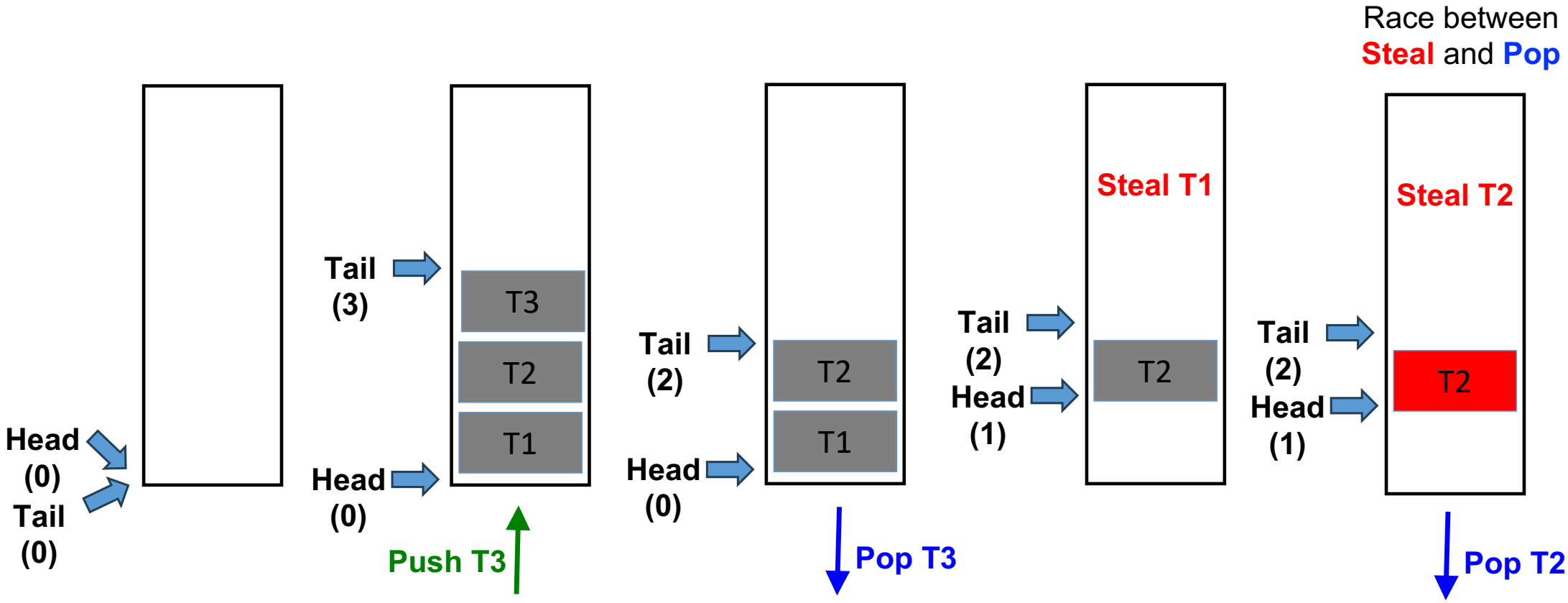
Lecture 16: Cache Coherency

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Class

- Lock free work-stealing deque (left over from last lecture)
- Cache coherency
  - MSI protocol
  - MESI protocol
- Writing cache friendly code

Lock Free Work-Stealing Deque (Chase-Lev)

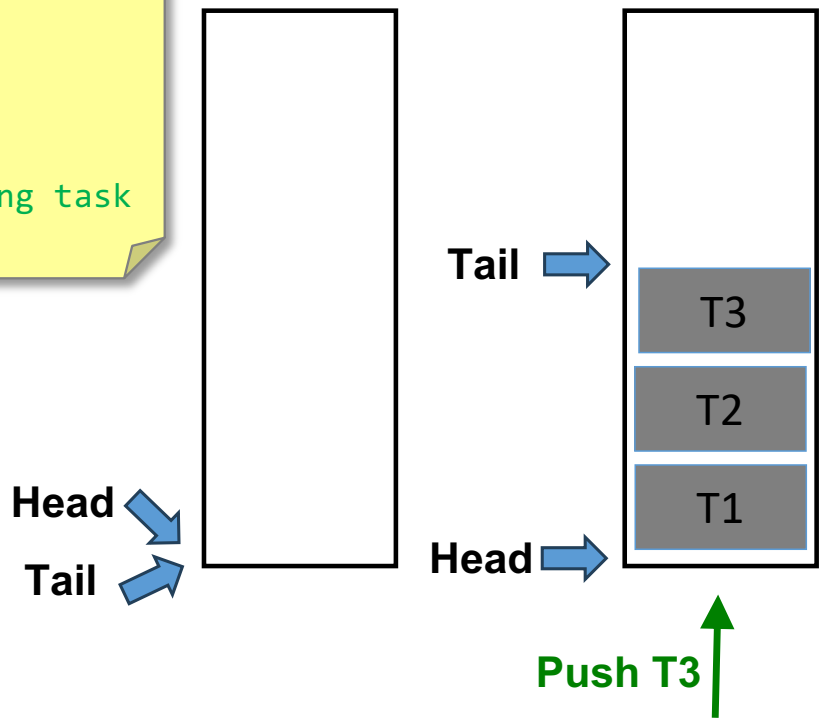


The push Operation

```
void push(deque_t* deque, task_t* task) {
  int head = deque->head.load();
  int tail = deque->tail.load();
  if((tail – head) == SIZE) throw_error(“Deque Full”);
  deque->buffer[tail % SIZE] = task; // First store the task
  deque->tail.store(tail + 1);       // Increment after storing task
}
```

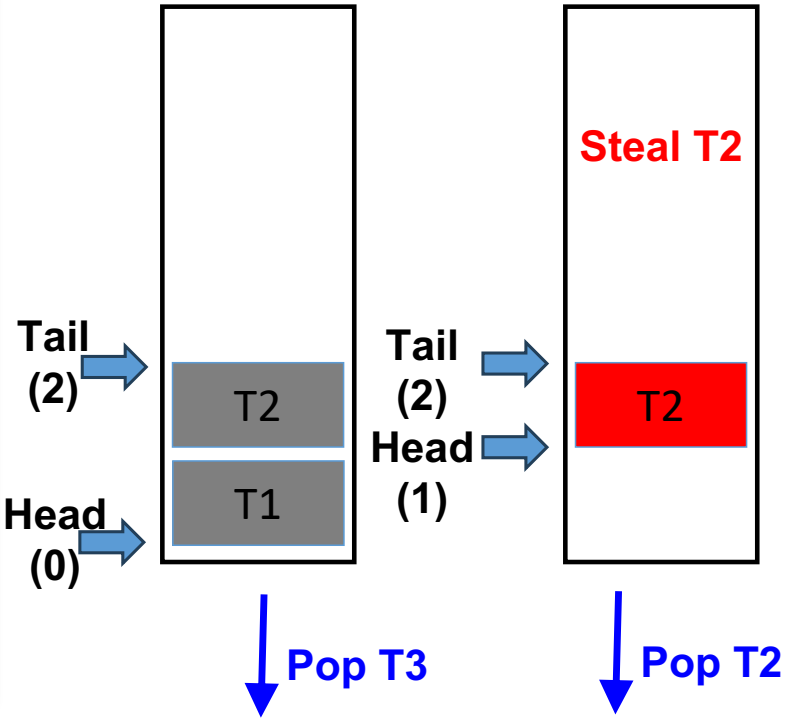
std::atomic<int> tail, head;

