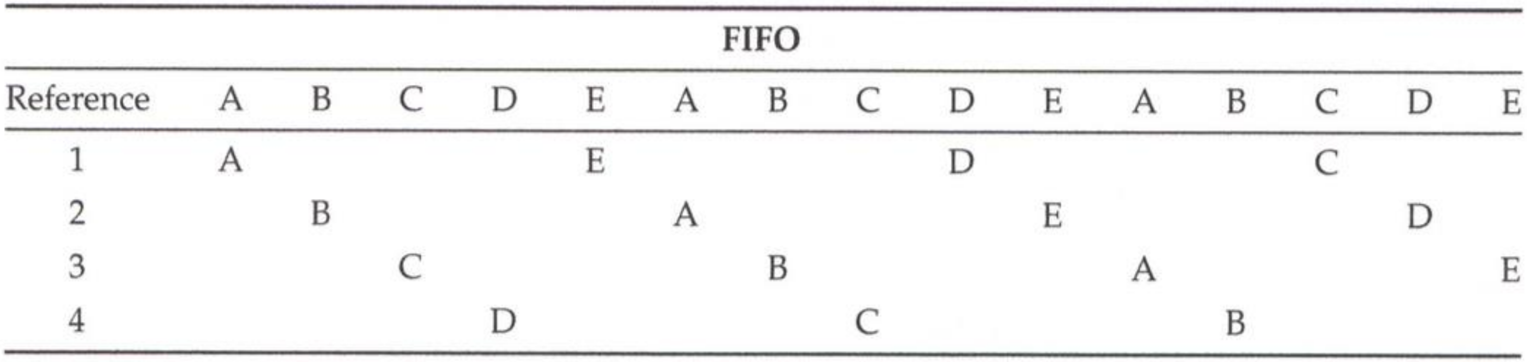
REPLACEMENT ALGORITHMS

Replacement Policies

- On a miss, a new block must be brought in
- This requires evicting a current block residing in the cache
- Optimal Replacement Policy
  - MIN: Replace block used the farthest in the future
    - Requires knowledge of the future
- Practical Replacement policies
  - FIFO: First-in first-out (oldest block replaced)
  - LRU: Least recently used (usually best but hard to implement)
  - Random: Actually performs surprisingly well
- What about Least Frequently Used (LFU?)

Replacement Aglorithms

- FIFO can be pessimal (worst possible) for repeated linear scans that don't fit in the cache
- Consider cache of 4 blocks with a repeated iteration through an array that requires 5 blocks of storage



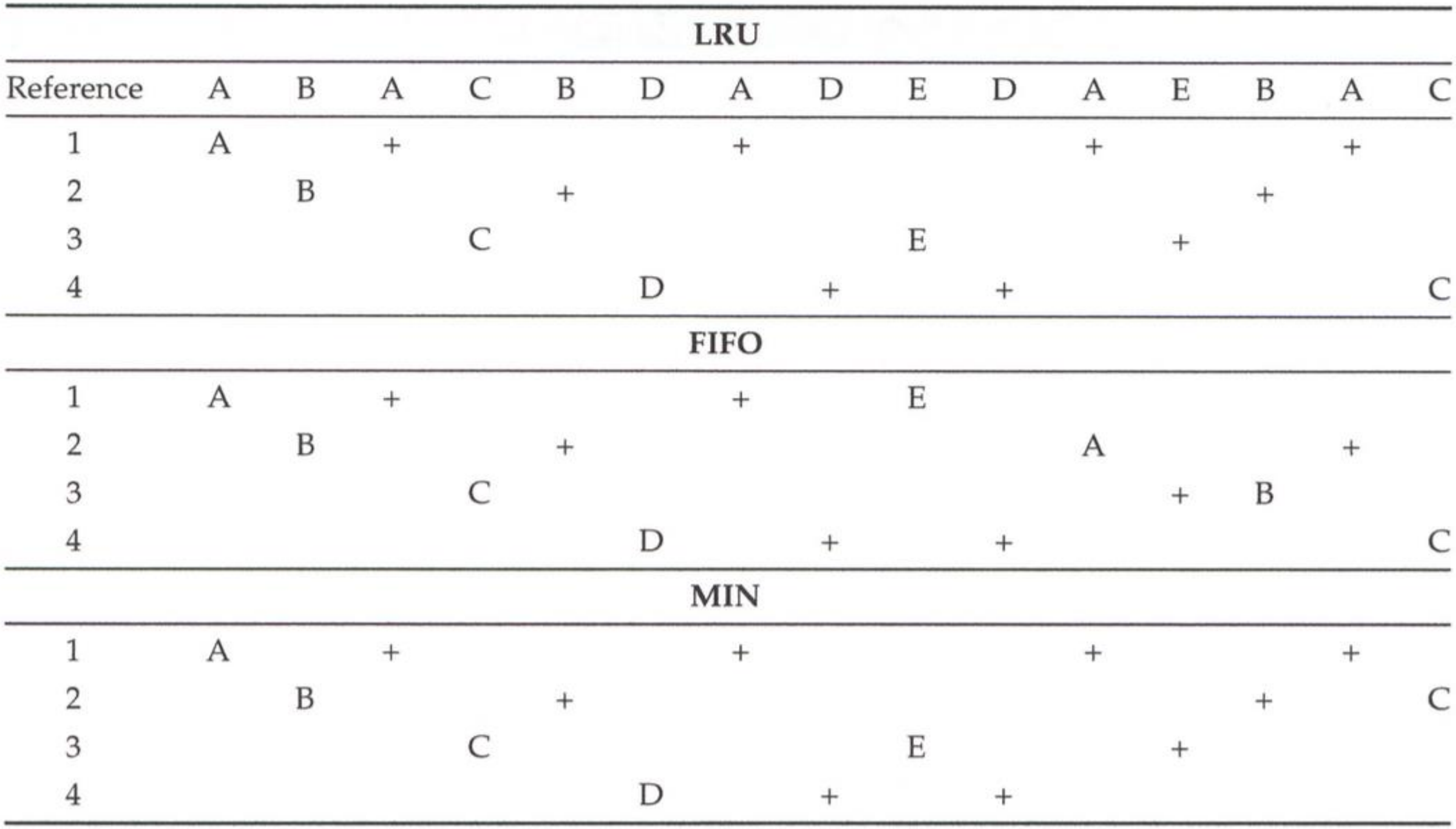
FIFO

Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

OS:PP 2nd Ed. Fig 9.13

Replacement Aglorithms

- Compare the following replacement algorithms for a pattern exhibiting temporal locality

