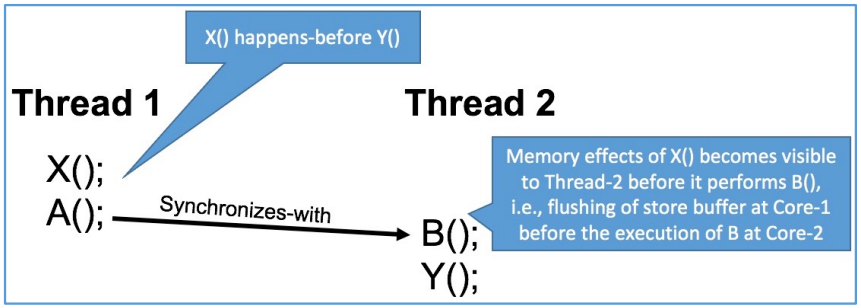
Lecture 24: Cache Coherency and False Sharing

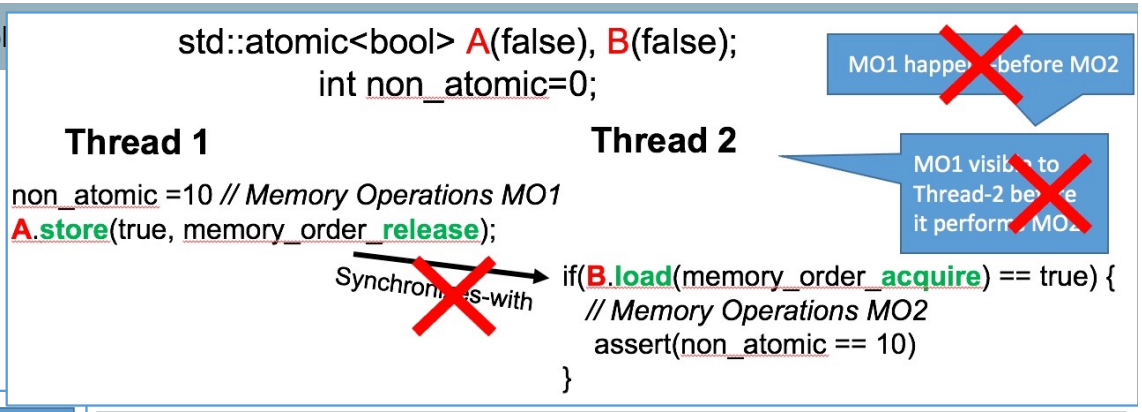