std::memory_order_seq_cst
by default everywhere



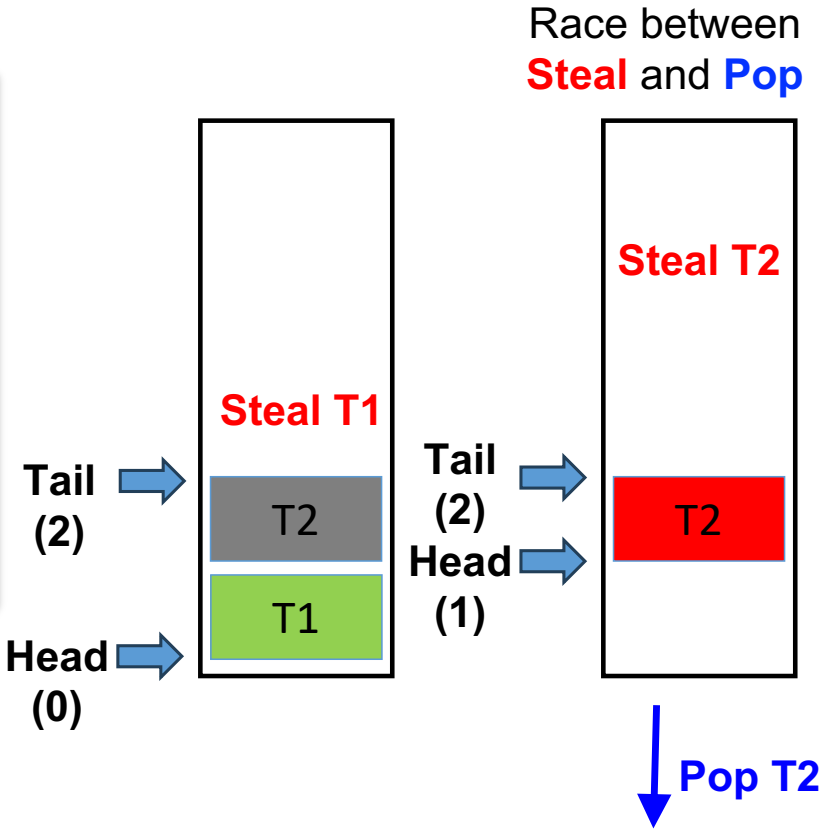
The pop Operation

```
task_t* pop(deque_t* deque) {
  int head = deque->head.load();
  int tail = deque->tail.load();
  deque->tail.store(tail – 1);
  task_t* task = deque->buffer[tail];
  if(tail < head) { // Deque just got empty
    deque->tail.store(head) // Both head and tail point to same slot
    task = NULL;
  }
  else if(tail > head) { // Sufficient tasks available on deque
    // Do nothing. We are all set
  } else { // Only one task remaining – race with thief
    if(!deque->head.compare_and_exchange_strong(head, head + 1)) {
      task = NULL; // Thief won the race
    }
    deque->tail.store(tail + 1); // increment tail to match head
  }
  return task;
}
```



The steal Operation

```
task_t* steal(deque_t* deque) {
  int head = deque->head.load();
  int tail = deque->tail.load();
  if((tail <= head) {  // No task available
    return NULL;
  } else {           // Tasks are available (count doesn't matter)
    task_t* task = deque->buffer(head % SIZE];
    if(!deque->head.compare_and_exchange_strong(head, head + 1)) {
      task = NULL; // Race lost with either victim or other thief
    }
    return task;
  }
}
```



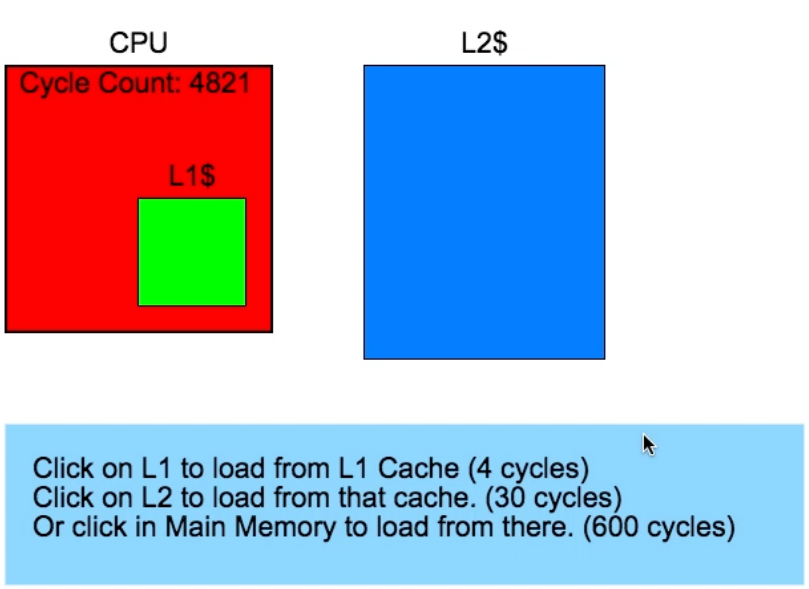
Using release–acquire Ordering

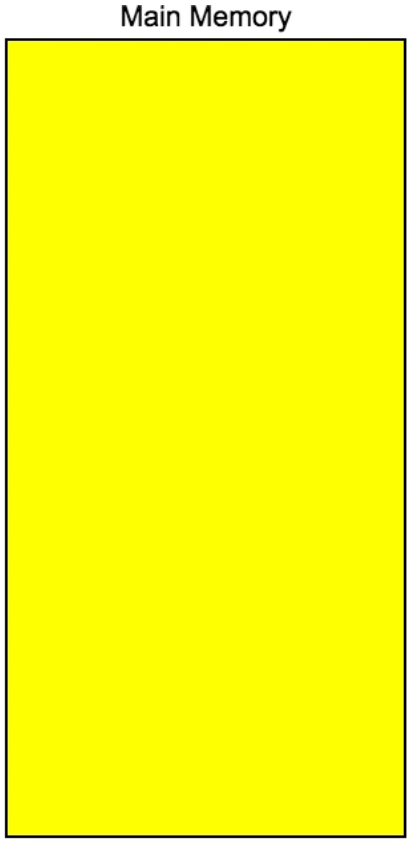
- Loads from “tail” follows **relaxed** ordering at victim
  - As only updated locally by victim
- Loads from “head” follows **relaxed** ordering at both victim and thief
- Stores to “tail” from victim (**release**) should synchronize with load from “tail” at thief (**acquire**)
  - Thief should be aware of updated value of “tail” at steal
- Stores on “head” at both victim and thief must use CAS operation (**acquire_release**)

Today's Class

- Lock free work-stealing deque (left over from last lecture)
- Cache coherency
  - MSI protocol
  - MESI protocol
- Writing cache friendly code

How Bad is Memory Access Latency?

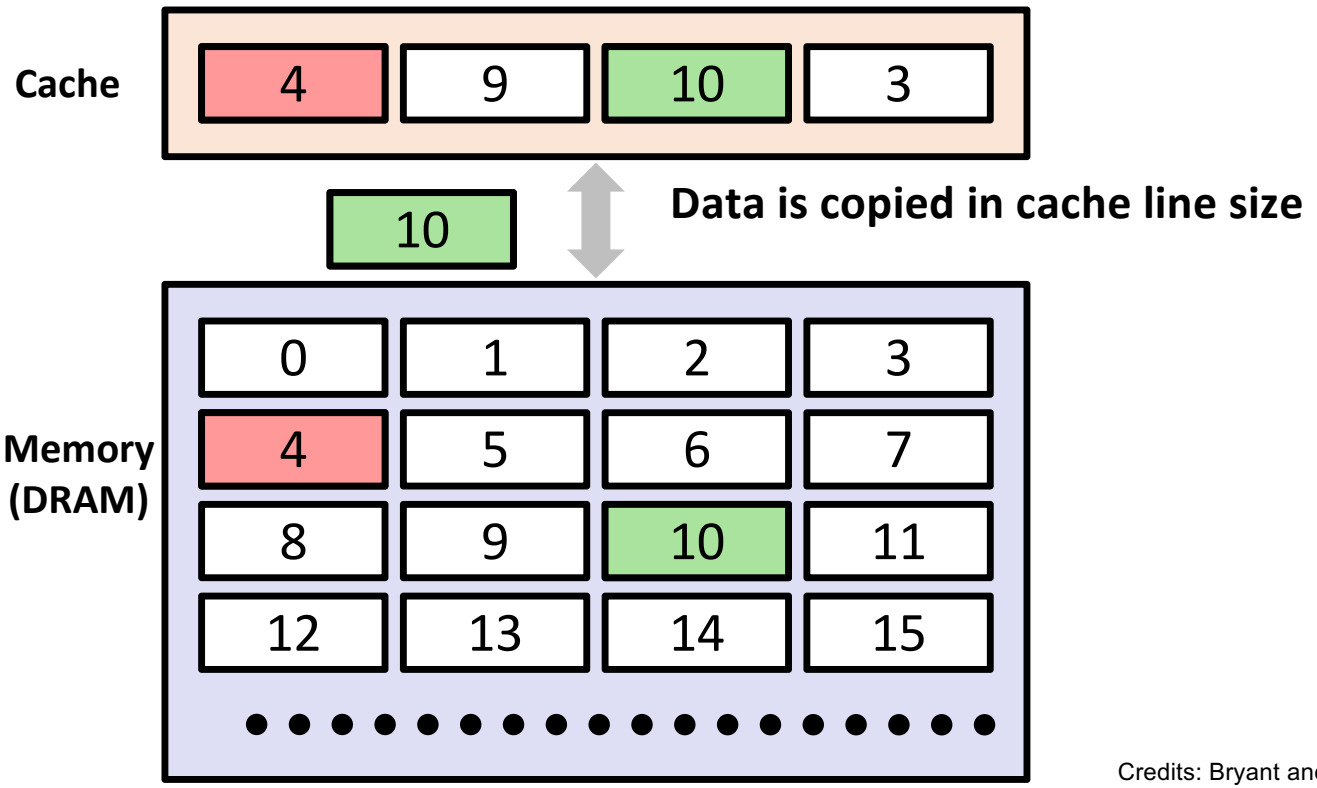




- Another analogy
  - Normalizing with L1 latency, and assuming one seconds is equal to 4 cycles
    - L1 = one second
    - L2 = 7.5 seconds
    - Main memory = 2.5 minutes
    - Hard drive = in several days!