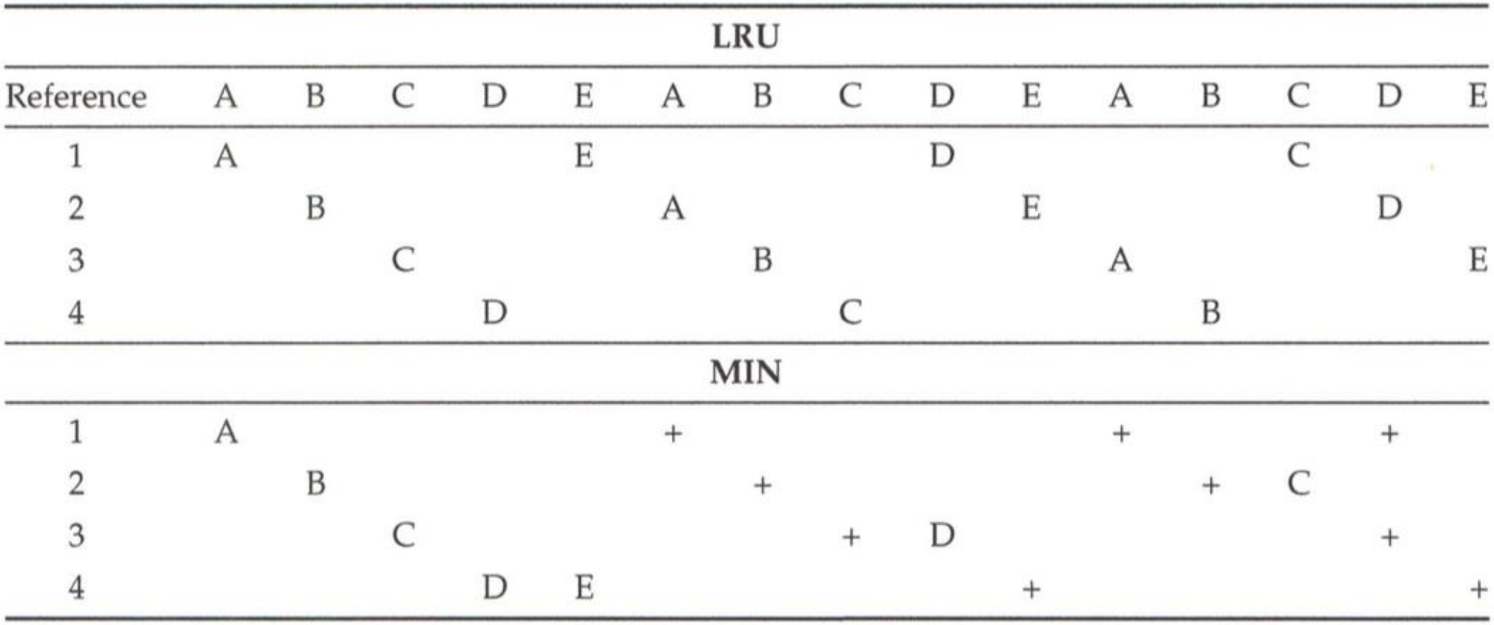


								LRU							
Reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C
								FIFO							
1	A		+				+		E						
2		B			+						A			+	
3				C								+	B		
4						D		+		+					C
								MIN							
1	A		+				+				+			+	
2		B			+								+		C
3				C					E			+			
4						D		+		+					

OS:PP 2nd Ed. Fig 9.14

Replacement Aglorithms

- Compare LRU & MIN following replacement algorithms for a pattern that repeatedly scans through memory



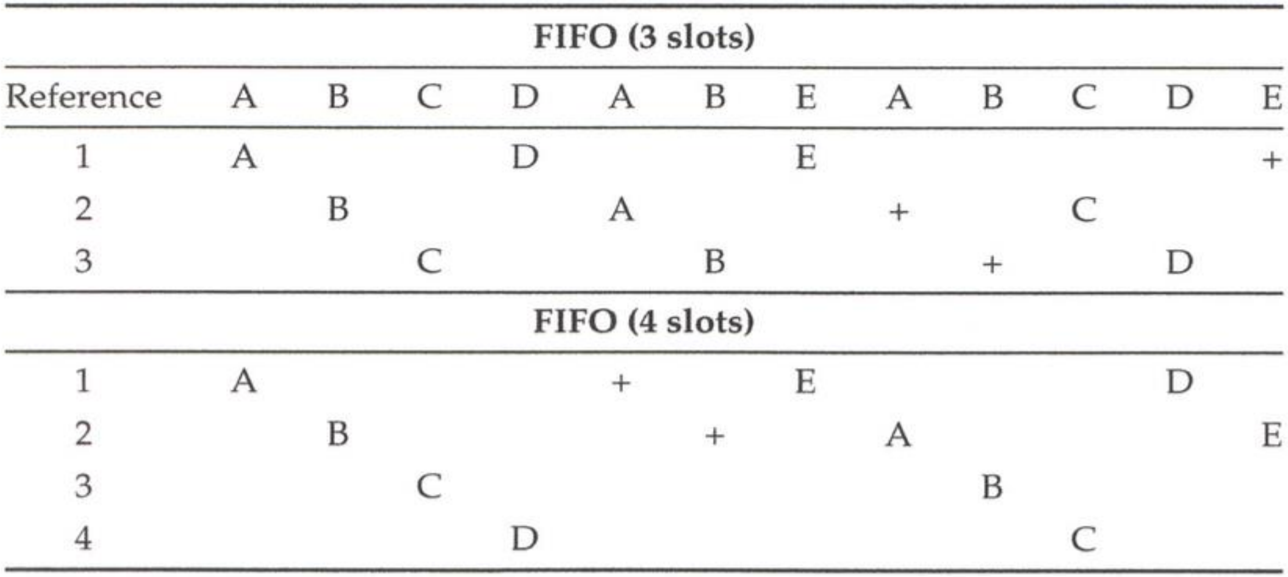
LRU

Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			
MIN															
1	A					+					+			+	
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

OS:PP 2nd Ed. Fig 9.15

Belady's Anomaly

- Adding space to a cache generally helps improve the hit rate
- **BUT NOT ALWAYS!**
- For FIFO, more slots may actually decrease hit rate: **Belady's anomaly**
  - Other algorithms like LRU, MIN, and LFU can be proven to show that adding slots to the cache will ONLY HELP
- Compare the hit rate for FIFO replacement with 3 vs. 4 slots



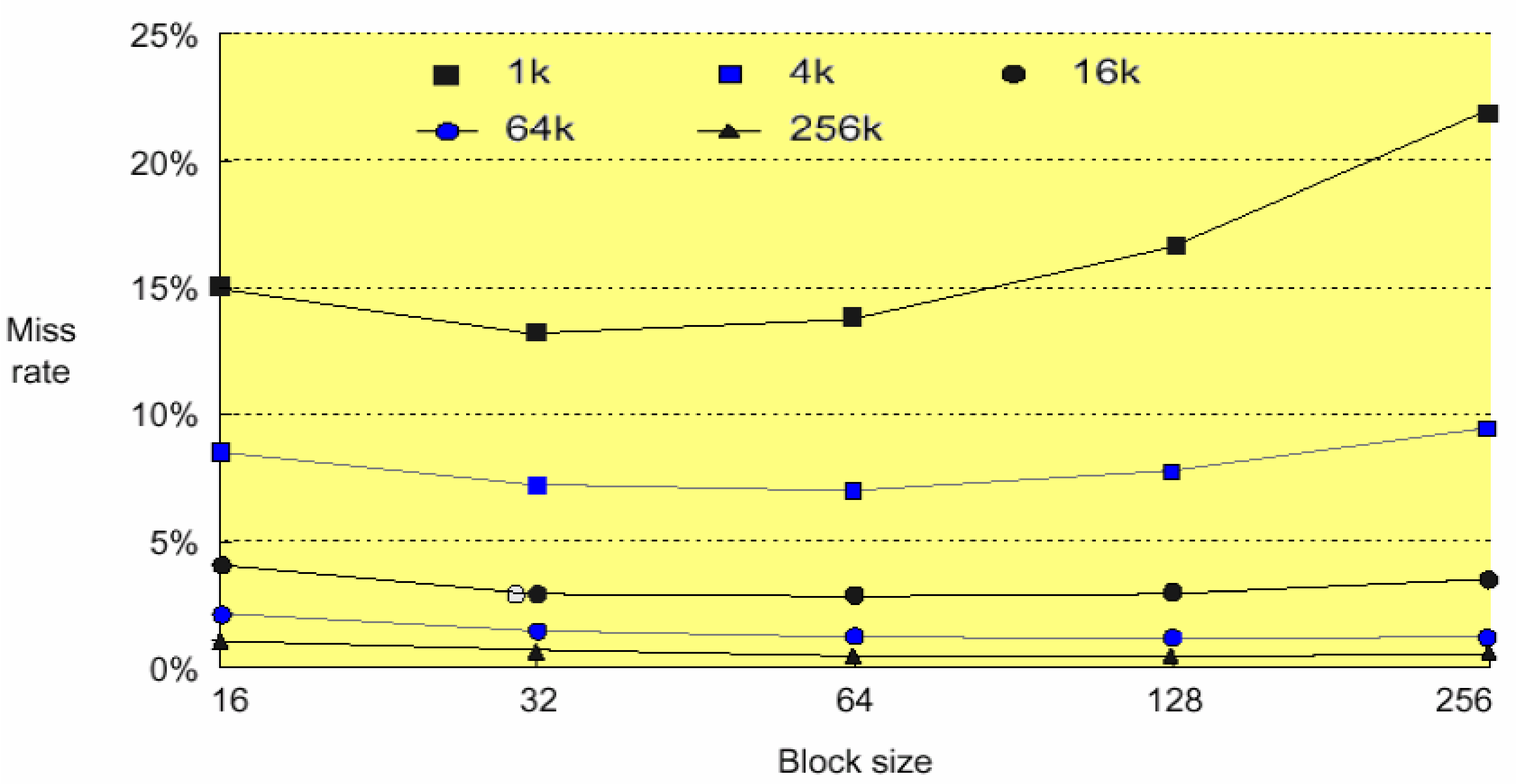
					FIFO (3 slots)							
Reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	
					FIFO (4 slots)							
1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