Vivek Kumar

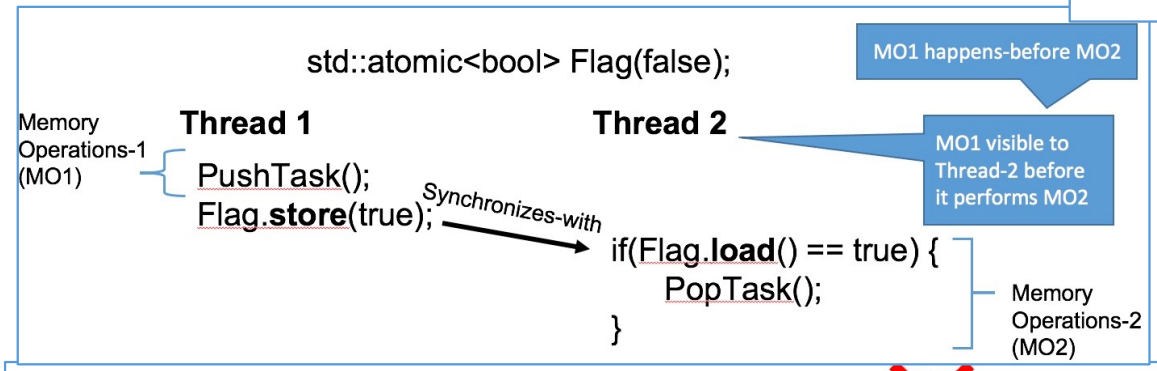
Computer Science and Engineering

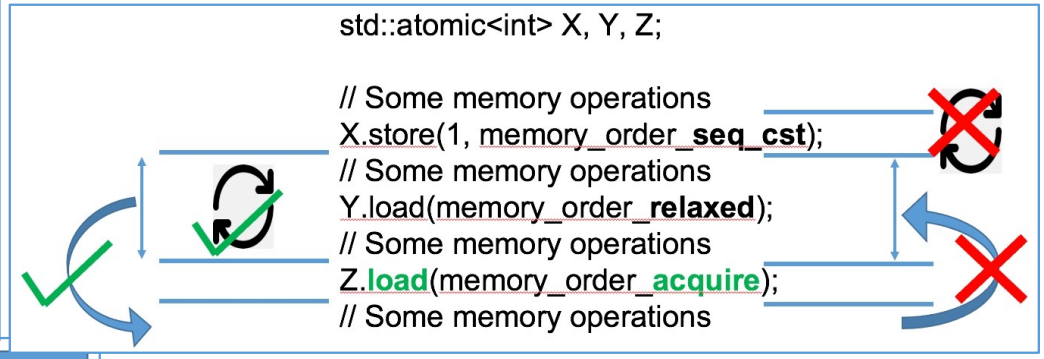
IIIT Delhi

