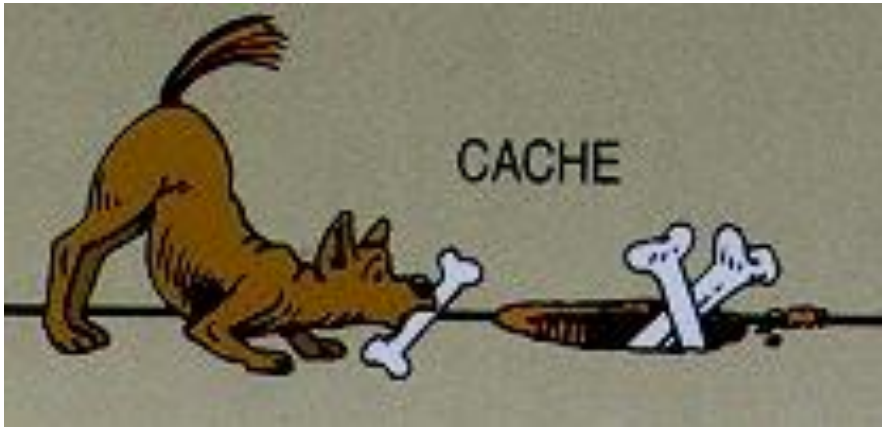
ENCE360 Operating Systems

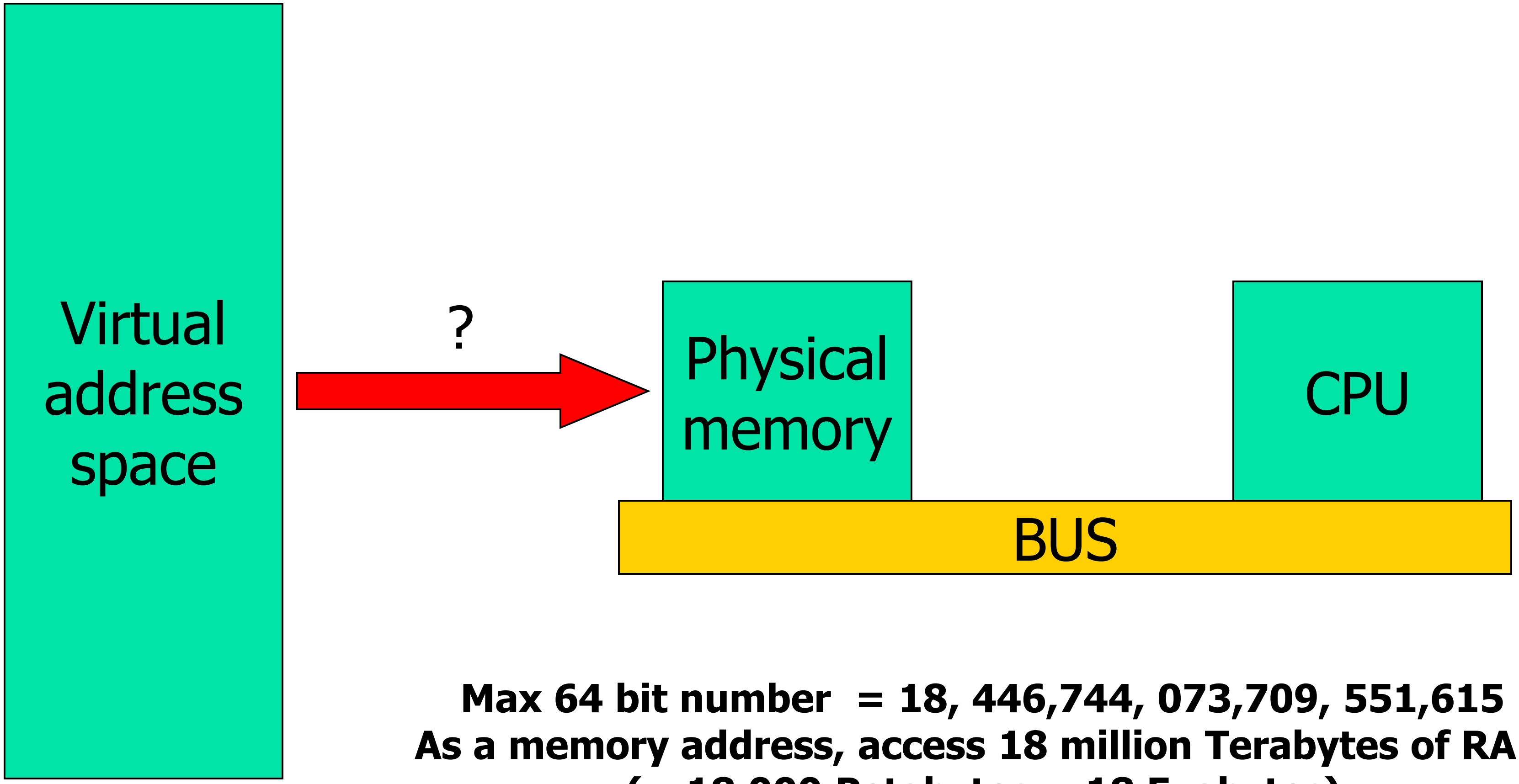


Memory Management

Caching for Faster Memory

MOS Ch 3

Virtual memory



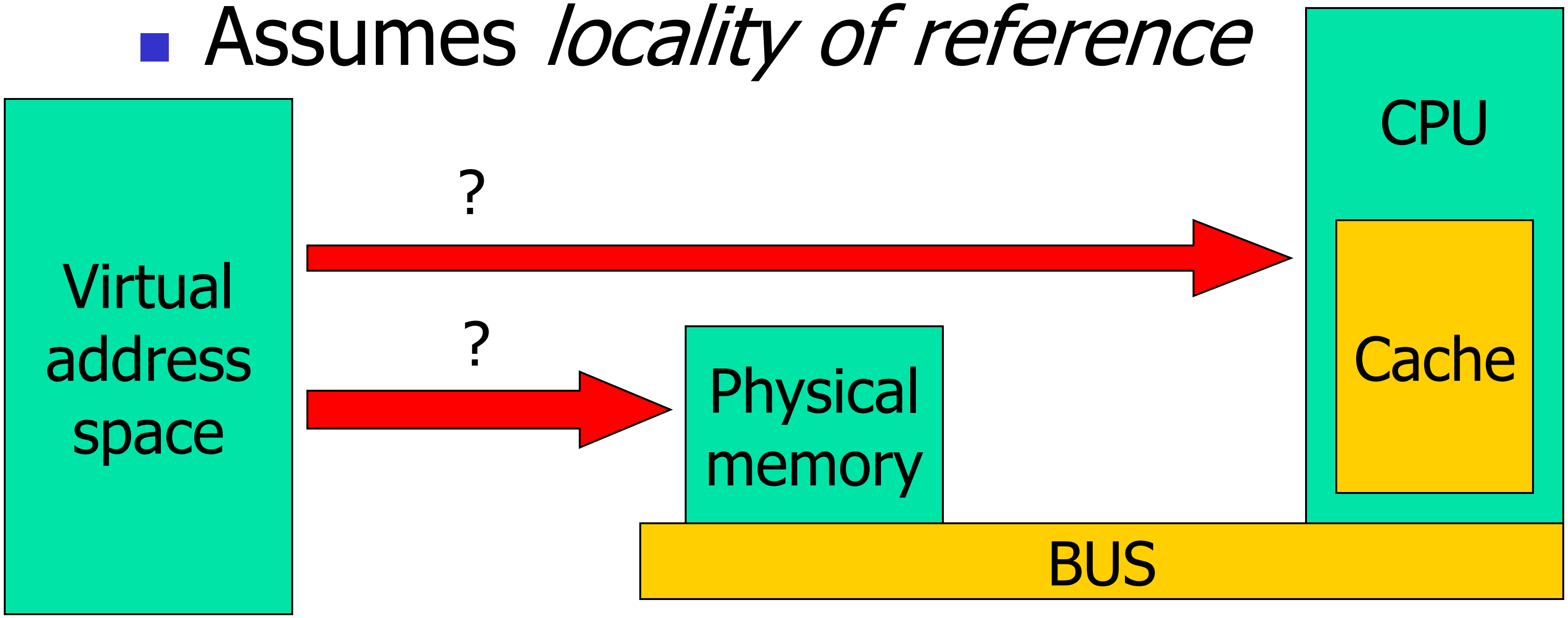
Max 64 bit number = 18, 446,744, 073,709, 551,615
As a memory address, access 18 million Terabytes of RAM
(= 18,000 Petabytes = 18 Exabytes)