OS:PP 2nd Ed. Fig 9.15

Miss Rate

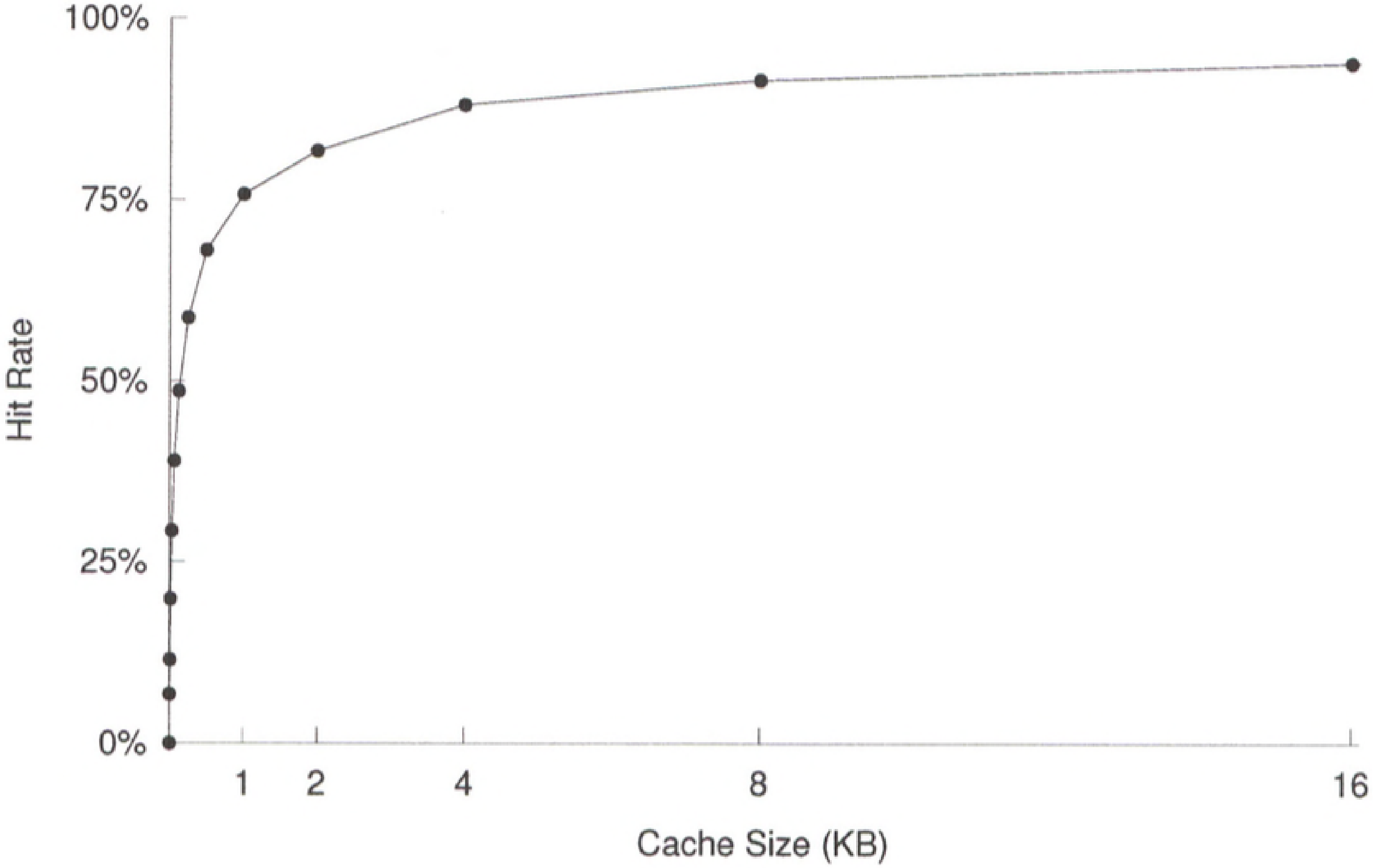
- Reducing Miss Rate means lower T_{AVE}
- To analyze miss rate categorize them based on why they occur
  - Compulsory Misses
    - First access to a block will always result in a miss
  - Capacity Misses
    - Misses because the cache is too small
  - Conflict Misses
    - Misses due to mapping scheme (replacement of direct or set associative)

Miss Rate & Block Size



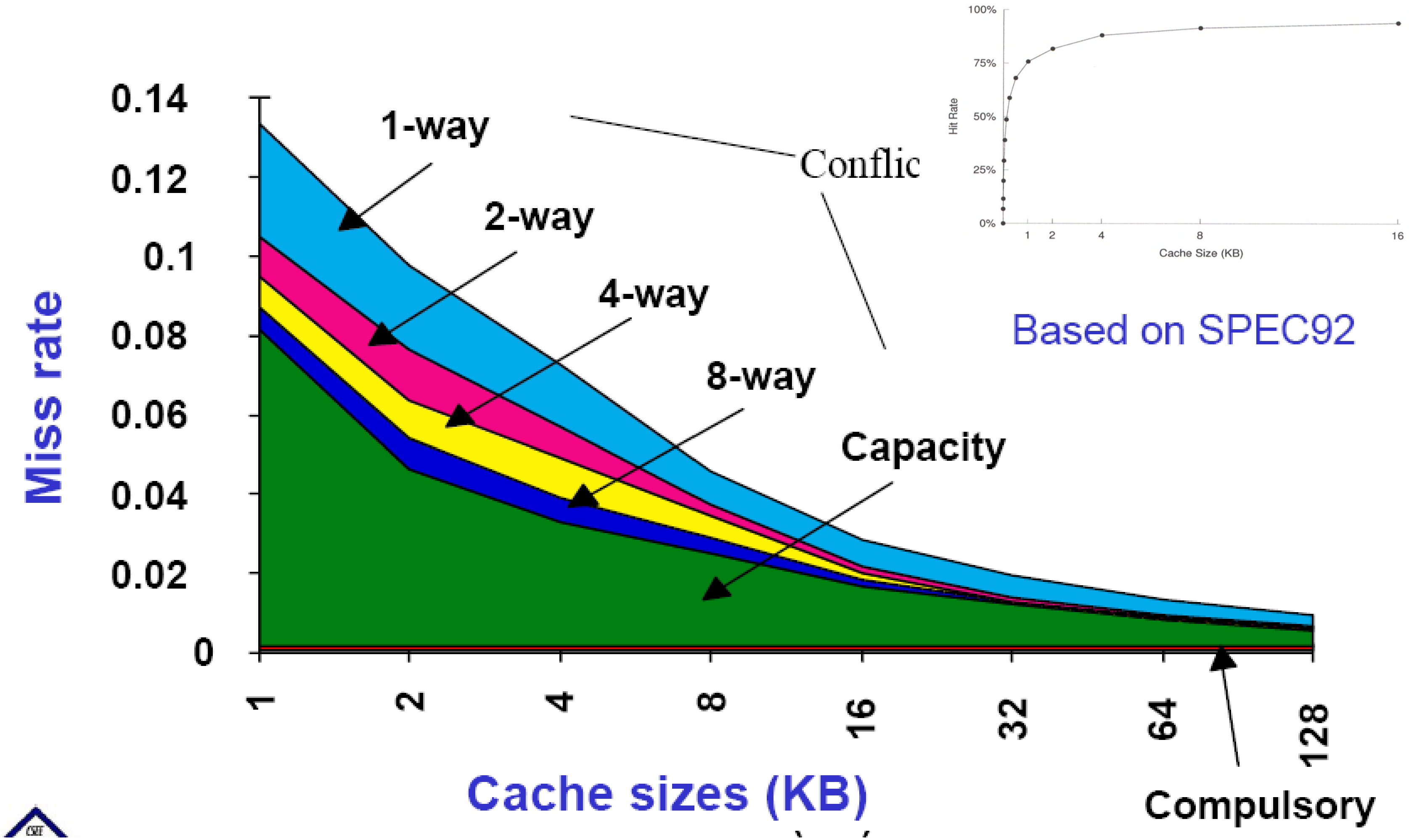
Graph used courtesy “Computer Architecture: AQA, 3rd ed.”, Hennessey and Patterson