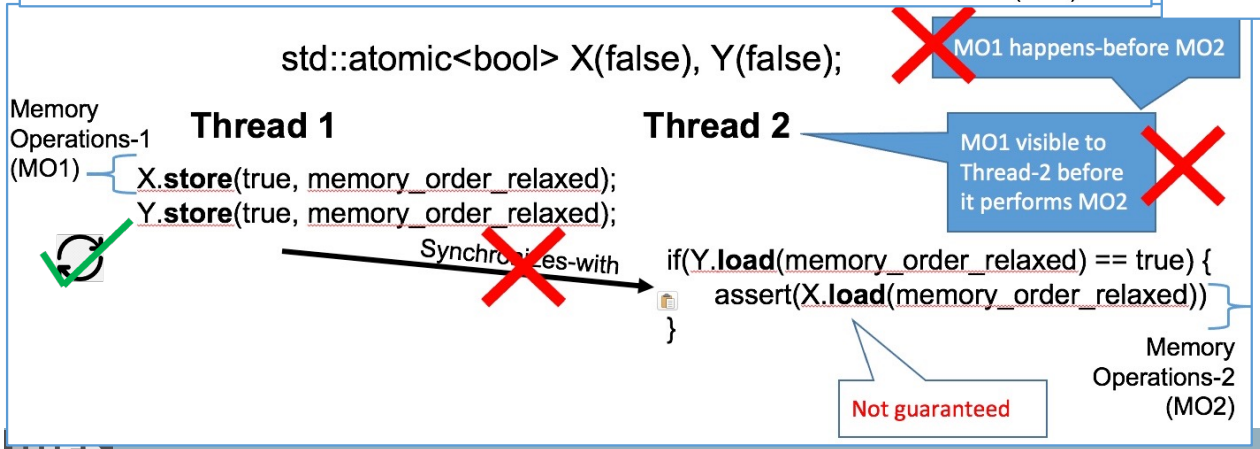
vivekk@iiitd.ac.in









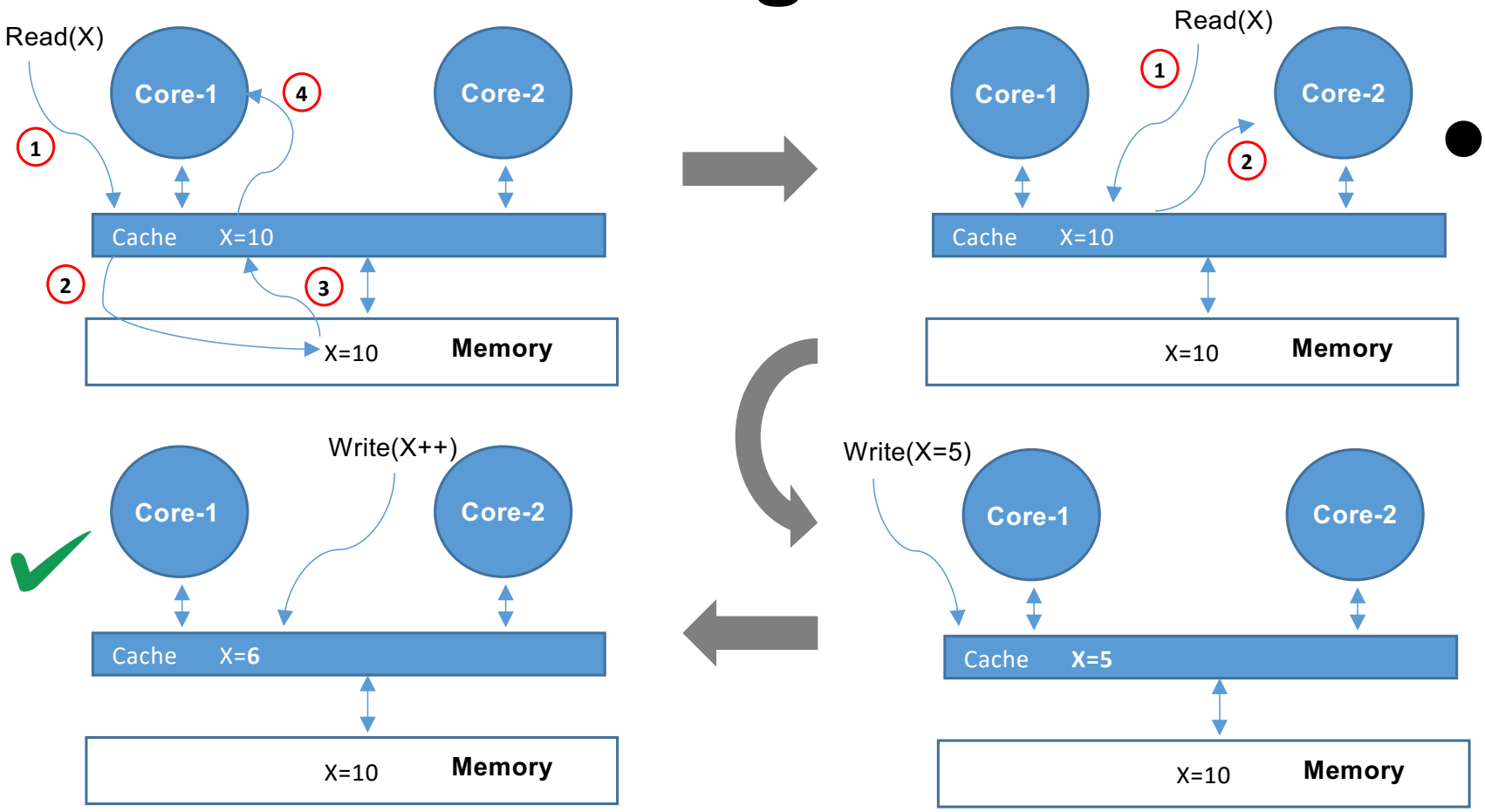


Last Lecture (Recap)