Virtual Memory

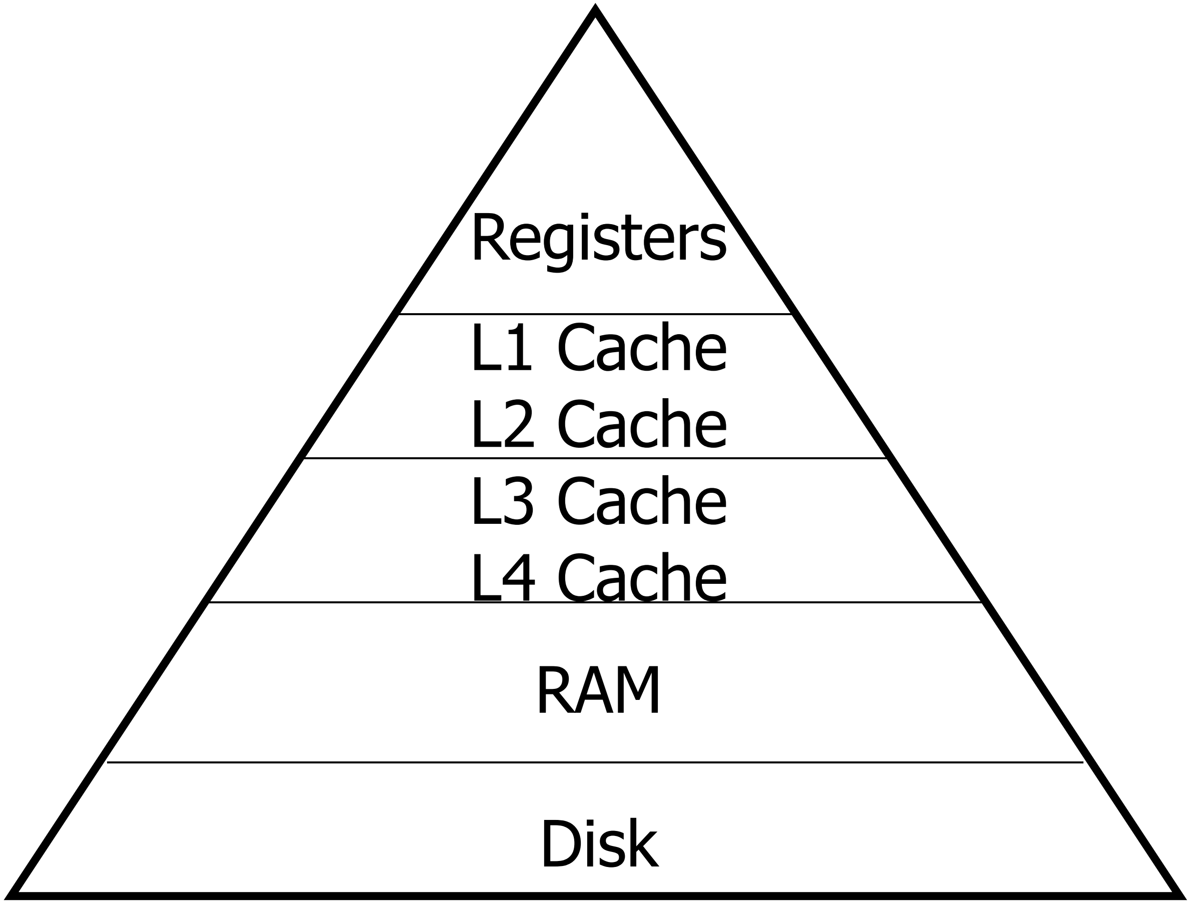
- Programmer access *virtual* addresses
- Hardware translates into *physical* addresses
- *Bus* retrieves the data from physical memory
- *Bus transaction* moves 1 or more bytes of data in from memory

Caching

- Memory *much* slower than processor
- Put faster “cache” close to CPU
- Assumes *locality of reference*