Animation source: https://overbyte.com.au/misc/Lesson3/CacheFun.html

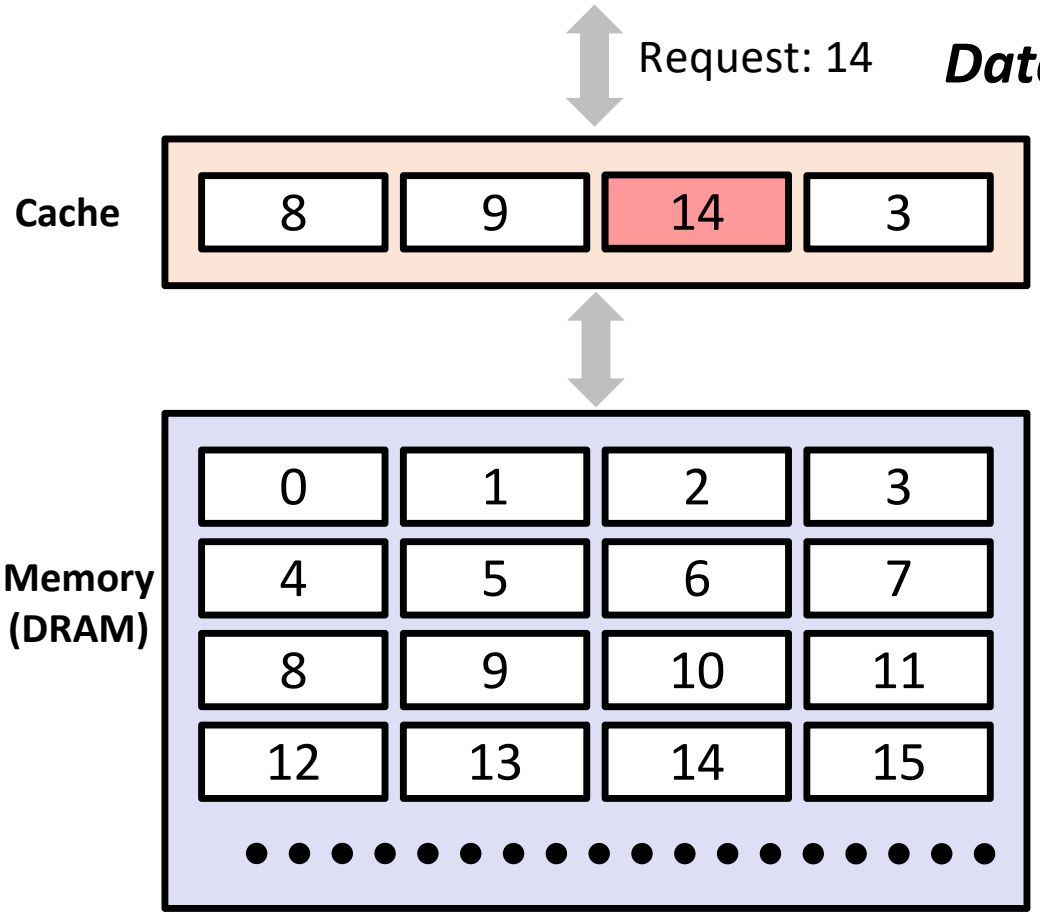
General Cache Concepts



- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device
  - Temporal and spatial locality

Credits: Bryant and O'Hallaron, Lecuture 9, CMU 15-213/18-243

General Cache Concepts



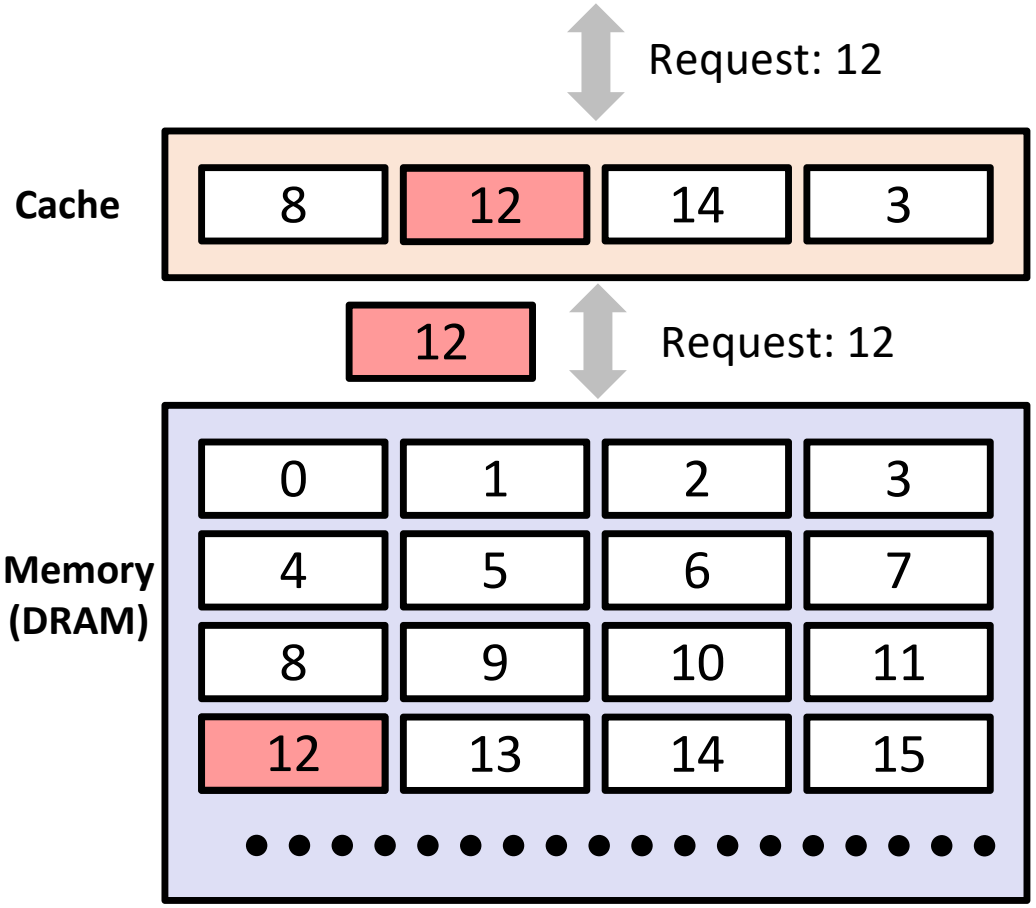
Line b is in cache:

Hit!

- Cache hit
  - Data is already in the cache in some cache line
    - Cache line size is 64 bytes on x86 processors
    - Cache line is the smallest granularity of load/store of any memory block from DRAM

Credits: Bryant and O'Hallaron, Lecuture 9, CMU 15-213/18-243

General Cache Concepts



Data in line b is needed

Line b is not in cache:
Miss!