Today's Class

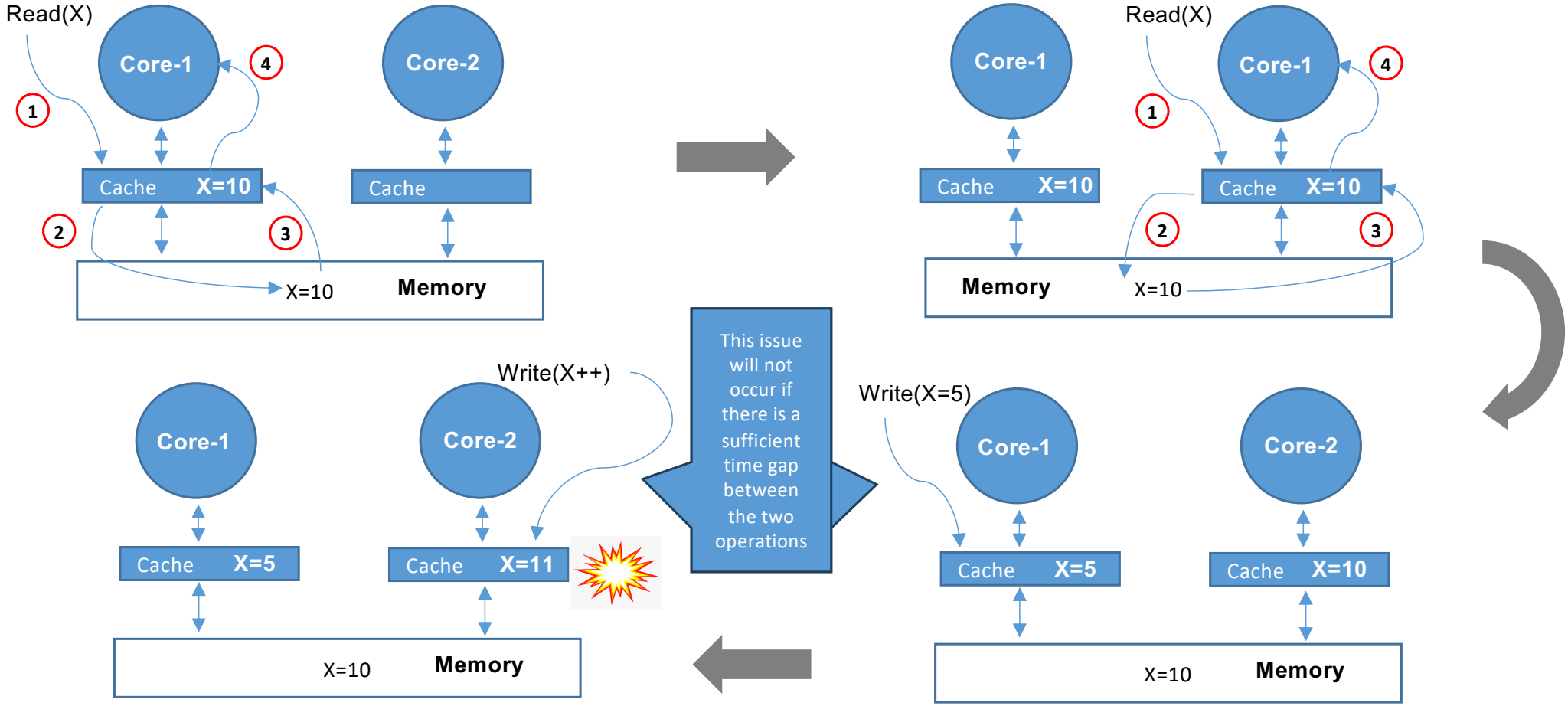
- Cache coherency
  - MSI protocol
  - MESI protocol
- False sharing
- Writing cache friendly code

Coherence using Shared Cache Only



- While it is easy to implement, it would be very costly and inefficient

Private Caches: Cache Coherence Problem



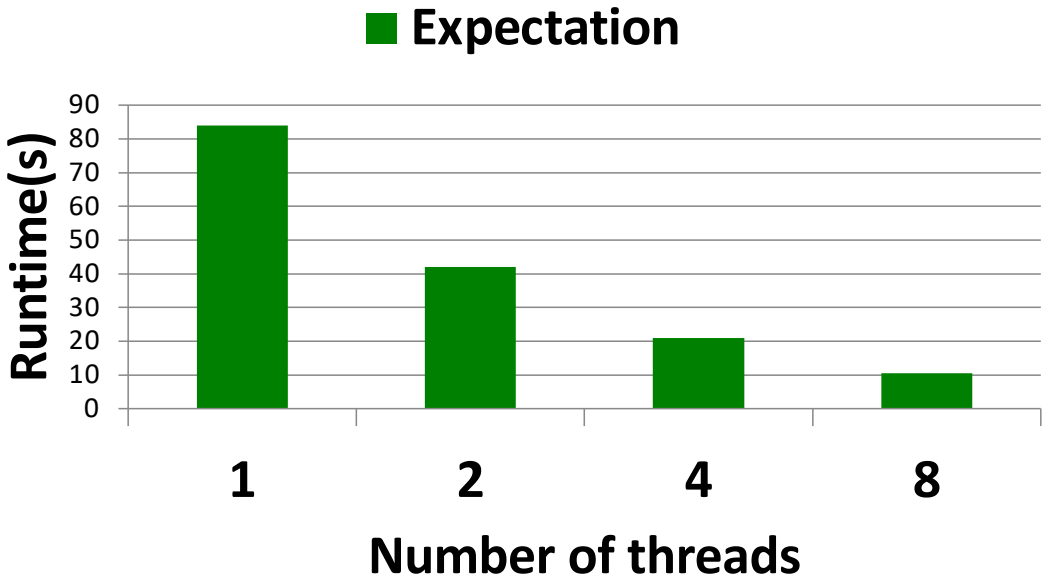
Defining Cache Coherence

- **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

Parallel Updates

```
int count[8]; //Global array

thread_func(int id) {
    for(i = 0; i < M; i++)
        count[id]++;
}
```