Memory hierarchy



Registers

L1 Cache

L2 Cache

L3 Cache

L4 Cache

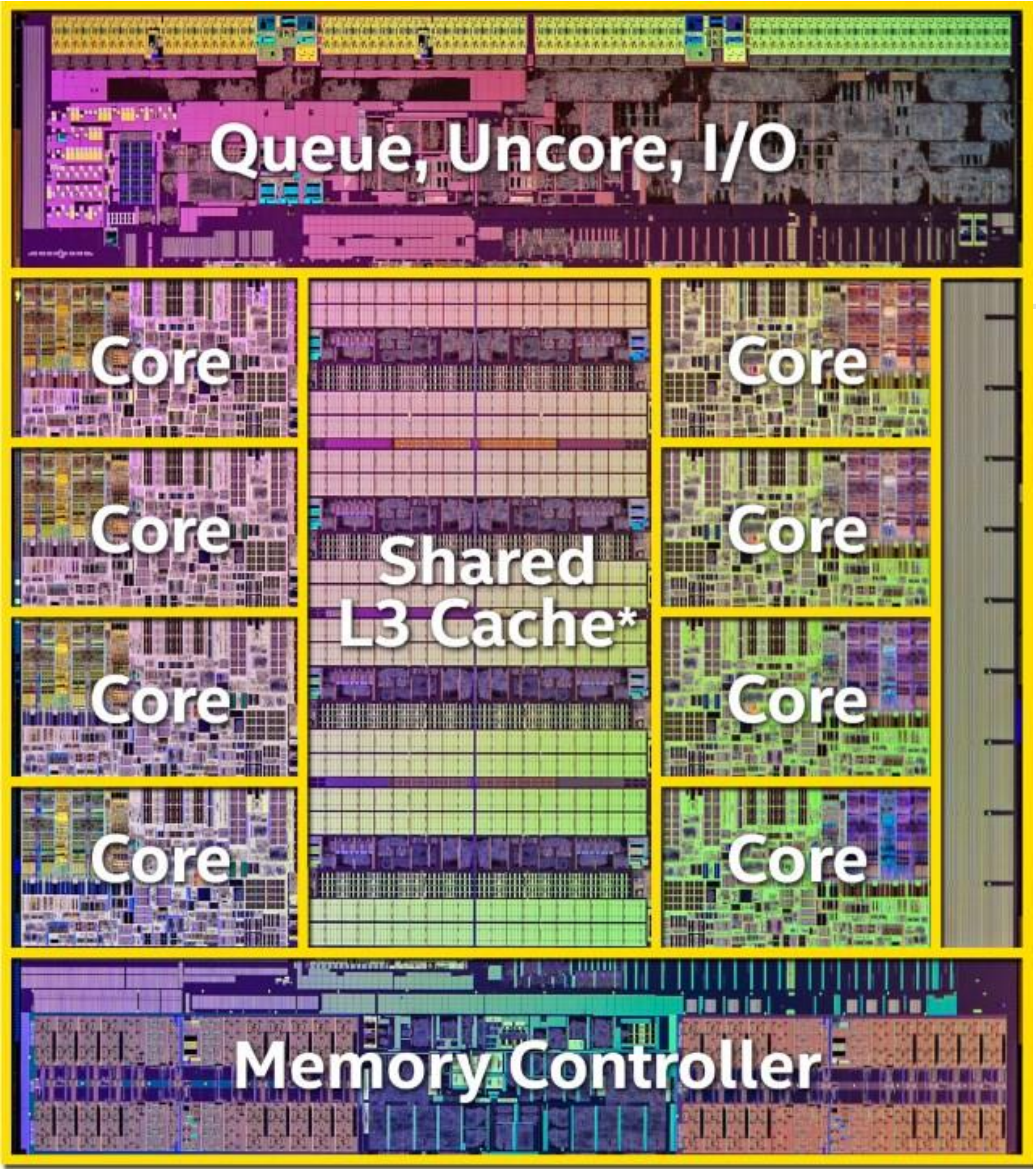
RAM

Disk

Supporting multiple cores

- past: at its simplest level, an L3 cache was just a larger, slower version of the L2 cache. Back when most chips were single-core processors, this was generally true - first L3 caches were actually built on the motherboard itself, connected to the CPU via the backside bus...but now:
- L1 cache per core (e.g. L1 32KB, L2 256KB per core, L3 8MB shared)
- 1999 L2 cache moved on-die (per core, but can be per 2 cores)
- 2008 L3 cache shared between all the cores (8MB L3/4 cores)
- 2013 L4 cache exists off-die, but on-package (uncommon)

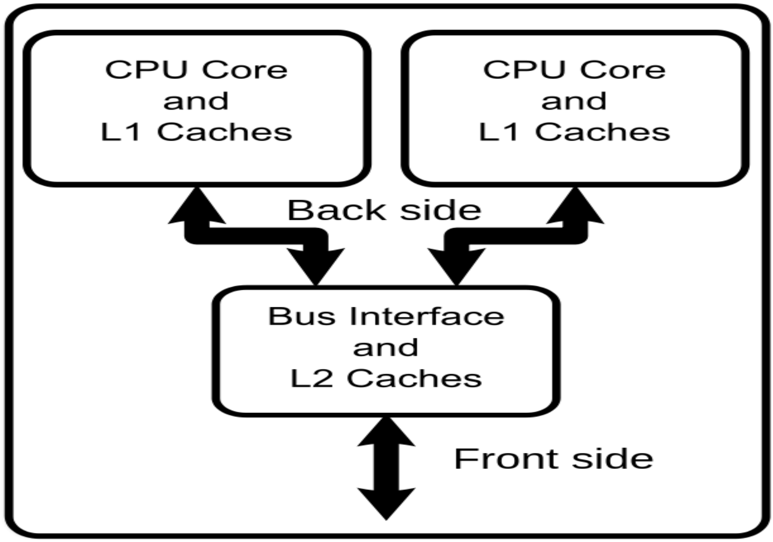
  L4 acts as a victim cache to the L3 cache, meaning anything evicted from L3 cache immediately goes into the L4 cache

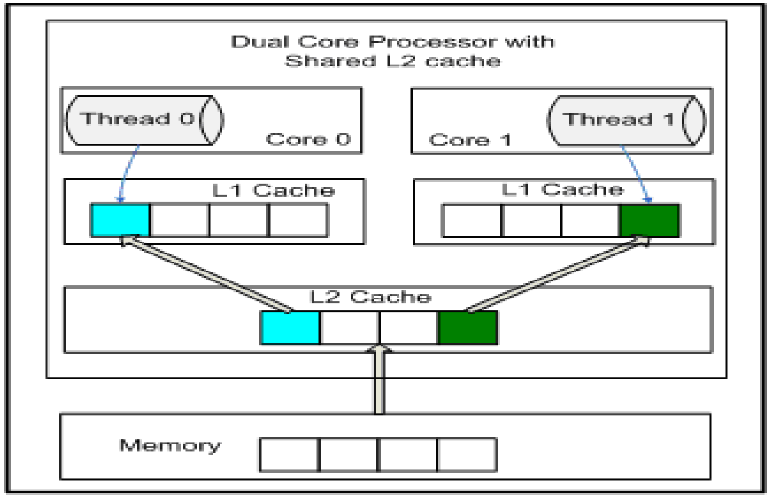


Intel
Haslwell

Supporting multiple cores

- Dual-core processor with CPU-local level-1 caches and a shared, on-die level-2 cache
- The benefits of such a shared cache system include:
  - Flexibility for programmers
  - Reduce cache-coherency complexity
  - Reduce data-storage redundancy
  - Reduce front-side bus traffic





Cache entries

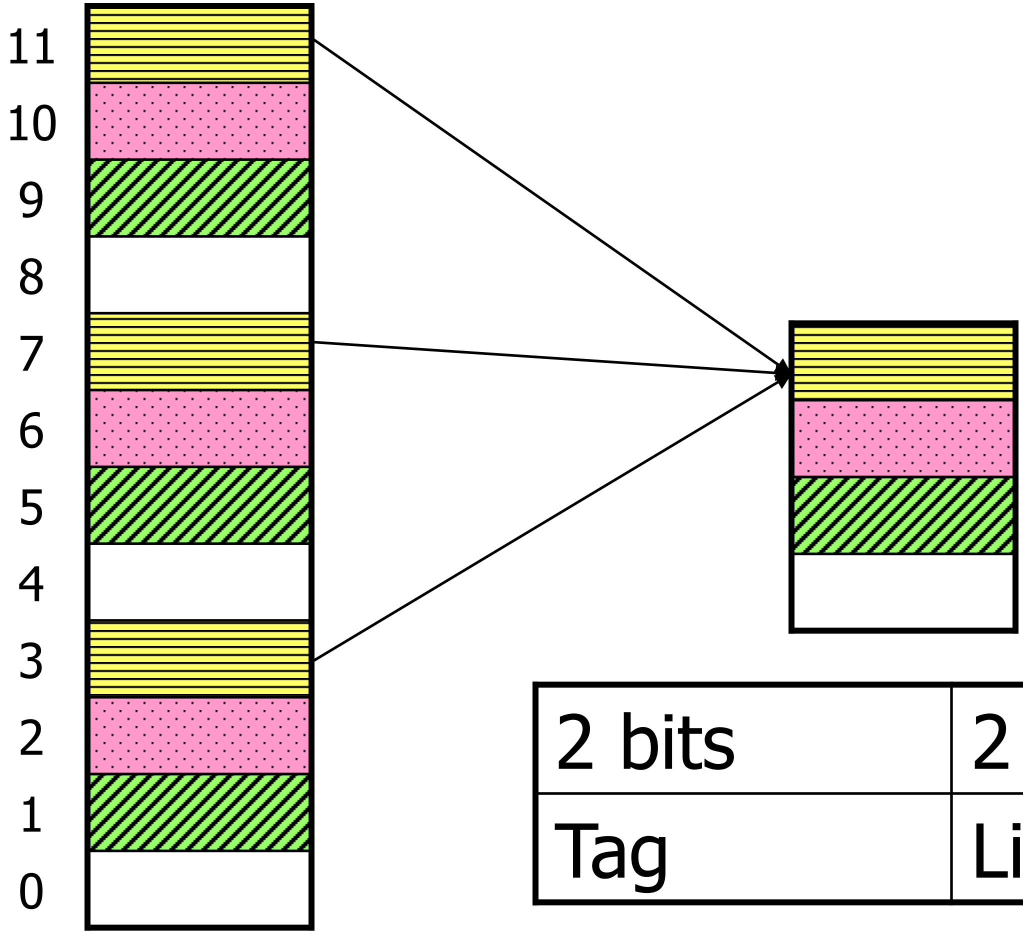
- Cache *line*: fixed length (4 to 64 bytes)
- Valid bit (covered later)
- Cache tag: identifies address range