Hit/Miss Rate vs. Cache Size



OS:PP 2nd Ed.: Fig. 9.4

Miss Rate & Associativity



Based on SPEC92

Graph used courtesy "Computer Architecture: AQA, 3rd ed.", Hennessey and Patterson

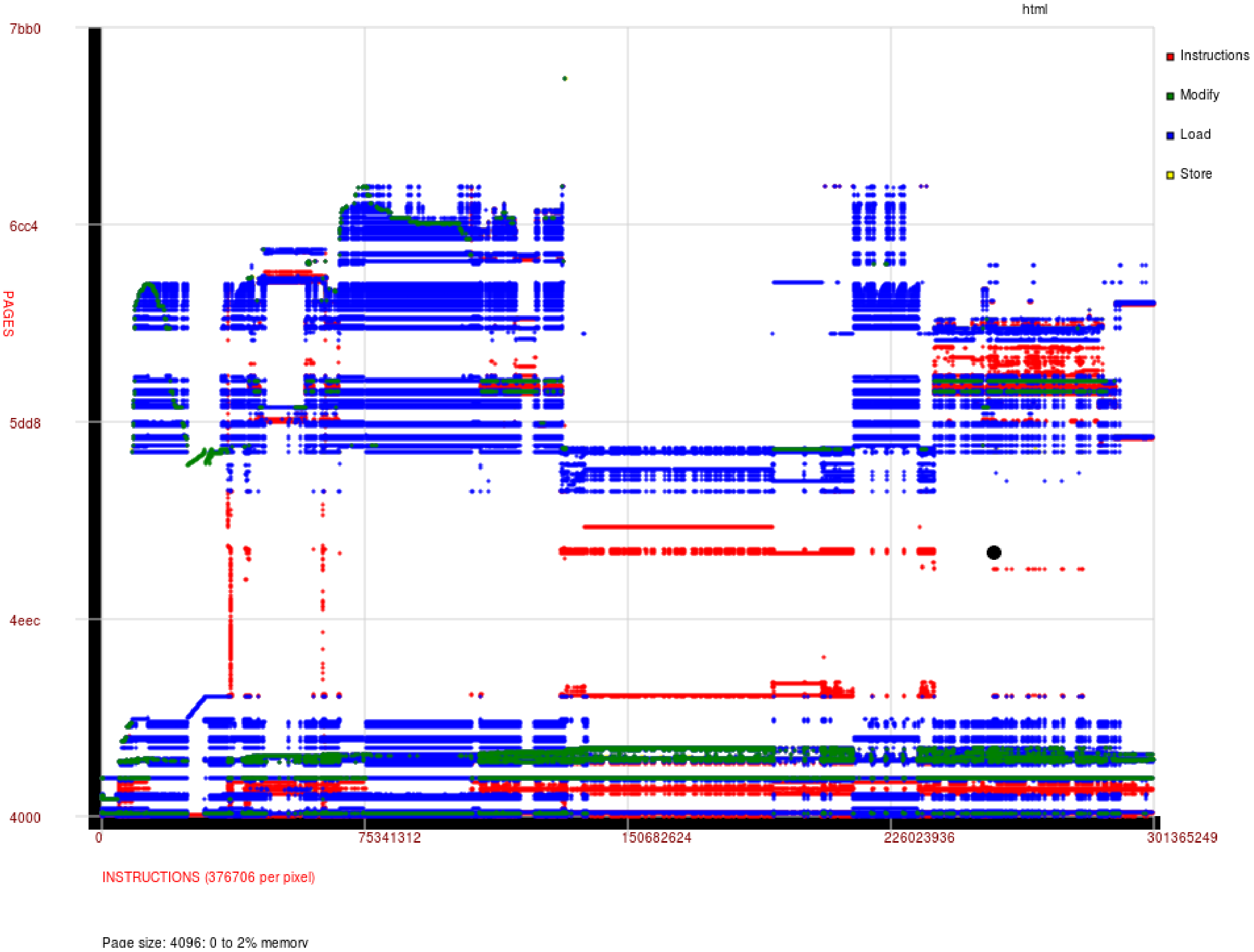
Prefetching

- Hardware Prefetching
  - On miss of block i, fetch block i and i+1
- Software Prefetching
  - Special “Prefetch” Instructions
  - Compiler inserts these instructions to give hints ahead of time as to the upcoming access pattern

CACHE CONSCIOUS PROGRAMMING

Working Sets

- Generally a program works with different sets of data at different times
  - Consider an image processing algorithm akin to JPEG encoding
    - Perform data transformation on image pixels using several weighting tables/arrays
    - Create a table of frequencies
    - Perform compression coding using that table of frequencies
    - Replace pixels with compressed codes
- The data that the program is accessing in a small time window is referred to as its working set
- We want that working set to fit in cache and make as much reuse of that working set as possible while it is in cache
  - Keep weight tables in cache when performing data transformation
  - Keep frequency table in cache when compressing

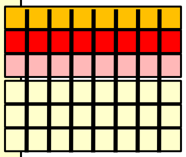


https://cartesianproduct.wordpress.com/tag/working-set/

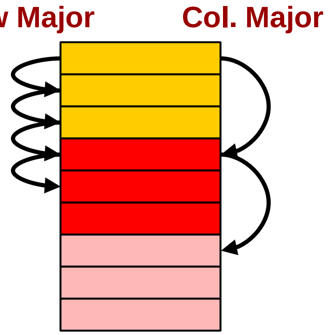
Cache-Conscious Programming

- Order of array indexing
  - Row major vs. column major ordering
- Blocking (keeps working set small)
- Pointer-chasing
  - Linked lists, graphs, tree data structures that use pointers do not exhibit good spatial locality
- General Principles
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)
  - Static structures usually better than dynamic ones

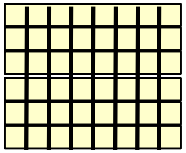
```
for(i=0; i<SIZE; i++) {
  for(j=0; j<SIZE; j++) {
    // Row-major
    A[i][j] = A[i][j]*2;
    // Column-major
    A[j][i] = A[j][i]*2;
} }
```



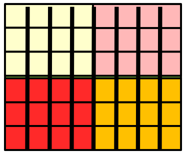
Example of row vs. column major ordering