Line b is fetched from *memory*

Line b is stored in cache
By evicting some old line

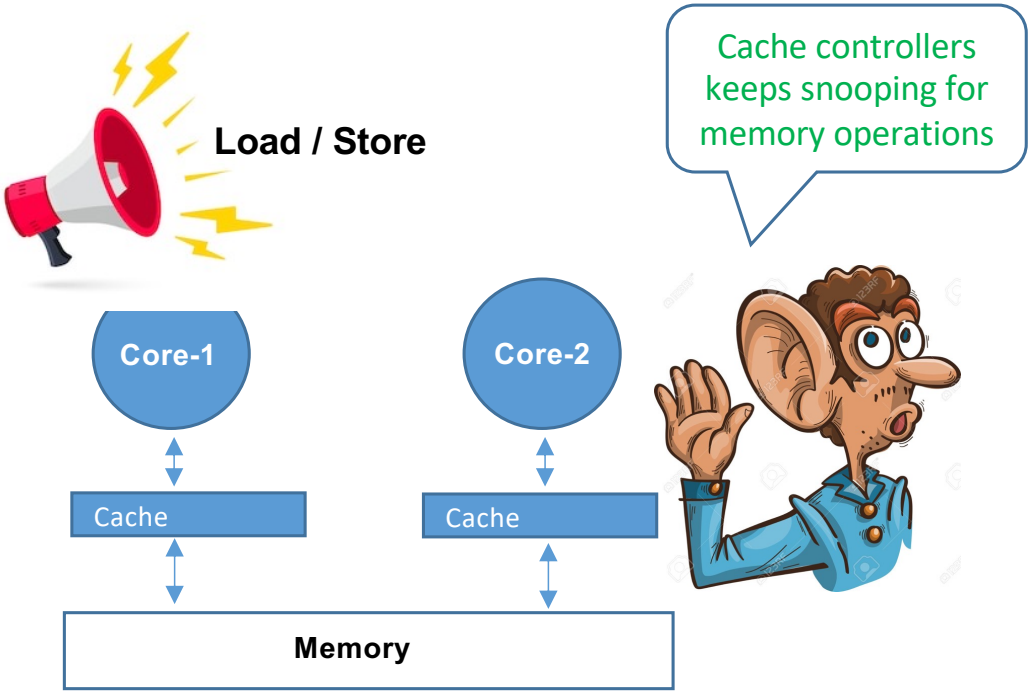
Credits: Bryant and O'Hallaron, Lecuture 9, CMU 15-213/18-243

Defining Cache Coherence Differently (As we have studied Memory Consistency)

- **Program order** must be maintained at a single processor
  - A read by processor P to address X that follows a write by P to address X, should return the value of the write by P
    - Assuming no other processor wrote to X in between
- **Write propagation** to other processors
  - A read by processor P1 to address X that follows a write by processor P2 to X returns the written value... if the read and write are "sufficiently separated" in time (store buffers!)
    - Assuming no other writes to X occurs in between
- **Write serialization**
  - Writes to the same address are serialized: two writes to address X by any two processors are observed in the same order by all processors
    - E.g., if values 1 and then 2 are written to address X, no processor observes X having value 2 before value 1

Credits: Fatahalian and Bryant, CMU 15-418/618

Coherence using Private Caches

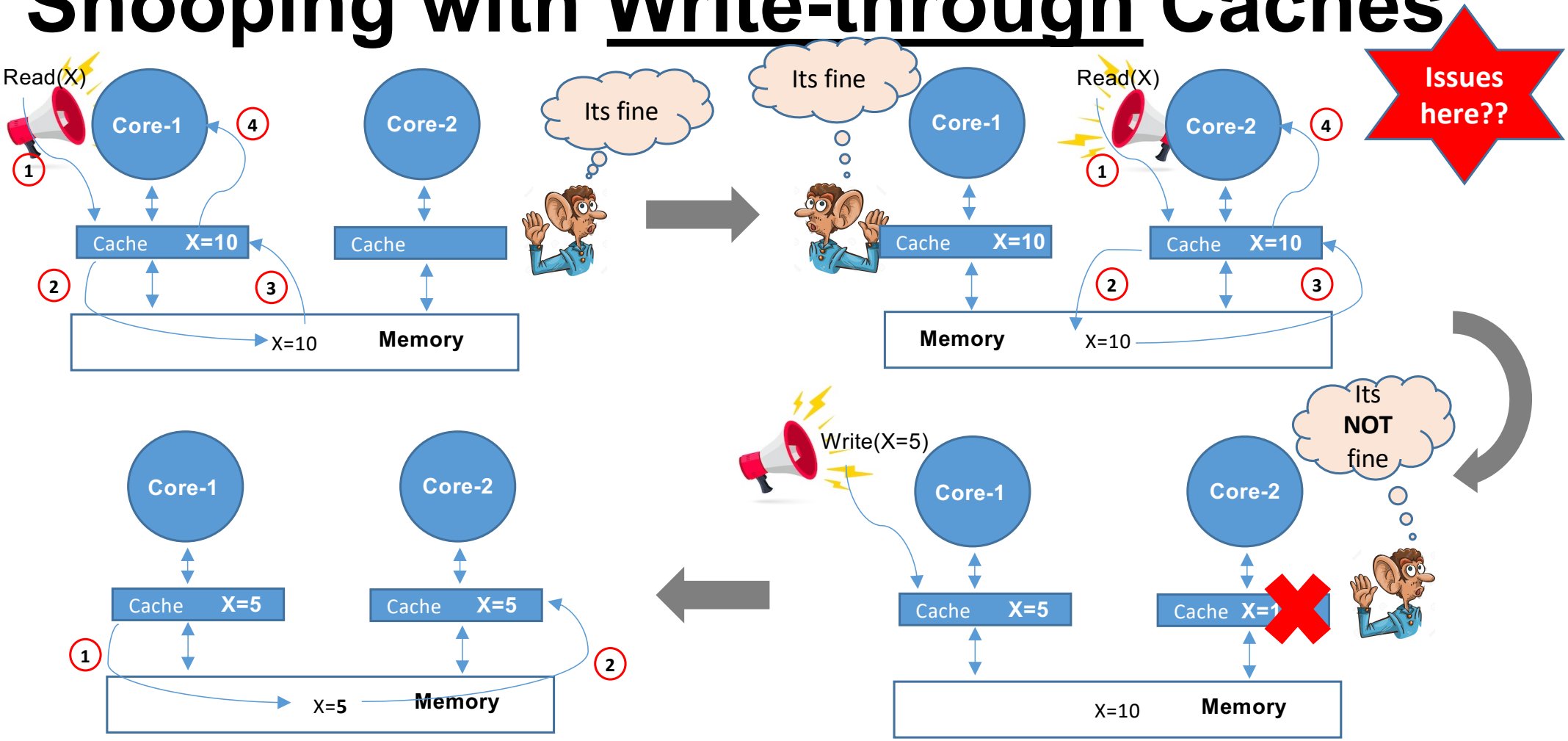


- Snooping based coherency protocol
  - Each core's cache controller broadcast any memory operations it wishes to perform before actually carrying out that operation
  - Rest of the core's cache controllers having that memory acts like a good citizen and help others to carry out their intended operation