Valid	Tag	Data 1	Data 2	Data 3	Data 4

Cache flavours

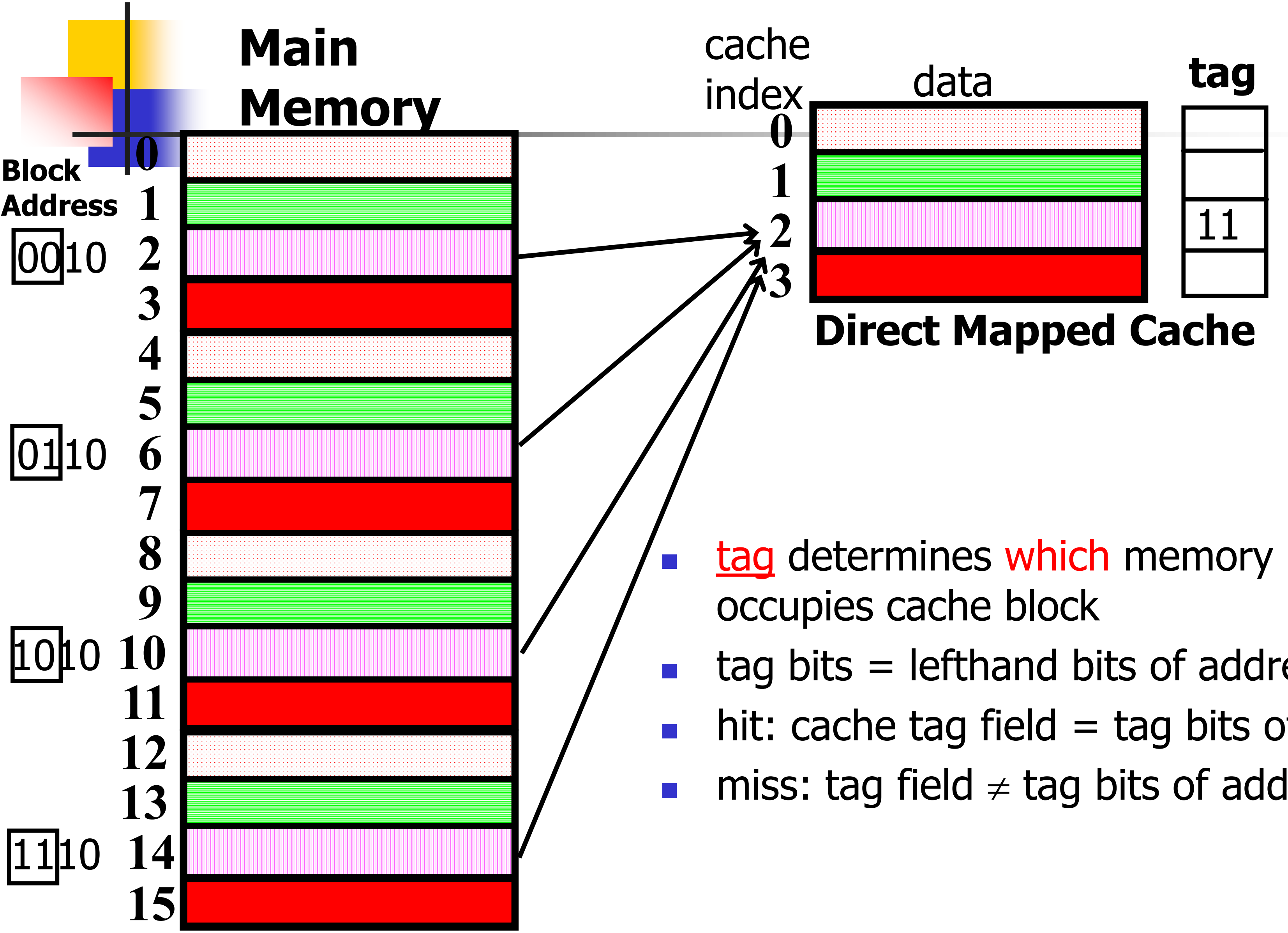
- Direct mapping
  - Address indicates where it is (fast!)
- Associative
  - Any cache line for any address
  - Address identified by tag
- Set associative
  - Combination of the two

Direct mapping



2 bits	2 bits
Tag	Line

Simplest Cache: Direct Mapped w/Tag



- tag determines which memory block occupies cache block
- tag bits = lefthand bits of address
- hit: cache tag field = tag bits of address
- miss: tag field ≠ tag bits of addr.

Direct mapping pros and cons

- Pros:
  - FAST! Direct access to the cache
  - Simple
- Cons:
  - thrashing

Associative caches

- Cache entry is “associated” with address by the tag
  - Search all entries, OR
  - Comparator circuit for each cache line

Tag	Byte
001101	01

Tag	D0	D1	D2	D3
000010				
101001				
001101	🟥			
001110				

Associative cache

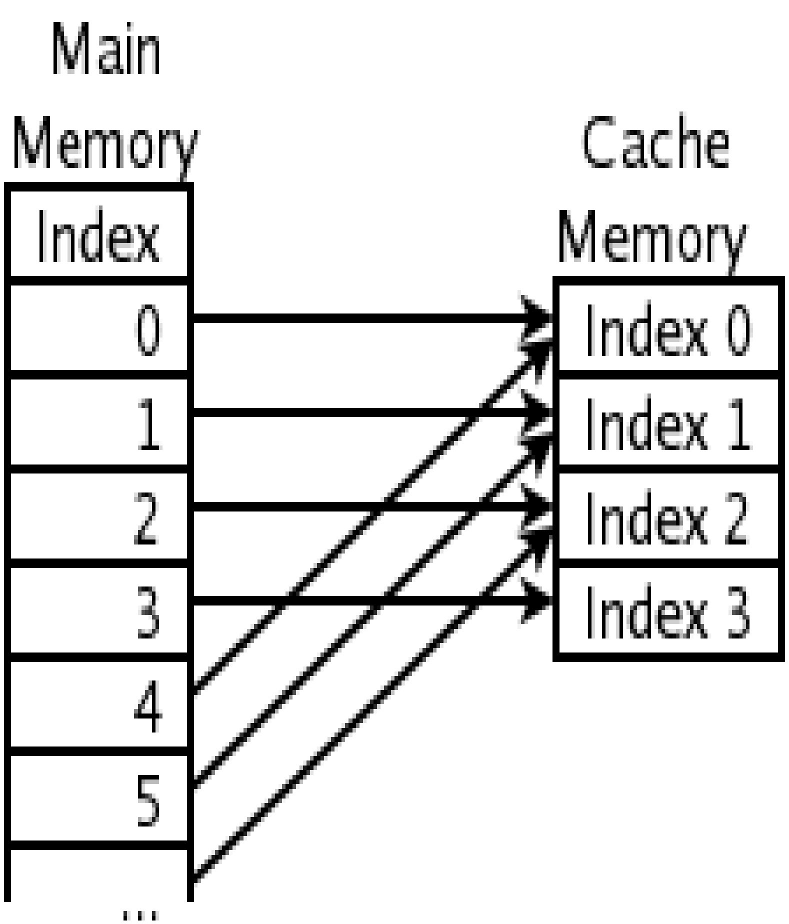
- Pros:
  - Maximises cache use
  - Minimises collisions