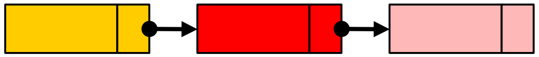
Memory Layout of matrix A



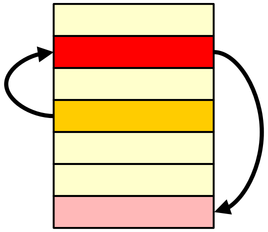
Original Matrix



Blocked Matrix



Linked Lists

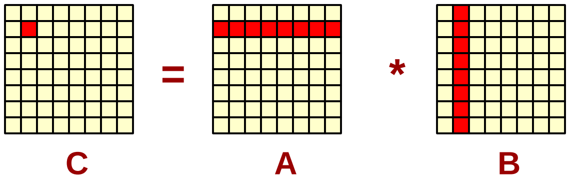


Memory Layout of Linked List

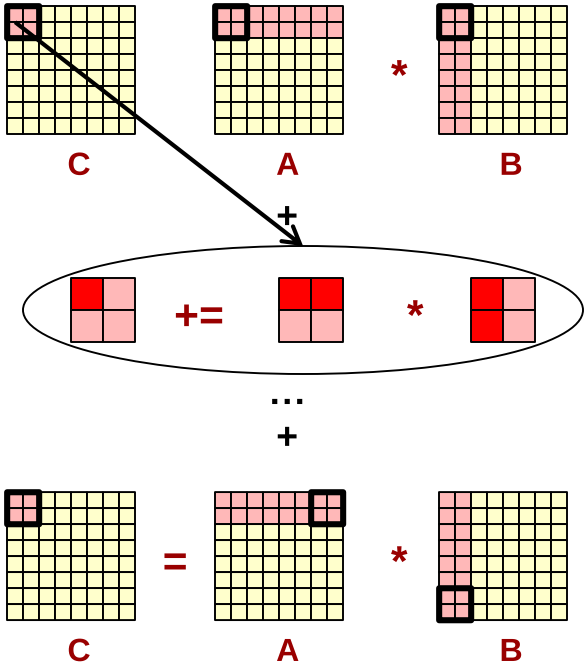
Blocked Matrix Multiply

- Traditional working set
  - 1 row of C, 1 row of A, NxN matrix B
- Break NxN matrix into smaller BxB matrices
  - Perform matrix multiply on blocks
  - Sum results of block multiplies to produce overall multiply result
- Blocked multiply working set
  - Three BxB matrices

```
for(i = 0; i < N; i+=B) {
 for(j = 0; j < N; j+=B) {
  for(k = 0; k < N; k+=B) {
   for(ii = i; ii < i+B; ii++) {
    for(jj = j; jj < j+B; jj++) {
     for(kk = k; kk < k+B; kk++) {
       Cb[ii][jj] += Ab[ii][kk] * Bb[kk][jj];
} } } } } }
```