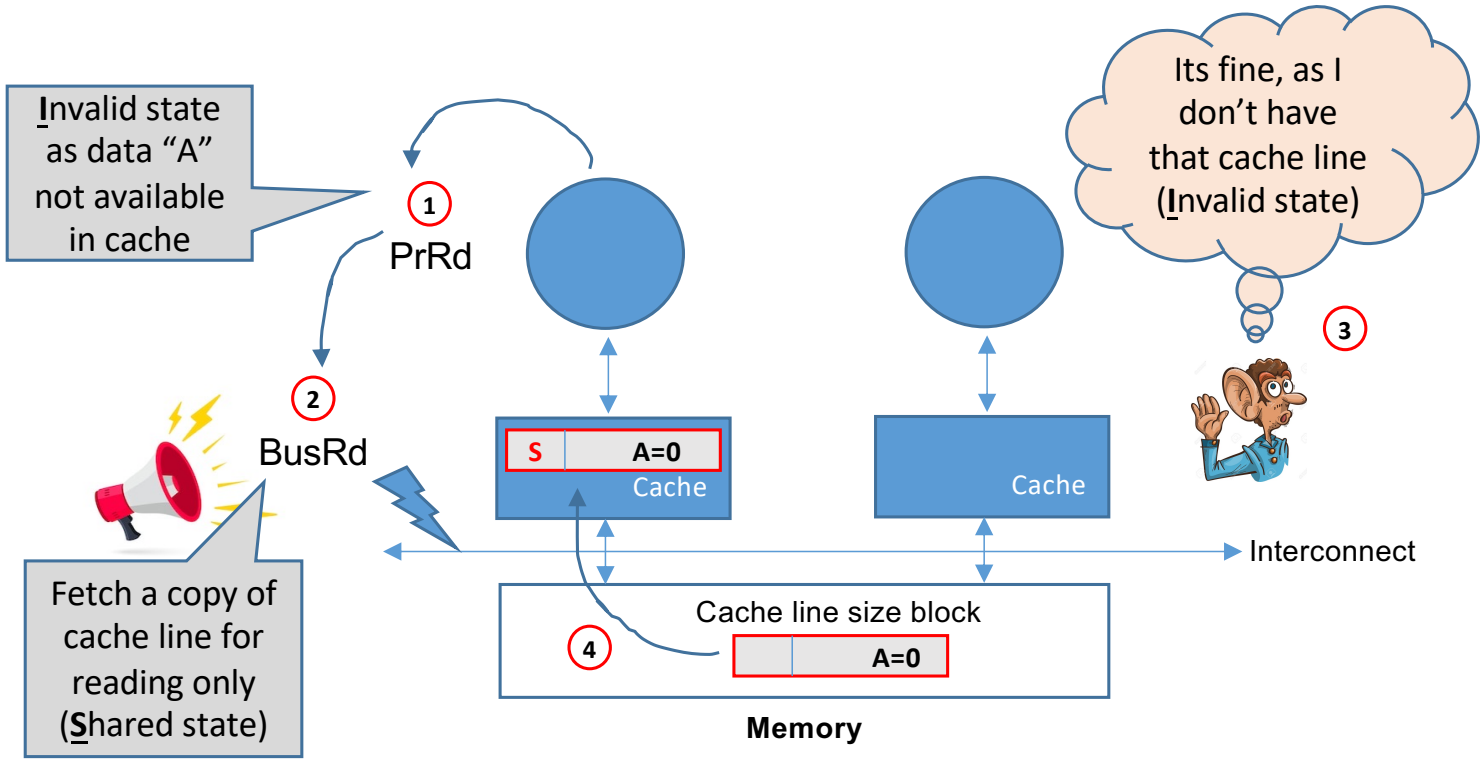
Private v/s Shared Cache Coherency

- Coherency using shared cache
  - Cache just have to look up to the processor and do the load/store instructions issued by the processor
- Coherency using private caches
  - Each cache has its own core to which it look after, but it also pays attention to what is going on in other caches or what is going over the interconnect
    - They are snooping!

Snooping with Write-through Caches

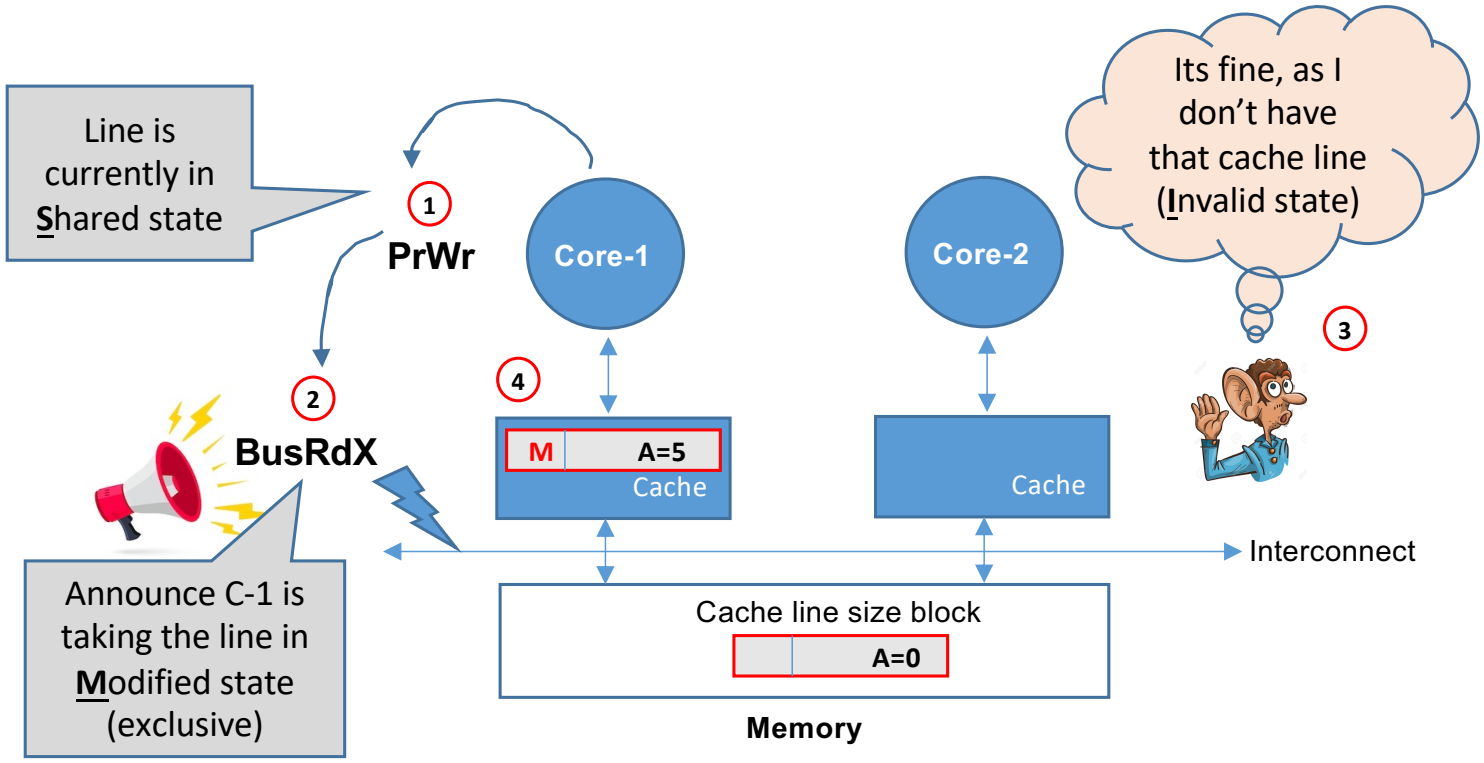


Snooping using Write-back Caches (1/5)



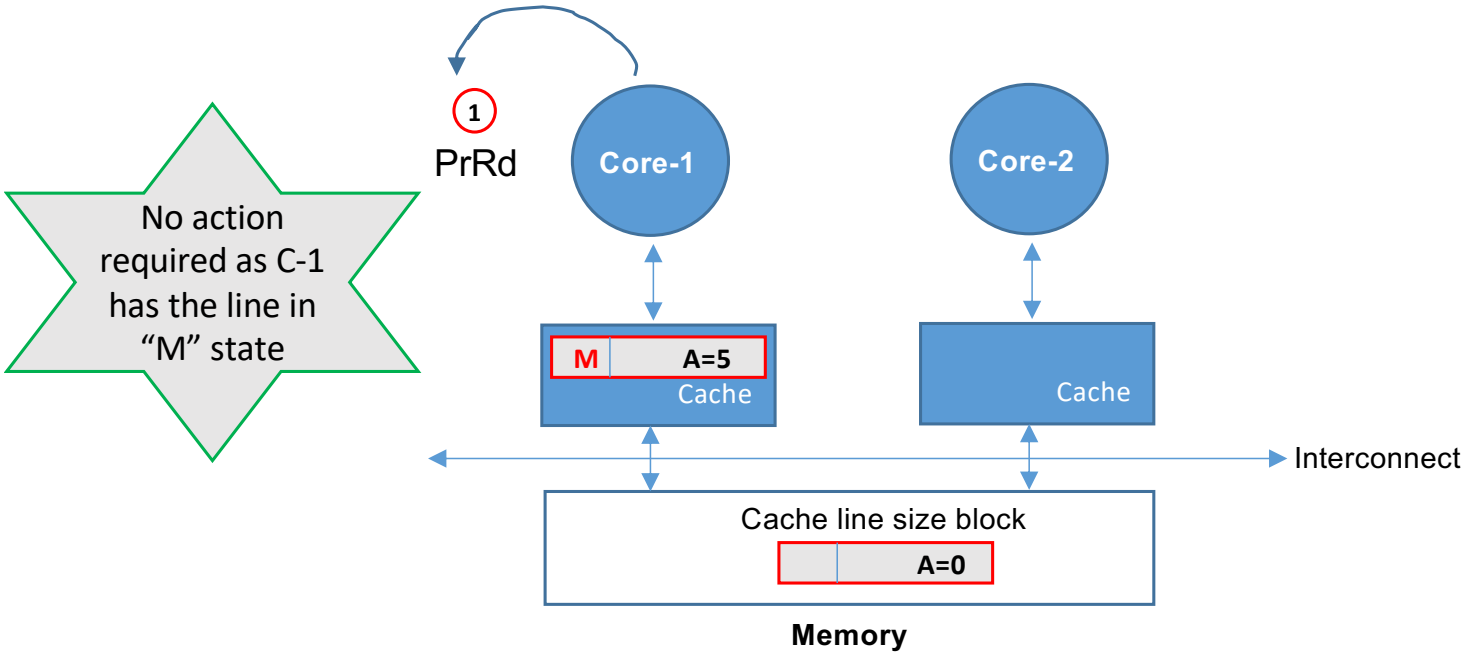
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