- Cons
  - Slow, OR
  - Complex/expensive

Set-associative

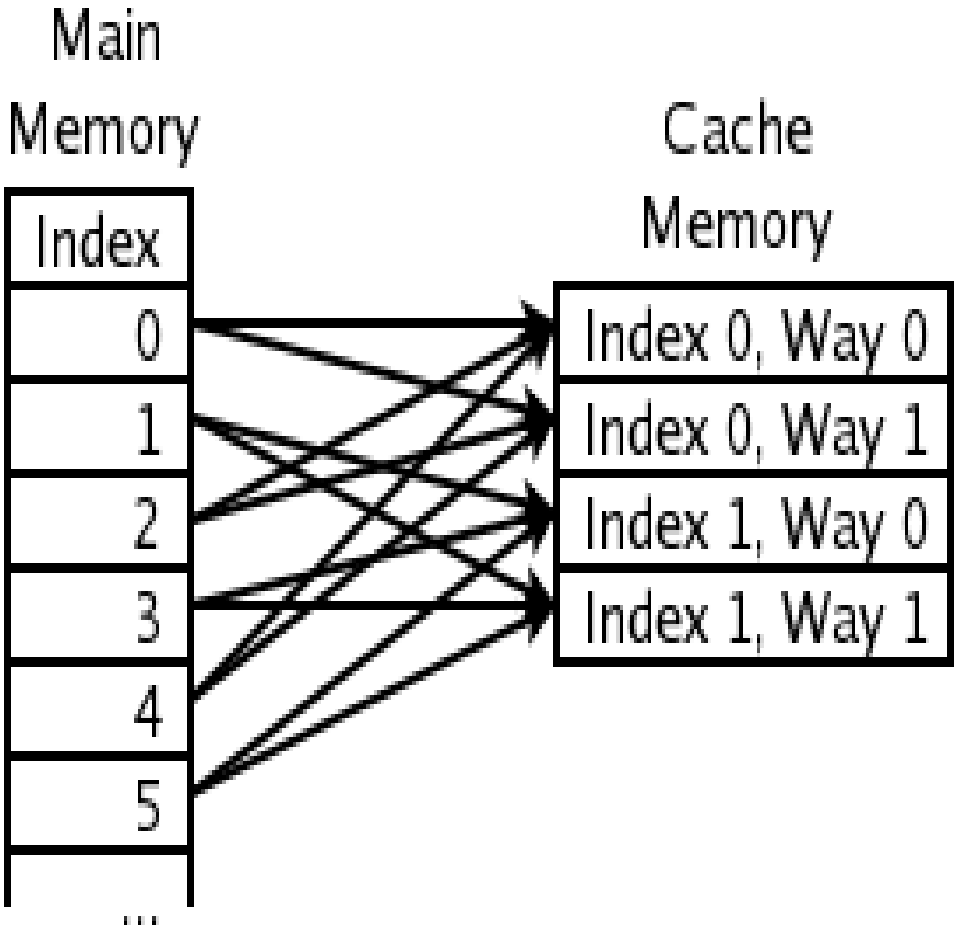
- Combination of direct and associative
- Direct-mapped set of associative caches
- *N-way* cache: n associative entries per direct address map

Direct Mapped Cache Fill



Each location in main memory can be cached by just one cache location.

2-Way Associative Cache Fill



Each location in main memory can be cached by one of two cache locations.

for highest speed CPU caches (previous entry always evicted)

for high speed CPU caches (1 bit for least recently used of 2)

Set-associative example

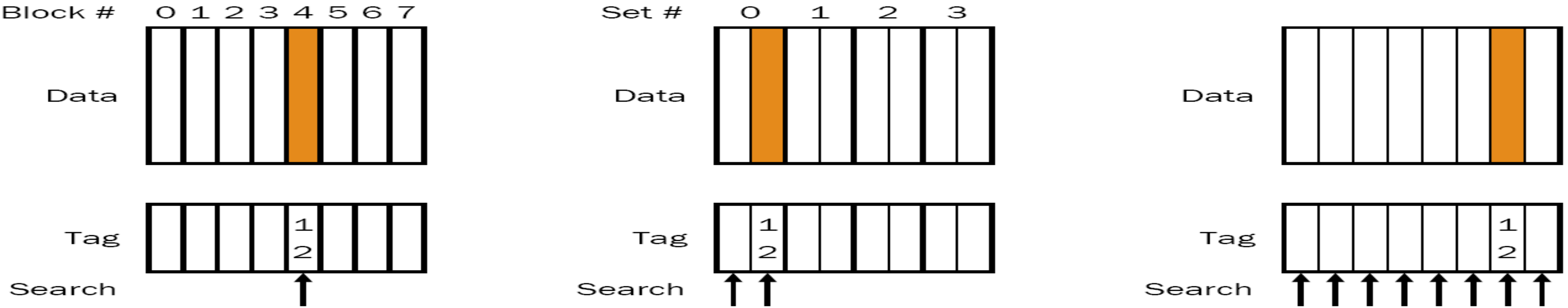
	V	Tag	Data	V	Tag	Data
7		0011			0101	
6		1010			1100	
5		1101			0001	
4		0101			0010	
3		1111			1001	
2		0000			0101	
1		1010			1111	
0		1101			0000	
	Entry A			Entry B		

Set-associative pros and cons

- Pros
  - High hit rate (8-way = 60% lower miss rate)
  - Easier to build / faster than fully-associative
- Cons
  - Slower access (if clocked)

Direct Mapped (D) vs Fully Associative (A) vs Set Associative (SA) Caching

- **D**: each address can be stored in one cache location
  - address = memory block (tag) + cache line (+ offset in line)
- 2 way **SA**: each address stored in 2 cache locations
  - address = memory block (tag) + set number (+ offset in line)
- **A**: any address can be stored in any cache location
  - address = line in memory (+ offset in line)
  - but all tags must be compared simultaneously (associatively)



Direct/Set/Associate Relationship

Conceptually, the direct mapped and fully associative caches are just "special cases" of the N-way set associative cache.

You can set "N" to 1 to make a "1-way" set associative cache. If you do this, then there is only one line/block per set, which is the same as a direct mapped cache because each memory address is back to pointing to only one possible cache location.

On the other hand, suppose you make "N" really large; say, you set "N" to be equal to the number of entries in the cache. If you do this, then you only have one set, containing all of the lines/blocks, and every memory location points to that huge set. This means that any memory address can be in that one (read any) cache entry, and you are back to a fully associative cache.