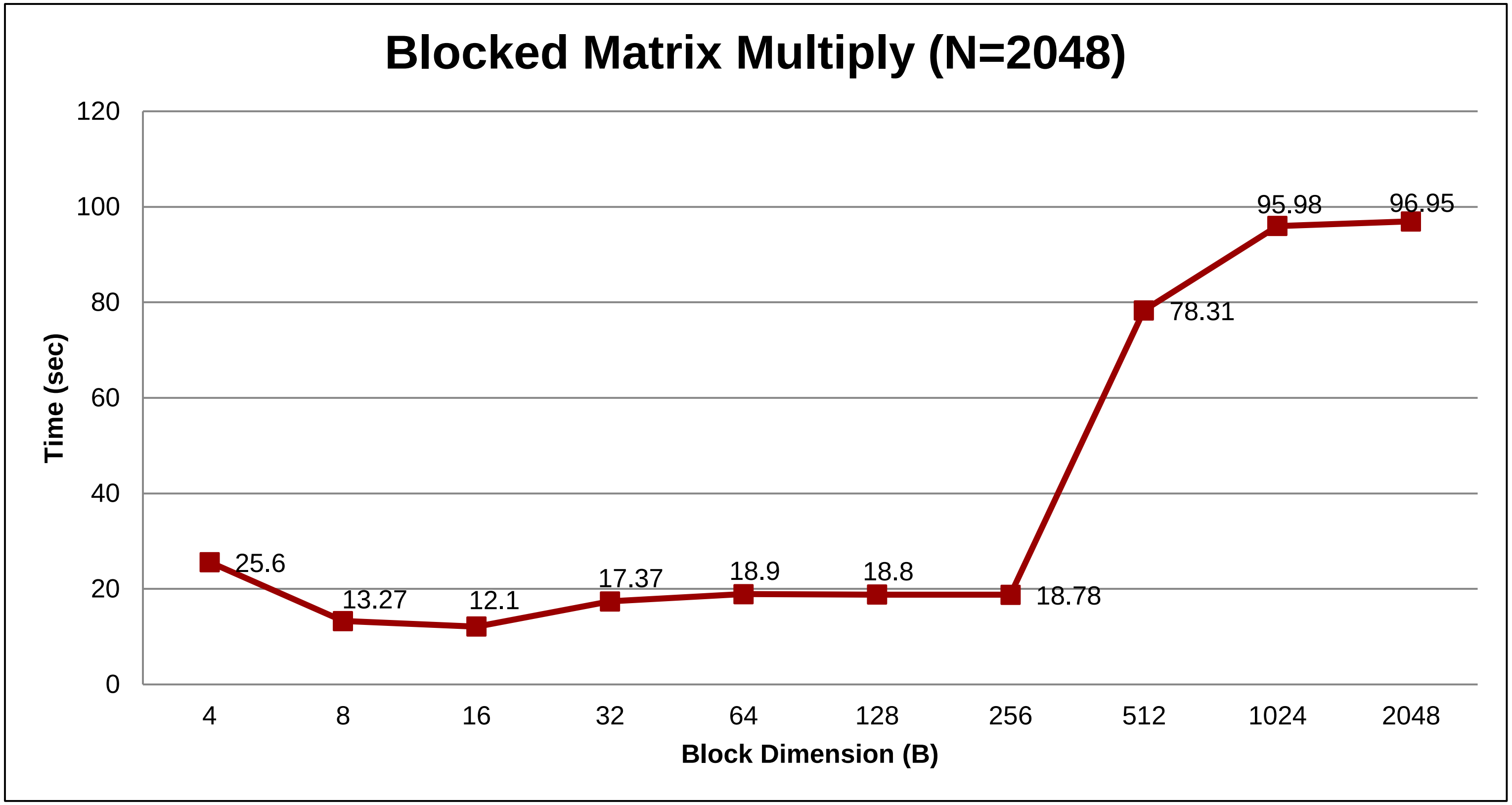
Traditional Multiply



Blocked Multiply

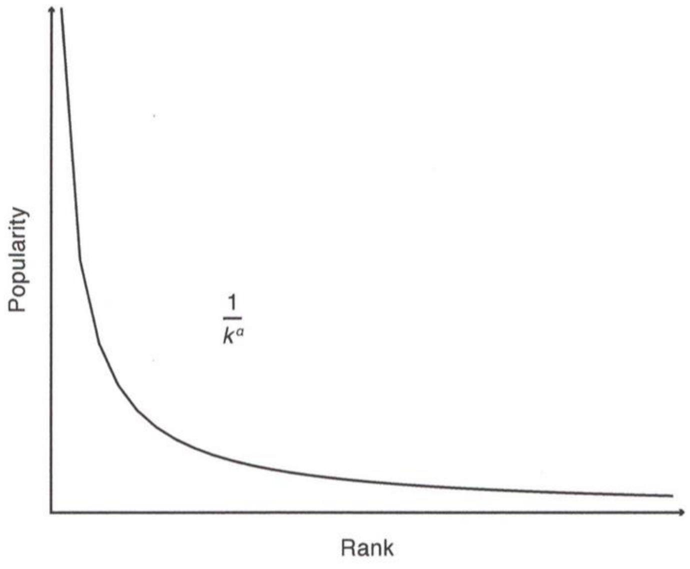
Blocked Multiply Results

- Intel Nehalem processor
  - L1D = 32 KB, L2 = 256KB, L3 = 8 MB



Zipf Distribution

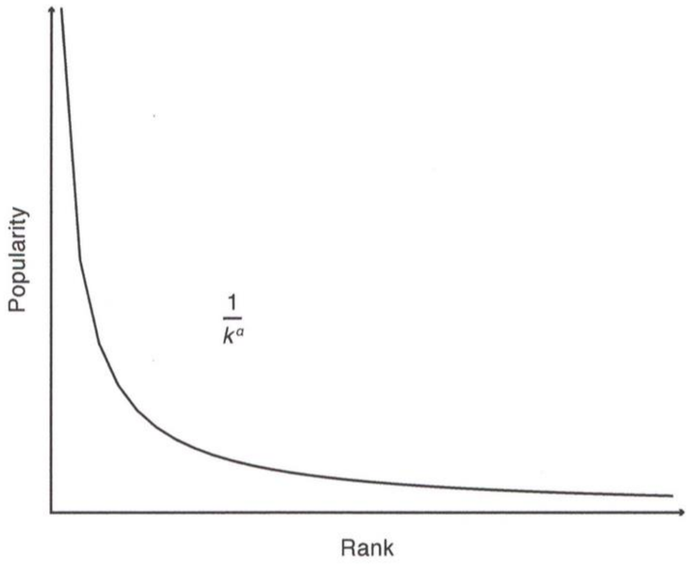
- Zipf modeled the frequency of word usage in larger text bodies
- Zipf model says the frequency of access of the k-th most frequent/popular item from a set is $1/k^{\alpha}$ *where [1 < α < 2]*
- Applies to may other domains
  - Web page access on the Internet
  - Popularity of cities, books, etc.
  - Size of friend lists in social networks



OS:PP 2nd Ed.: Fig. 9.7

Cache Implications

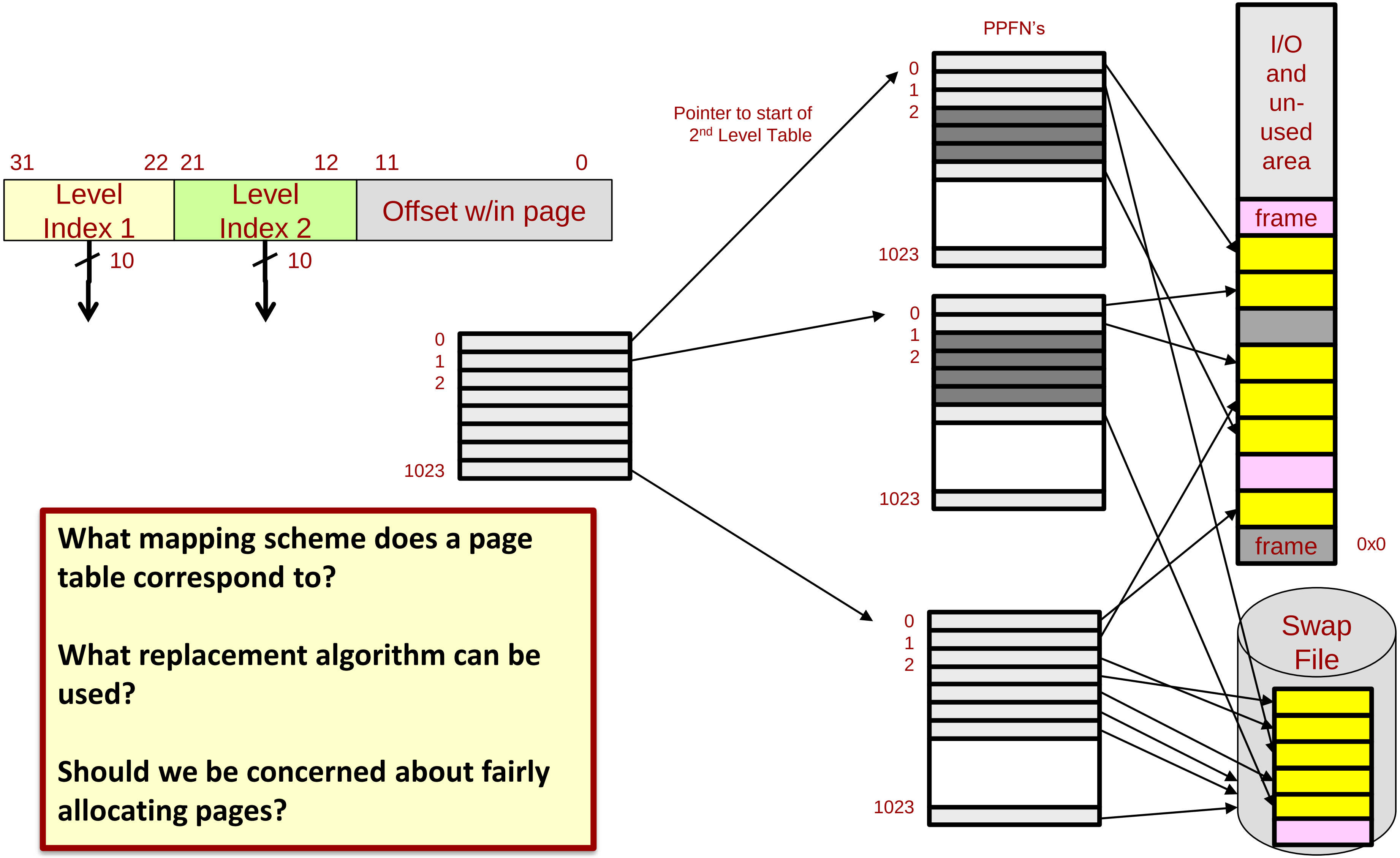
- Zipf-ian distributions may not perform well even on large caches due to the heavy-tail
- Web-page cache
  - New data: New pages are being added all the time
  - No working set: While there are some popular webpages, no small subset will cover the bulk of the accesses
- Diminishing returns as the cache size is increased