Snooping using Write-back Caches (2/5)



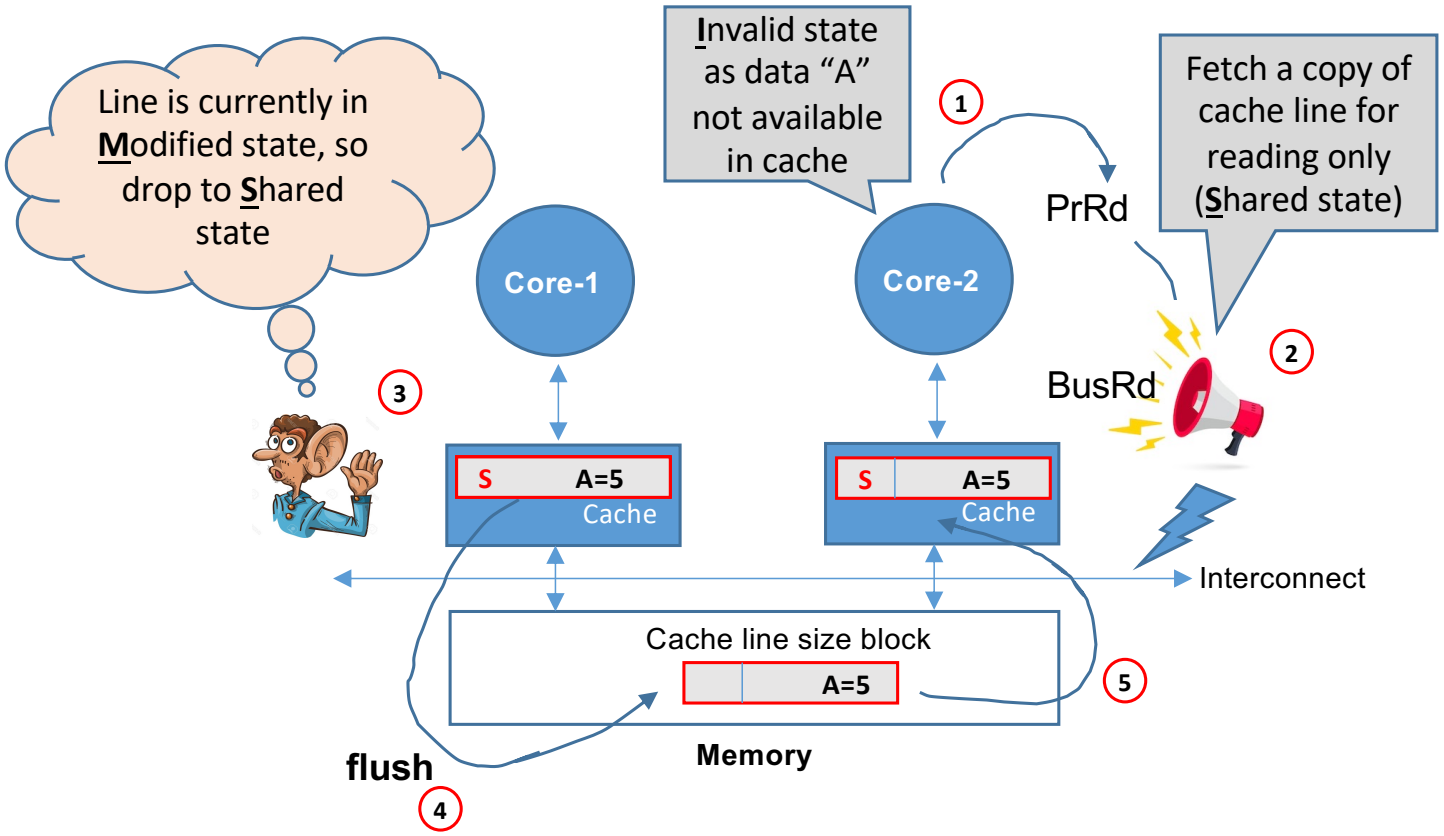
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

Snooping using Write-back Caches (3/5)



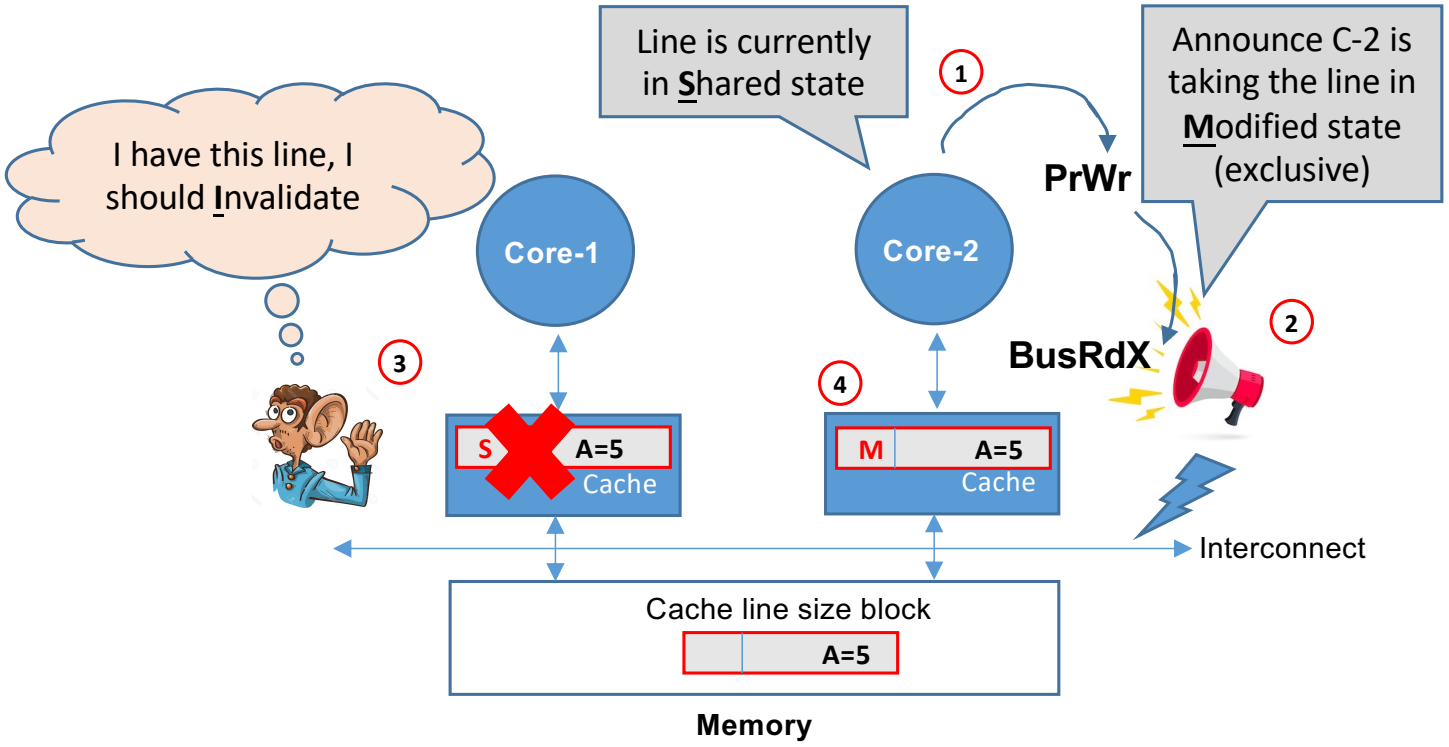
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

Snooping using Write-back Caches (4/5)