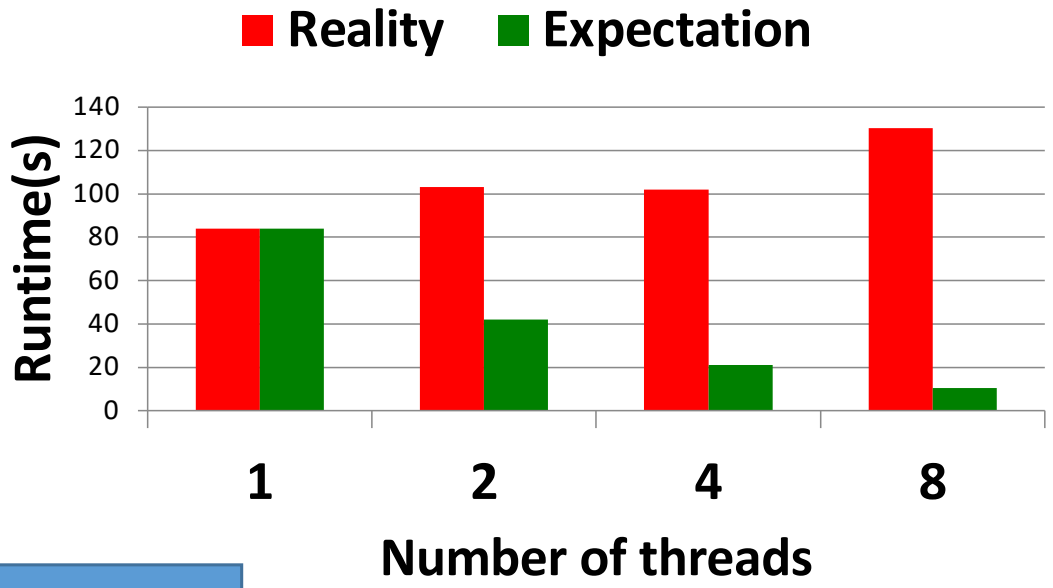


Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

- https://www.youtube.com/watch?v=NJ46OXN45eU

Reality v/s Expectation

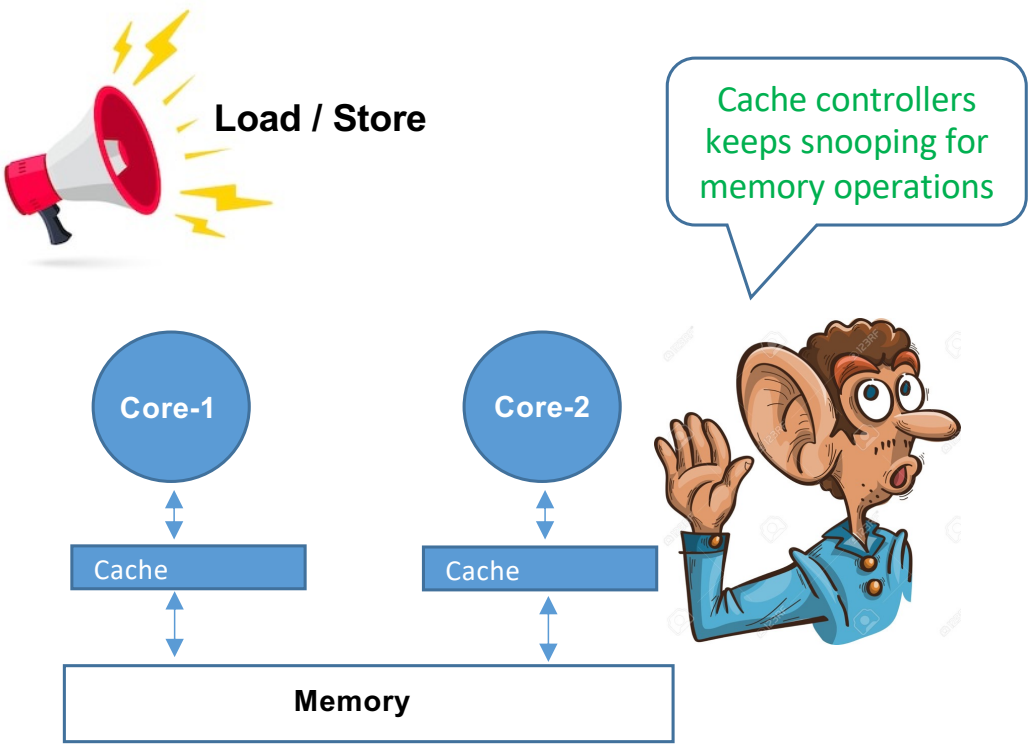
```
int count[8]; //Global array

thread_func(int id) {
    for(i = 0; i < M; i++)
        count[id]++;
}
```