OS:PP 2nd Ed.: Fig. 9.7

SWAPPING

Recall: VM Swap = Caching



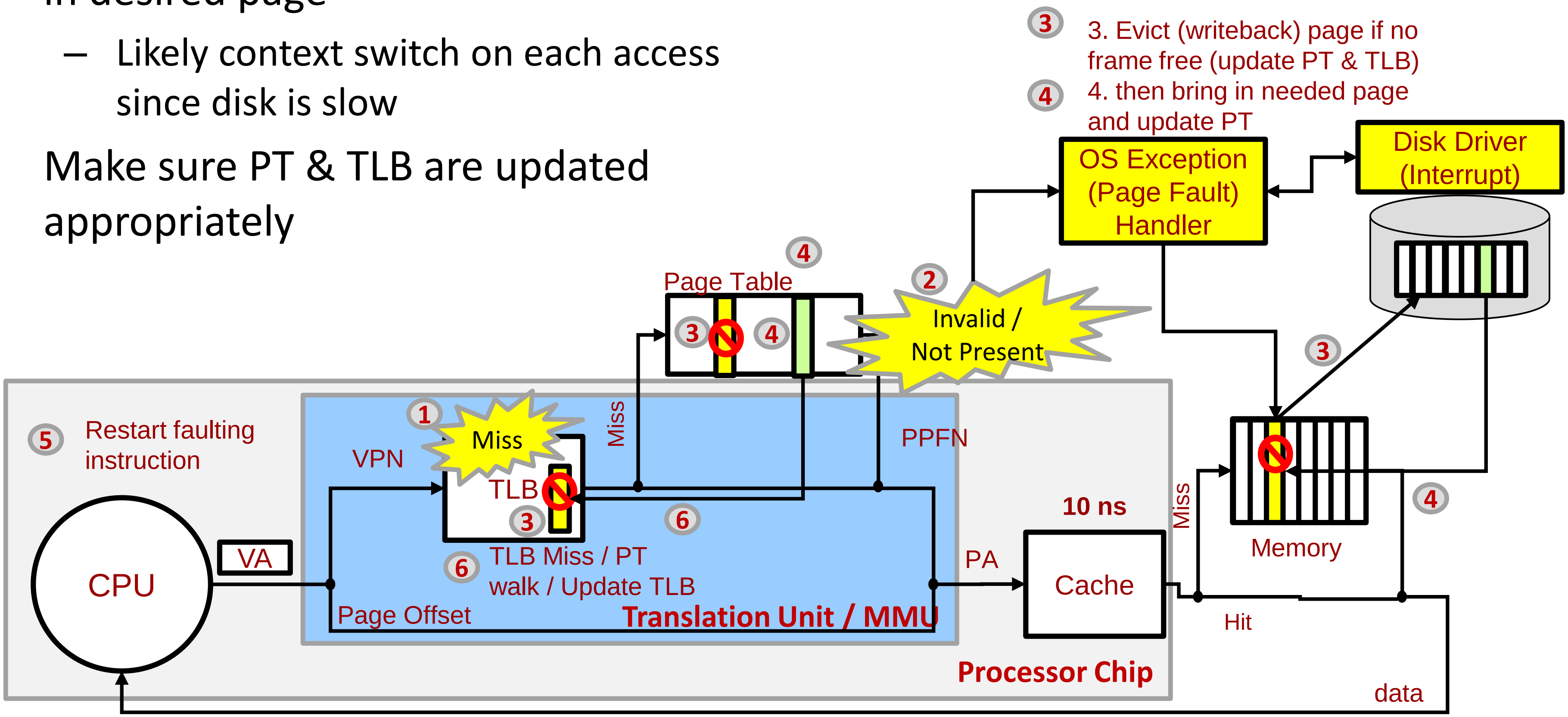
What mapping scheme does a page table correspond to?

What replacement algorithm can be used?

Should we be concerned about fairly allocating pages?

Page Fault Steps

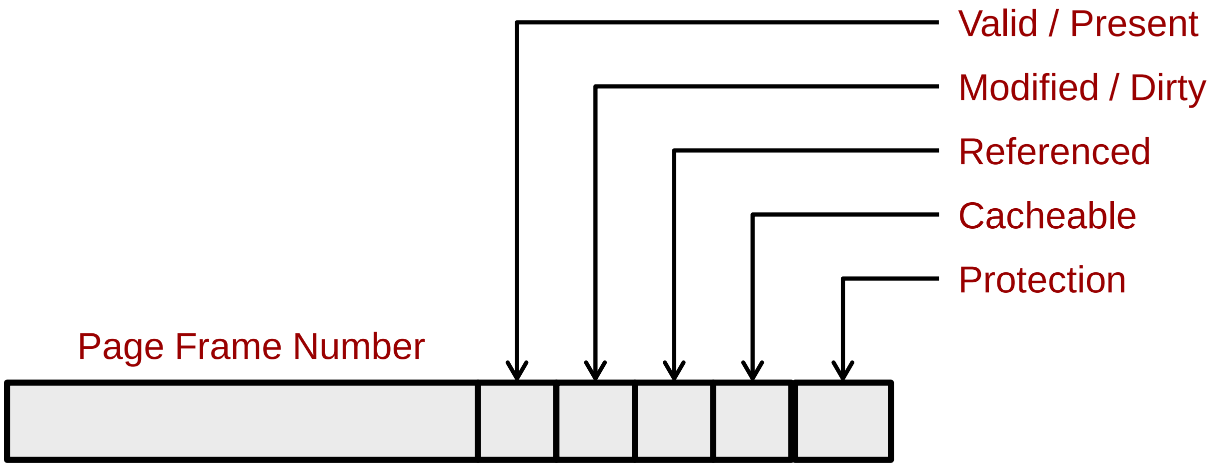
- On page fault, handler will access disk possibly on eviction and to bring in desired page
  - Likely context switch on each access since disk is slow
- Make sure PT & TLB are updated appropriately

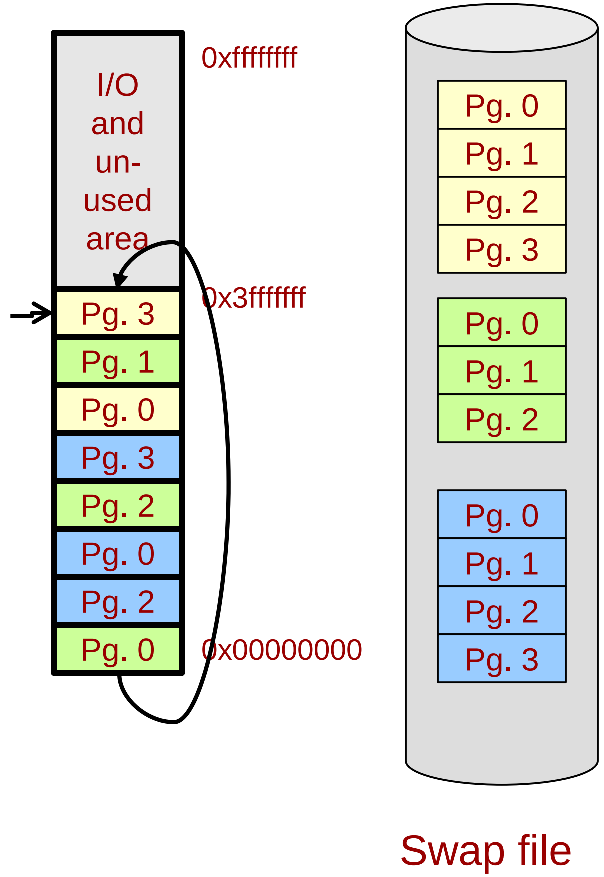


3
3. Evict (writeback) page if no frame free (update PT & TLB)
4
4. then bring in needed page and update PT
OS Exception (Page Fault) Handler
Disk Driver (Interrupt)
4
Page Table
3
4
2
Invalid / Not Present
3
5
Restart faulting instruction
1
Miss
VPN
Miss
PPFN
TLB
3
6
10 ns
Miss
Memory
4
CPU
VA
6
TLB Miss / PT walk / Update TLB
PA
Cache
Page Offset
Translation Unit / MMU
Hit
Processor Chip
data