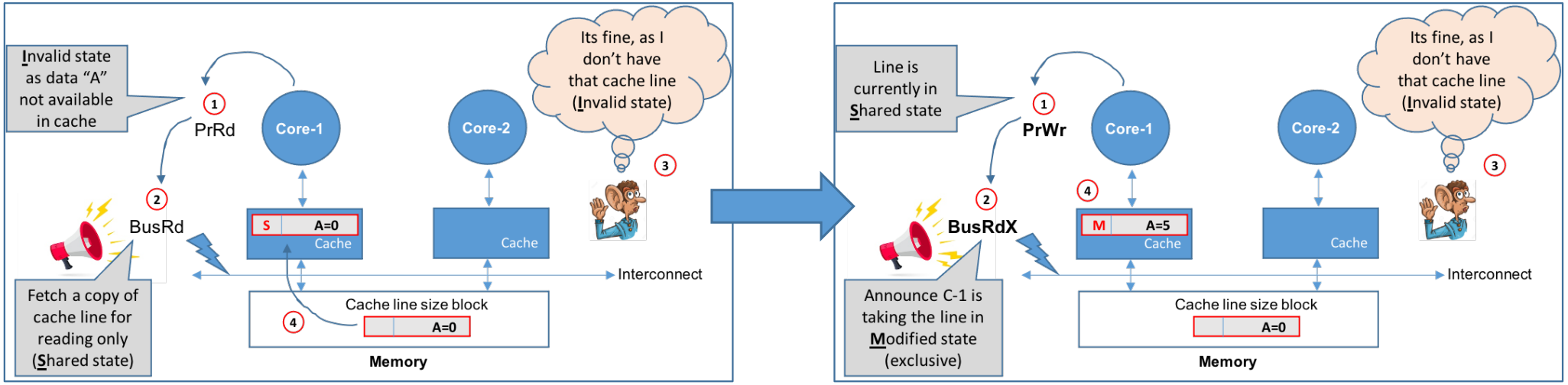
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

Snooping using Write-back Caches (5/5)



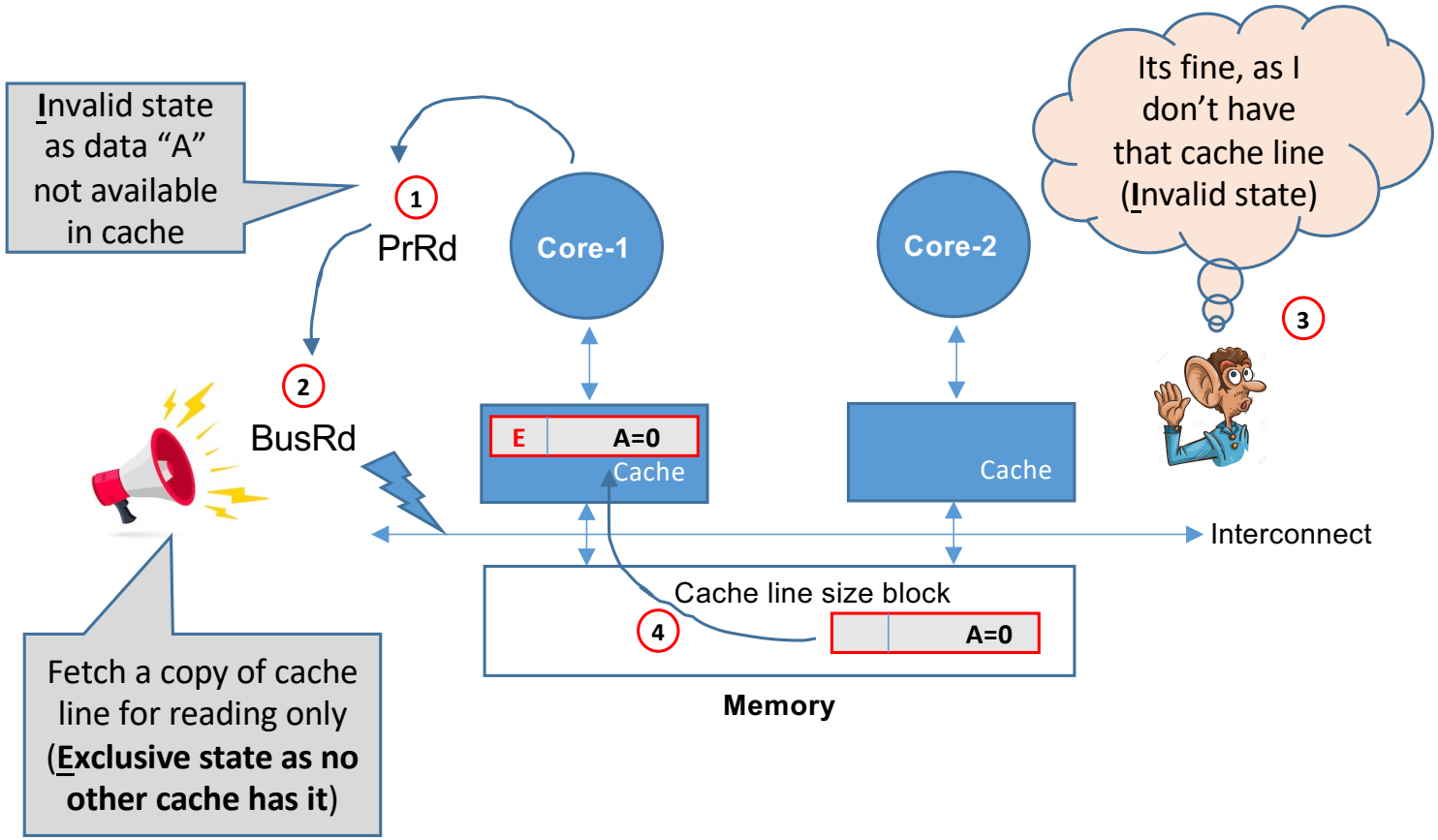
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

What is Wrong with MSI?



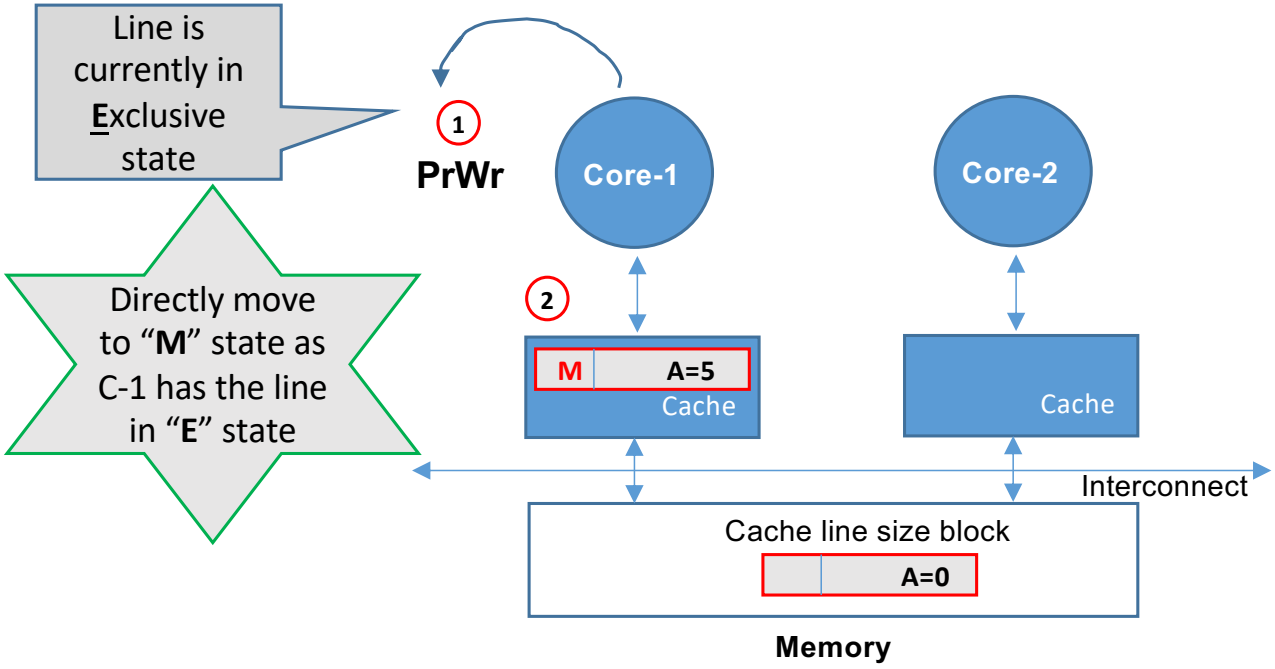
- Core-1 reads a data, and then wishes to modify it
  - The line is only in Core-1's, but it's cache controller still has to perform BusRdX operation for moving the line from "S" state to "M" state
    - Redundant traffic over the interconnect