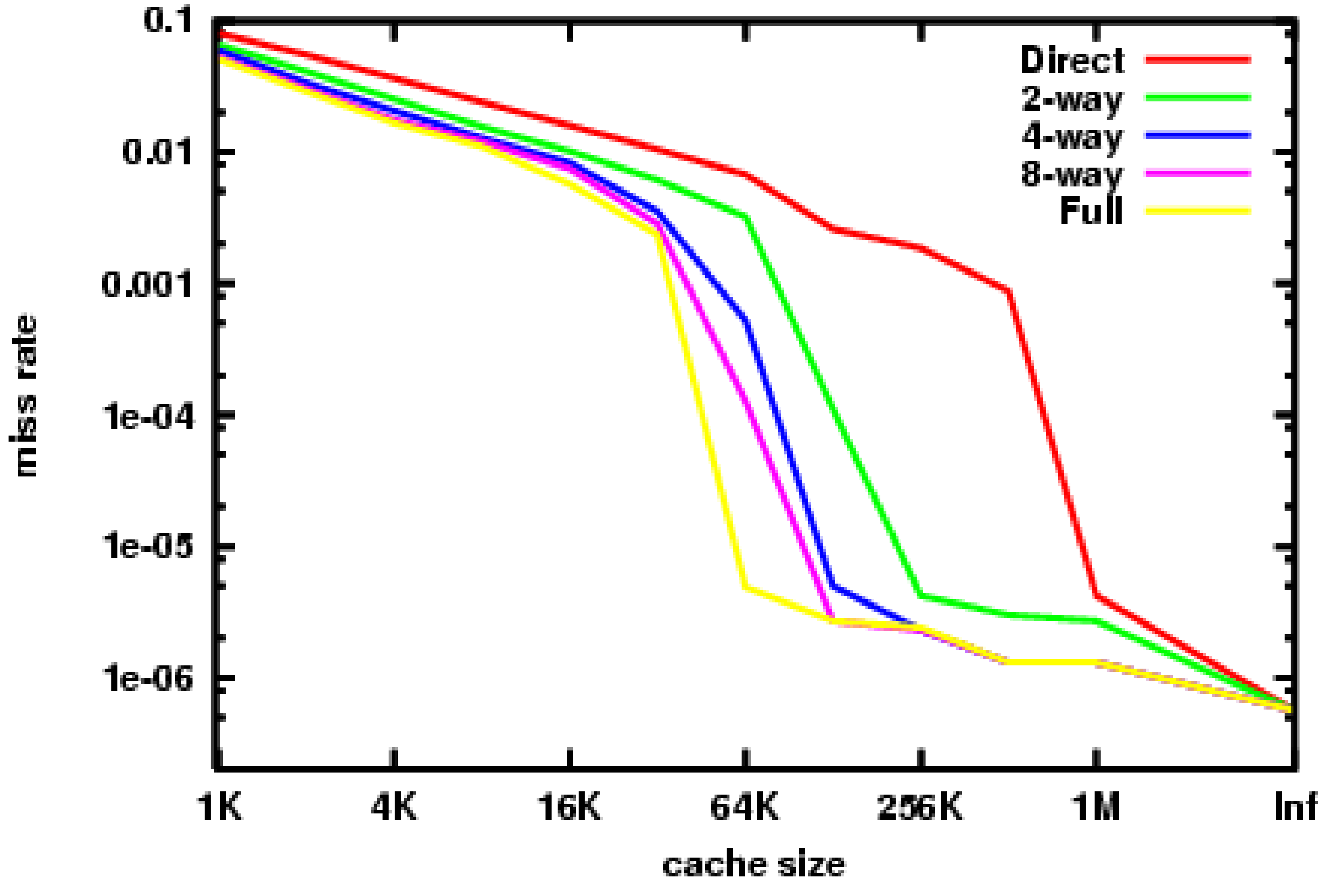
Cache Performance

Cache Type	Hit Ratio	Search Speed
Direct Mapped	Good	**Best**
Fully Associative	**Best**	Moderate
N-Way Set Associative, N>1	Very Good, Better as N Increases	Good, Worse as N Increases

Direct mapped and set associative caches are by far the most common.

Direct mapping is used more for level 3 or 4 caches on motherboards, while the higher-performance set-associative cache is found more commonly on the smaller primary caches contained within processors.

Also visit: www.pcguide.com/ref/mbsys/cache/func.htm



Miss rate versus cache size on the Integer portion of SPEC CPU2000

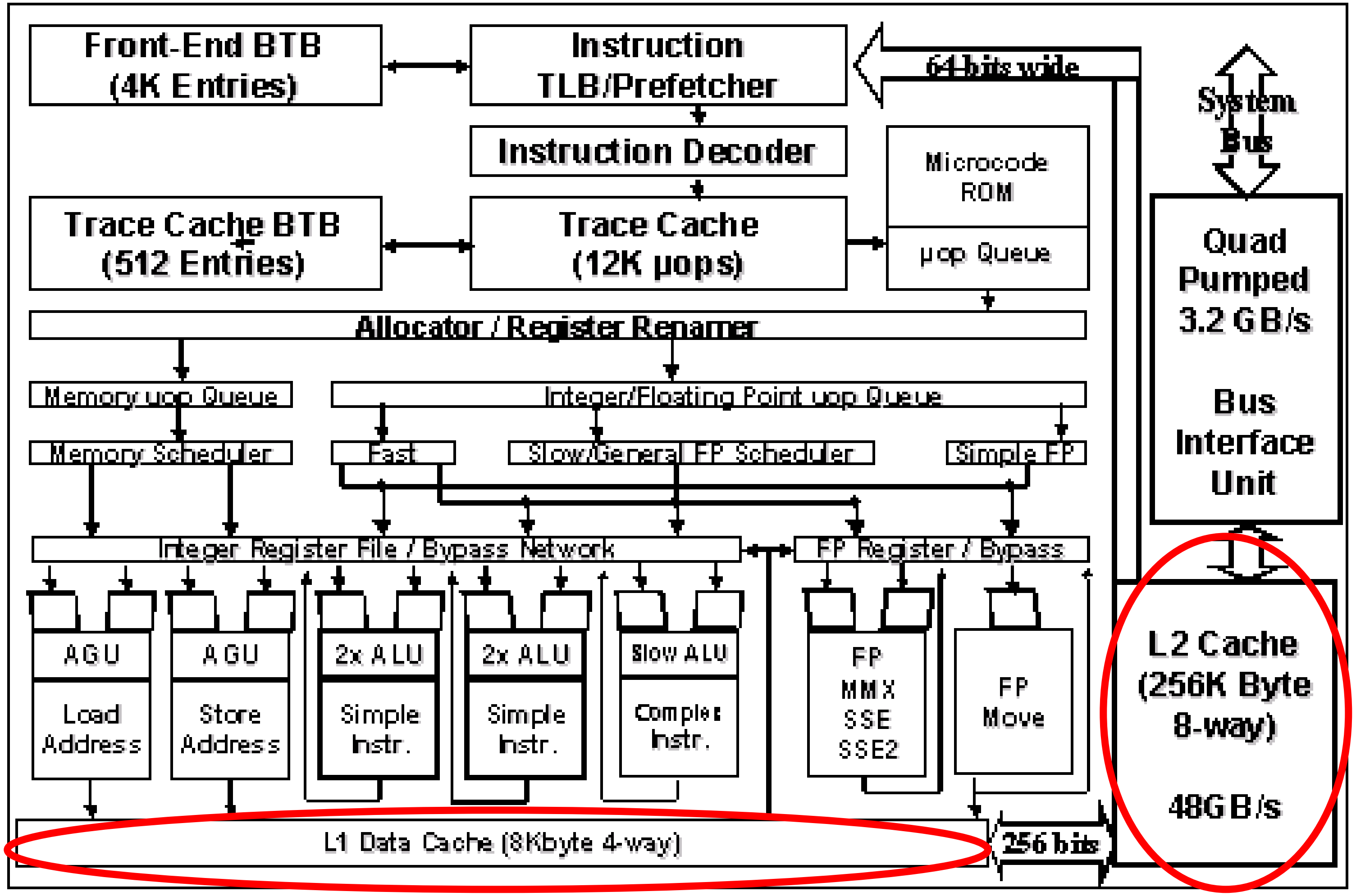
A clever compiler can take cache collisions into account, placing instructions and data in memory.