VM Eviction Algorithms

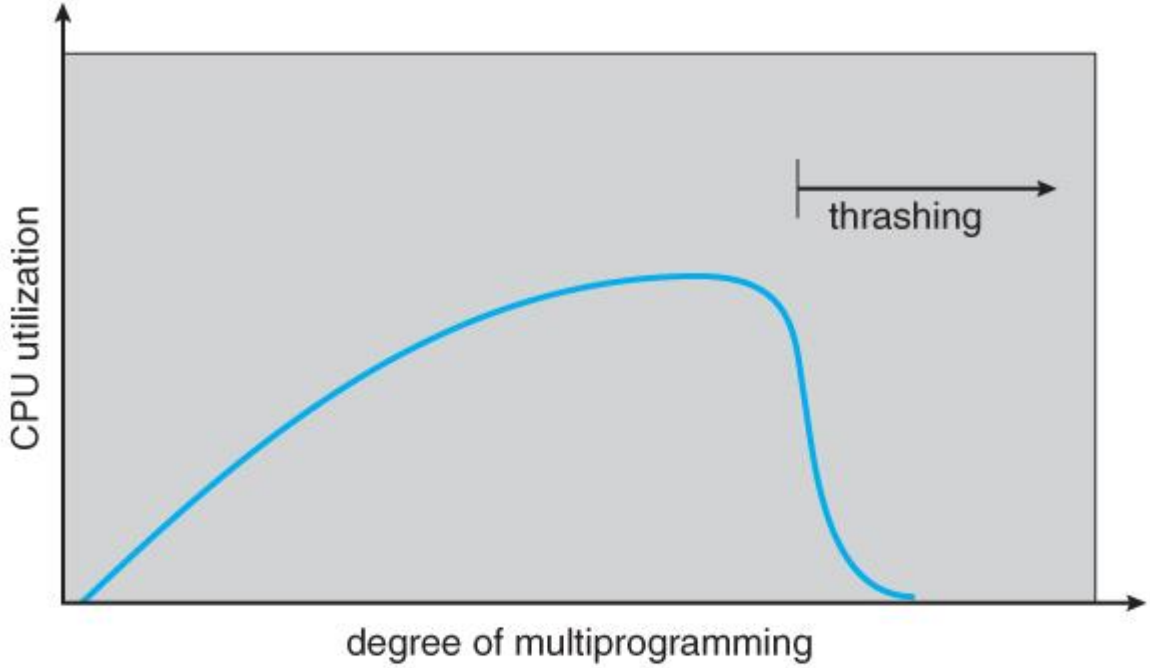
- Clock algorithm
  - Cycle through frames (circular queue)
- Second-chance Algorithm
  - Clock algorithm but pages w/ referenced bit set get a 2nd chance (wait until next cycle) to be evicted)
  - May give preference to dirty pages
- Pseudo-LRU
  - Use HW reference bits + OS-managed reference counts to perform some form of pseudo-LRU





Thrashing and Sharing

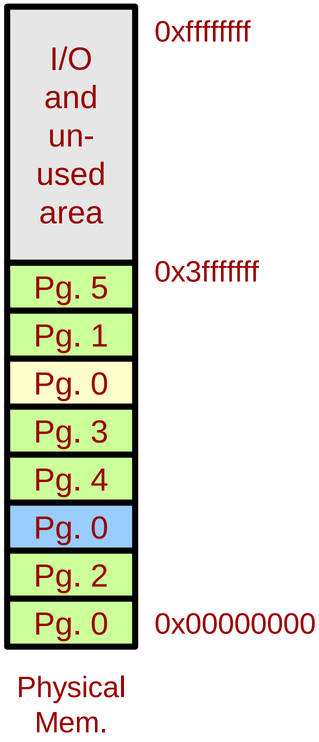
- When too many processes are sharing cache or main memory paging, thrashing may occur
- **Thrashing**: **Working set** cannot fit in memory causing constant, evictions and re-fetching of needed data
  - CPU is underutilized b/c it is constantly waiting on the memory system



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/9_VirtualMemory.html

Page Allocation Fairness

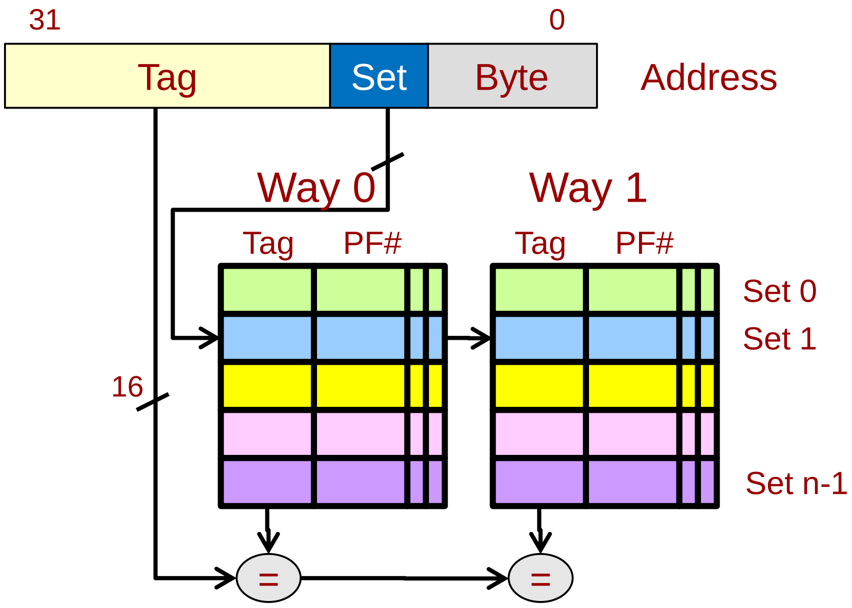
- Want to prevent a few processes from hogging all the physical resources (or possibly all the swap space)
- Max-min fairness for how many pages allocated to process
  - Maximize responsiveness to the minimum request and then redistribute remainder to other processes
- Example:
  - Solaris (Unix) has a background thread that can utilize some percentage of the CPU's time looking for pages to evict
  - Can enforce limits on how many frames a process is occupying

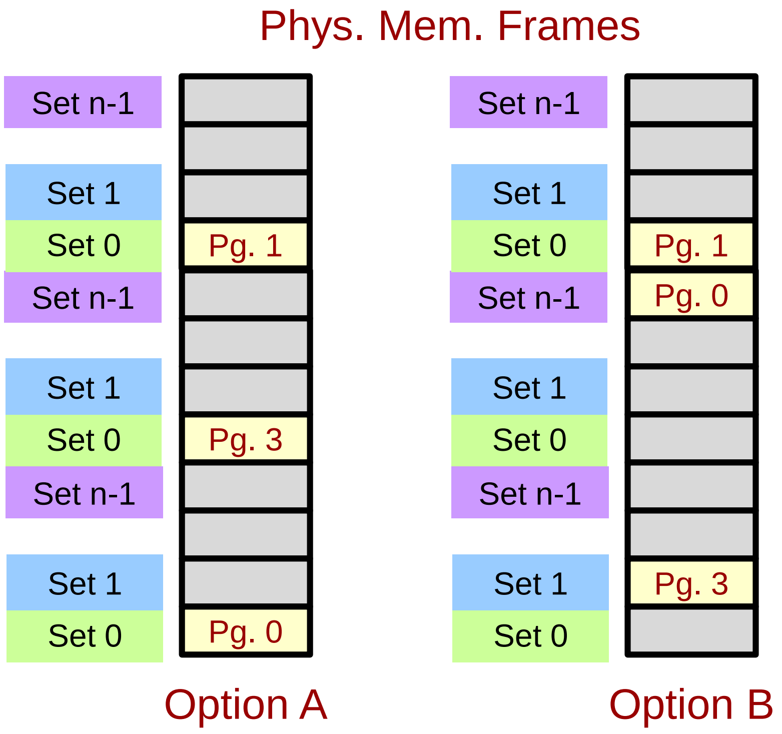


https://docs.oracle.com/cd/E23823_01/html/817-0404/chapter2-10.html

Page Coloring

- We would not want to allocate pages to a process that all map (hash) to the same cache
  - If so, then when that process runs it would be having to walk the page table much too often
- The OS can keep track of the sets that pages allocated to a given process hash to and then allocate a page that hash to a different set (color) on the next request


31
0
Tag
Set
Byte
Address
Way 0
Way 1
Tag
PF#
Tag
PF#
Set 0
Set 1
Set n-1
16
=
=


Phys. Mem. Frames
Set n-1
Set 1
Set 0
Set n-1
Set 1
Set 0
Set n-1
Set 1
Set 0
Pg. 1
Pg. 3
Pg. 0
Option A
Set n-1
Set 1
Set 0
Set n-1
Set 1
Set 0
Set n-1
Set 1
Set 0
Pg. 1
Pg. 0
Pg. 3
Option B