MESI Protocol (1/3)



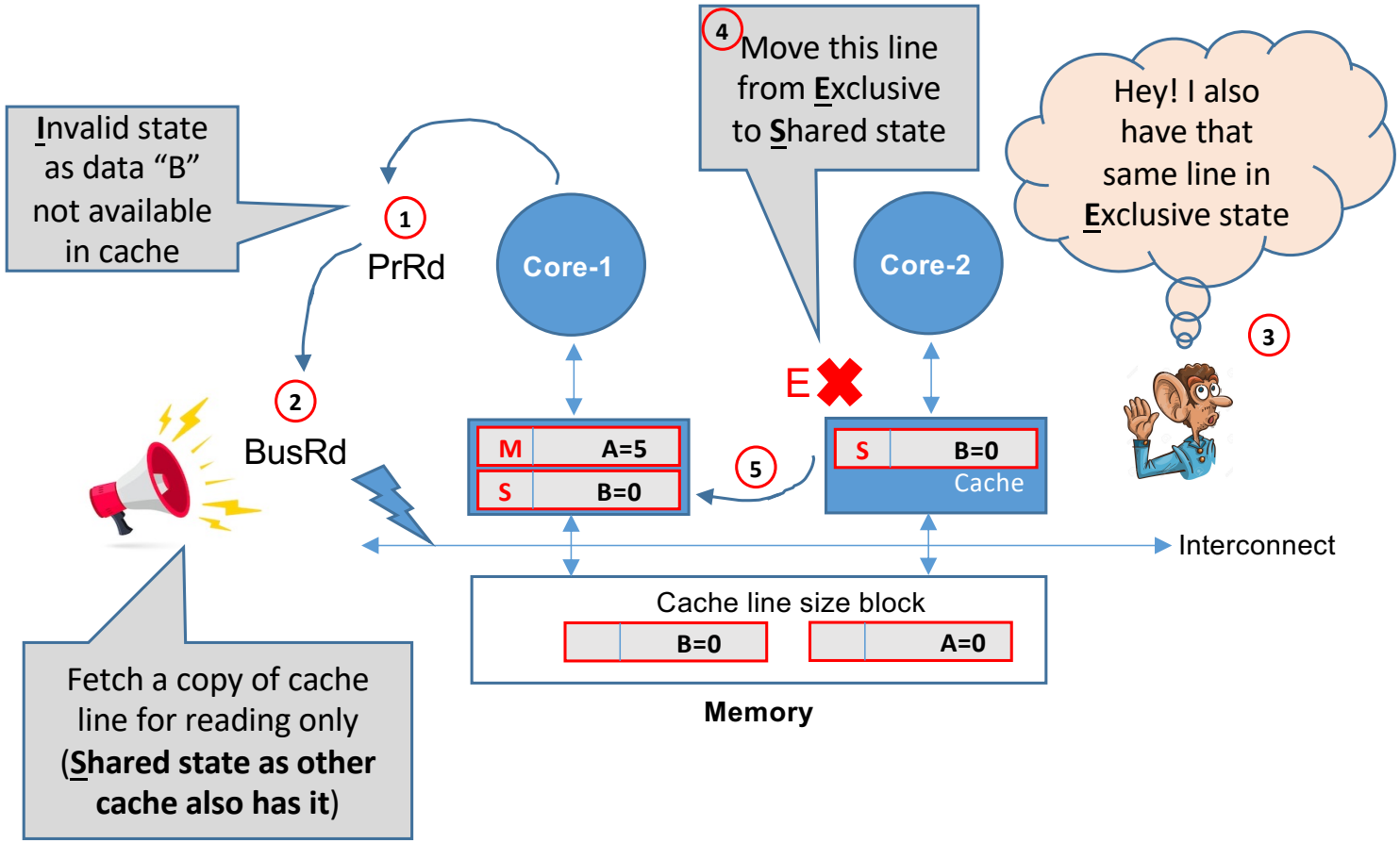
- An additional state **E**
  - Exclusive clean
  - Implies no other cache has a copy of this line

MESI Protocol (2/3)



- Moving from **E** to **M** state
  - No action required to be performed on interconnect
  - Present **E** state implies the line is not in any other cache

MESI Protocol (3/3)



- C-1 wants to read a line (B) which C-2 also has (E state)
  - C-1 cannot have it in **E** state, as C-2 also wants to own it for read purpose
  - C-2 drops it from E to S state
  - C-1 has the line in S state

Today's Class

- Lock free work-stealing deque (left over from last lecture)
- Cache coherency
  - MSI protocol
  - MESI protocol
- Writing cache friendly code

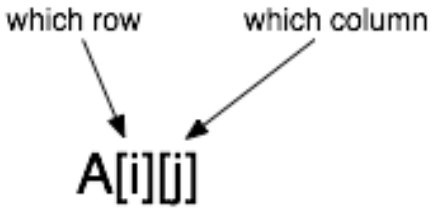
Writing Cache Friendly Code

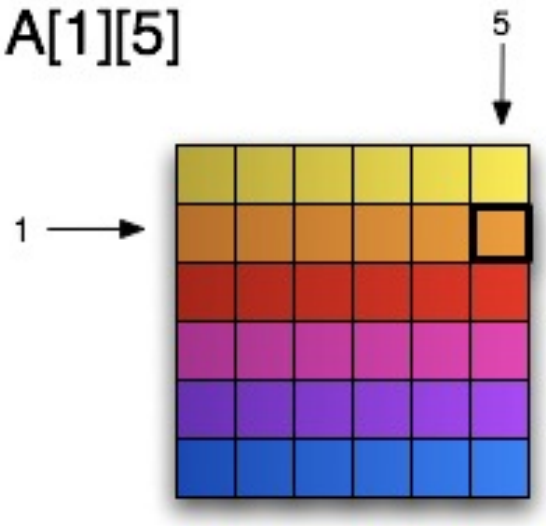
Two major rules:

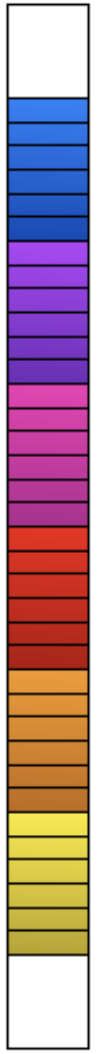
- Repeated references to data are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
  - Stride-1: i++
  - Stride-2: i+=2
  - ….

Layout of C Arrays in Memory

- C arrays allocated in row-major order
  - each row in contiguous memory locations

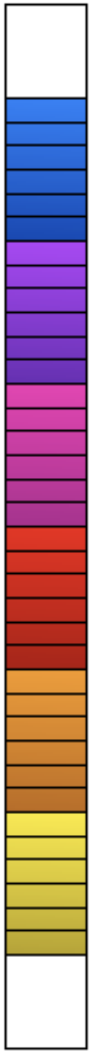






Layout of C Arrays in Memory

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    - `sum += a[0][i];`
  - Accesses successive elements
  - Exploits spatial locality (cache lines size here is 16 bytes)
    - miss rate = 4 bytes / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    - `sum += a[i][0];`
  - Accesses distant elements
  - No spatial locality!
    - miss rate = 1 (i.e. 100%)



Writing Cache Friendly Code

What is the miss rate in both the cases?

Two major rules:

- Repeated references to data are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Example: 4-byte int and cache line size of 4-ints

```
int sum_array_rows(int a[M][N]){
  int i, j, sum = 0;
  for (i = 0; i < M; i++)
     for (j = 0; j < N; j++)
        sum += a[i][j];
  return sum;
}
```

Miss rate = 1/4 = 25%

```
int sum_array_cols(int a[M][N]){
  int i, j, sum = 0;
  for (j = 0; j < N; j++)
     for (i = 0; i < M; i++)
        sum += a[i][j];
  return sum;
}
```

Miss rate = 100%

Which is Better?

```
float A[n], B[n], C[n], D[n]; //
initialized

for(int i=0; i<n; i++) {
    D[i] = A[i] + B[i] + C[i];
}
```

v/s

```
typedef struct Triplet {
    float A;
    float B;
    float C;
    float D;
} Triplet;

Triplet T[n]; // initialized

for(int i=0; i<n; i++) {
    T[i].D = T[i].A + T[i].B + T[i].D;
}
```

Next Lecture

- False sharing