Cache architectures

- Multi-level:
  - L1 – on-die (private to each core)
  - L2 – on-die (private to each core)
  - L3 – on-die (shared between cores)
  - L4 – off-die, but on-package (caches L3 evictions)

- Unified vs split
  - Unified: better usage (% full)
  - Split: different types/policies

Example: Pentium (single core)



Cache Position

- Physical vs Virtual:
  - Physical:
    - address translated first
    - *Context* not required
  - Virtual:
    - In parallel with address translation
    - But: context required

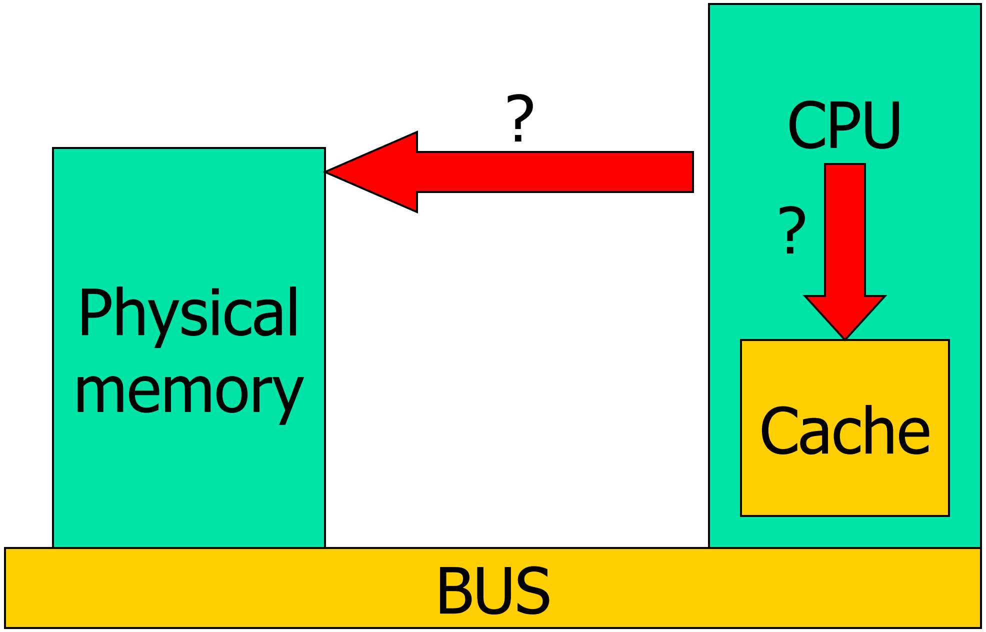
Replacement Policy

- Which cache entry to “evict”?
  - Direct mapping: no issue

  - Associative/set associative:
    - RANDOM
    - FIFO
    - Least recently used (LRU)
    - Least *frequently* used

And then it all went horribly wrong…

- Policy needed for writing to memory
  - Consistency – cache and memory “agree”
  - Speed - multiple writes



Write policies

- **Write-through**
  - always write to disk
- **Copy back** or **Write-back**
  - write to disk only when evicting (even if unchanged)
- **Write-deferred**
  - Mark cache entry as “dirty” (if changed)
  - Write out dirty entry on eviction
- **Write allocation**
  - bring missed entry into cache

Prefetching

- Load next *n* lines of memory into cache
- E.g. video
- May write over needed data
- Memory traffic increases
- Processor idle during fetch?
- Clever hardware can make worthwhile

Summary

- Most modern processors/operating systems use caches, usually a least two levels
- Cache design is complex
- Design affects performance
- Need to know cache design to optimise code

CACHE

Example Exam Question

The following is a fragment of a **16KByte, 2-way associative L1 cache with a line size of 16 bytes**, an update policy of write-deferred, write-allocate, and a replacement policy of least-recently-used (LRU).

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	101	Y	N	111	N	N
46	010	Y	N	110	Y	Y
45	011	Y	Y	000	N	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

For each of the following consecutive memory operations (in binary) for a 64KByte virtual memory space, indicate the ***number of memory transfers*** assuming that the bus width is 4 bytes, and show the **state of the "valid" and "dirty" bits** of the corresponding cache entry after the operation. ***Show your working***.

(a) **[2 marks]**	WRITE:	1010001011110100
(b) **[2 marks]**	READ:	1100001011100111
(c) **[2 marks]**	READ:	1100001011110000
(d) **[2 marks]**	WRITE:	0000001011111111
(e) **[2 marks]**	READ:	0100001011011010

Example Exam Question

The following is a fragment of a **16KByte, 2-way associative L1 cache with a line size of 16 bytes**, an update policy of write-deferred, write-allocate, and a replacement policy of least-recently-used (LRU).

Set	Tag	Valid?	Dirty?		Tag	Valid?	Dirty?	
47	101	Y	N	16KB	111	N	N	16KB
46	010	Y	N	16KB	110	Y	Y	16KB
45	011	Y	Y	16KB	000	N	N	16KB
44	111	Y	Y	16KB	101	Y	N	16KB
43	010	Y	N	16KB	111	Y	N	16KB

For each of the following consecutive memory operations (in binary) for a 64KByte virtual memory space, indicate the ***number of memory transfers*** assuming that the bus width is 4 bytes, and show the **state of the "valid" and "dirty" bits** of the corresponding cache entry after the operation. ***Show your working***.

(a) **[2 marks]**	WRITE:	1010001011110100
(b) **[2 marks]**	READ:	1100001011100111
(c) **[2 marks]**	READ:	1100001011110000
(d) **[2 marks]**	WRITE:	0000001011111111
(e) **[2 marks]**	READ:	0100001011011010

Solution

…16KByte, 2-way associative L1 cache with a line size of 16 bytes…

- 16 byte line/block size - **4** bits to address each byte
- 2-way associative means 32 bytes per set (2 x 16)
- 16KB cache @ 32 bytes per set = **512 sets** (**9** bits) (16KB/32B = $2^{14}/2^{5}$ = 2^{9} sets = 512 sets)
- 9 bits to represent set number leaves **3** bits for tag
- low **4** bits = **offset** (to address each byte in a line/block)
- next **9** bits = **set number** (cache entry number)
- high **3** bits = **tag** (block number in memory)

	3 bits	**9 bits**	**4 bits**
	tag	**set num**	**offset**
(a) WRITE	**101**	**000101111**	**0100**

Example Exam Question

The following is a fragment of a **16KByte, 2-way associative L1 cache with a line size of 16 bytes**, an update policy of write-deferred, write-allocate, and a replacement policy of least-recently-used (LRU).

Set	Tag	Valid?	Dirty?		Tag	Valid?	Dirty?	
511								
.								
.								
.								
47	101	Y	N	16KB	111	N	N	16KB
46	010	Y	N	16KB	110	Y	Y	16KB
45	011	Y	Y	16KB	000	N	N	16KB
44	111	Y	Y	16KB	101	Y	N	16KB
43	010	Y	N	16KB	111	Y	N	16KB
.								
.								
.								
0								

… for 512 sets …

Solution

	3 bit tag	9 bit set num	4 bit offset
(a) WRITE	**101**	**000101111**	**0100**

- Set number 101111_2 = set 47_{10} (decimal)
- Tag = **101** = Tag matched = Cache hit – that line/block now becomes dirty:

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	**101**	**Y**	**N**	111	N	N
46	010	Y	N	110	Y	Y
45	011	Y	Y	000	N	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

- No memory transfers because not write through.
- Valid = Y
- Dirty = Y

Solution

(b) READ: 110 000101110 0100

- Set number 101110_2 = set 46_{10}
- Tag = **110** = Tag matched = Cache hit:

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	101	Y	Y	111	N	N
46	010	Y	N	**110**	**Y**	**Y**
45	011	Y	Y	000	N	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

- No memory transfers required – cache remains the same
- Valid = Y
- Dirty = Y

Solution

(c) READ 110 000101111 0000

- Set number 101111_2 = set 47_{10}
- Tag = **110** = No tag matched = Cache miss as no tag 110 is found on set 47 – so need to bring one line/block into the cache to replace an invalid (empty) entry (or a least recently used entry):

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	101	Y	Y	**111**	**N**	**N**
46	010	Y	N	110	Y	Y
45	011	Y	Y	000	N	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

- 4 memory transfers because must read in 16 bytes over 32-bit bus
- Valid = Y
- Dirty = N

Solution

(d) WRITE 000 000101111 1111

- Set number 101111_2 = set 47_{10}
- Tag = **000** = No tag matched = Cache miss as no tag 000 is found on set 47 – so need to bring one line/block into the cache to replace the least recently used entry:

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	**101**	**Y**	**Y**	110	Y	N
46	010	Y	N	110	Y	Y
45	011	Y	Y	000	N	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

- 8 memory transfers: dirty & valid so write it out – 4 memory transfers and then bring in the new line/block – 4 memory transfers
- Valid = Y
- Dirty = Y

Solution

(e) READ 010 000101101 1010

- Set number 101101_2 = set 45_{10}
- Tag = **010** = No tag matched = Cache miss as no tag 010 is found on set 45 – so need to bring one line/block into the cache to replace an invalid (empty) entry or a least recently used entry:

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	000	Y	Y	110	Y	N
46	010	Y	N	110	Y	Y
45	011	Y	Y	**000**	**N**	**N**
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

- 4 memory transfers because must read in 16 bytes over 32-bit bus
- Valid = Y
- Dirty = N

Solution

	number of memory bus transfers
(a) Write set 47, tag 101	**0**
(b) Read set 46, tag 110	**0**
(c) Read set 47, tag 110	**4**
(d) Write set 47, tag 000	**16**
(d) Read set 45, tag 010	**4**

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	**000**	Y	**Y**	**110**	**Y**	N
46	010	Y	N	110	Y	Y
45	011	Y	Y	**010**	**Y**	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

Example Exam Question

Explain what each of the following mean, and give *one example for each* of how they are taken advantage of in a modern computer system (note: it is *not* sufficient to name a processor/operating system that uses it):

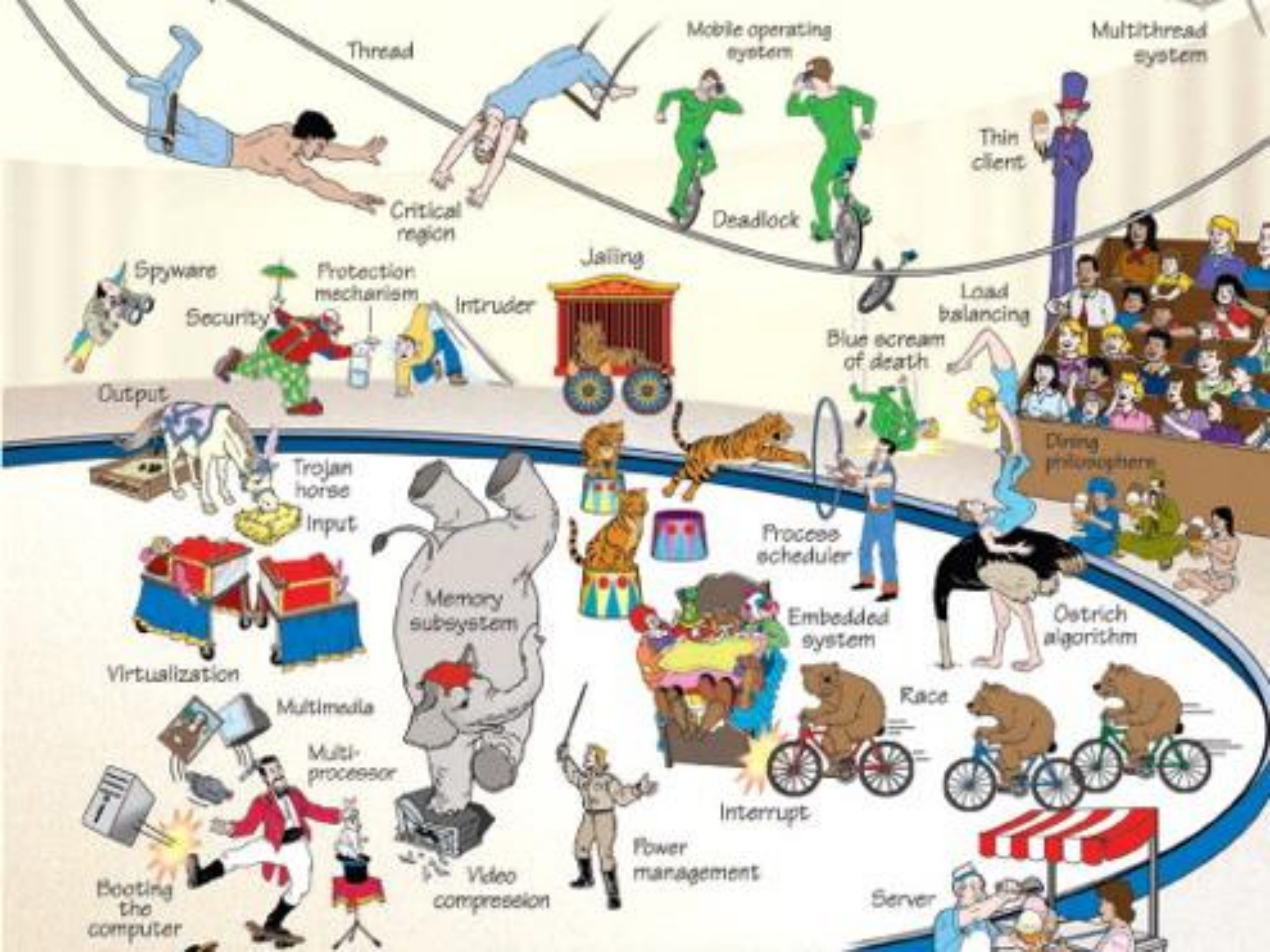
- (a) **[2 marks]** Delegation
- (b) **[2 marks]** Locality of reference

Solution: Delegation

- The CPU/processor offloads (delegates) work to co-processors.
- Examples: Disk controller, FPU, video/graphics card

Solution: Locality of reference

- Memory accesses (data or instructions) are likely to be near (spatially and temporally).
- Example: Caches take advantage of this to speed up memory access by paging in surrounding memory.


Thread
Mobile operating system
Multithread system
Critical region
Deadlock
Thin client
Spyware
Protection mechanism
Security
Intruder
Jailing
Load balancing
Blue scream of death
Output
Dining philosophers
Trojan horse
Input
Process scheduler
Memory subsystem
Embedded system
Ostrich algorithm
Virtualization
Race
Multimedia
Multi-processor
Interrupt
Power management
Video compression
Booting the computer
Server