Let's try to understand the problem in this code using the coherence protocol

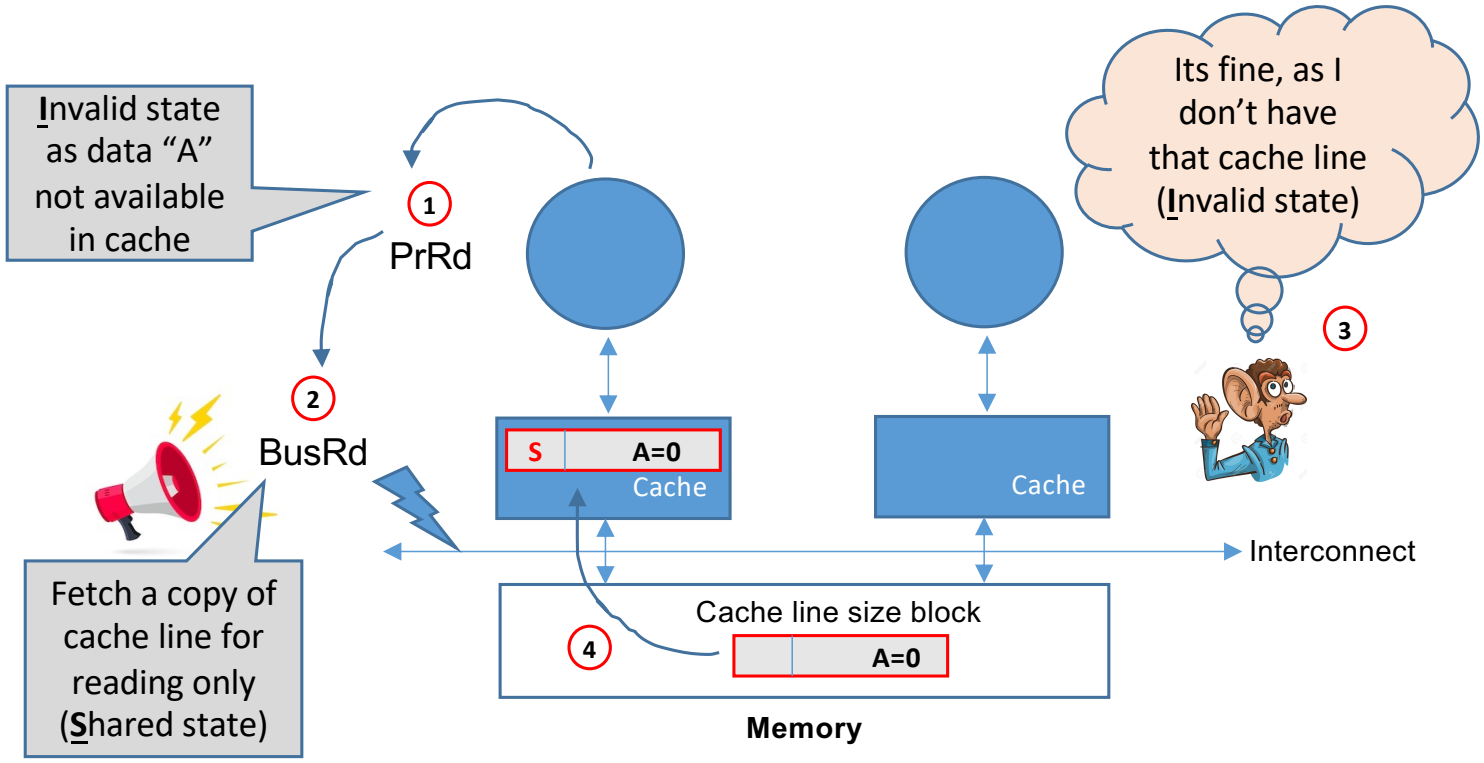
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Coherence using Private Caches



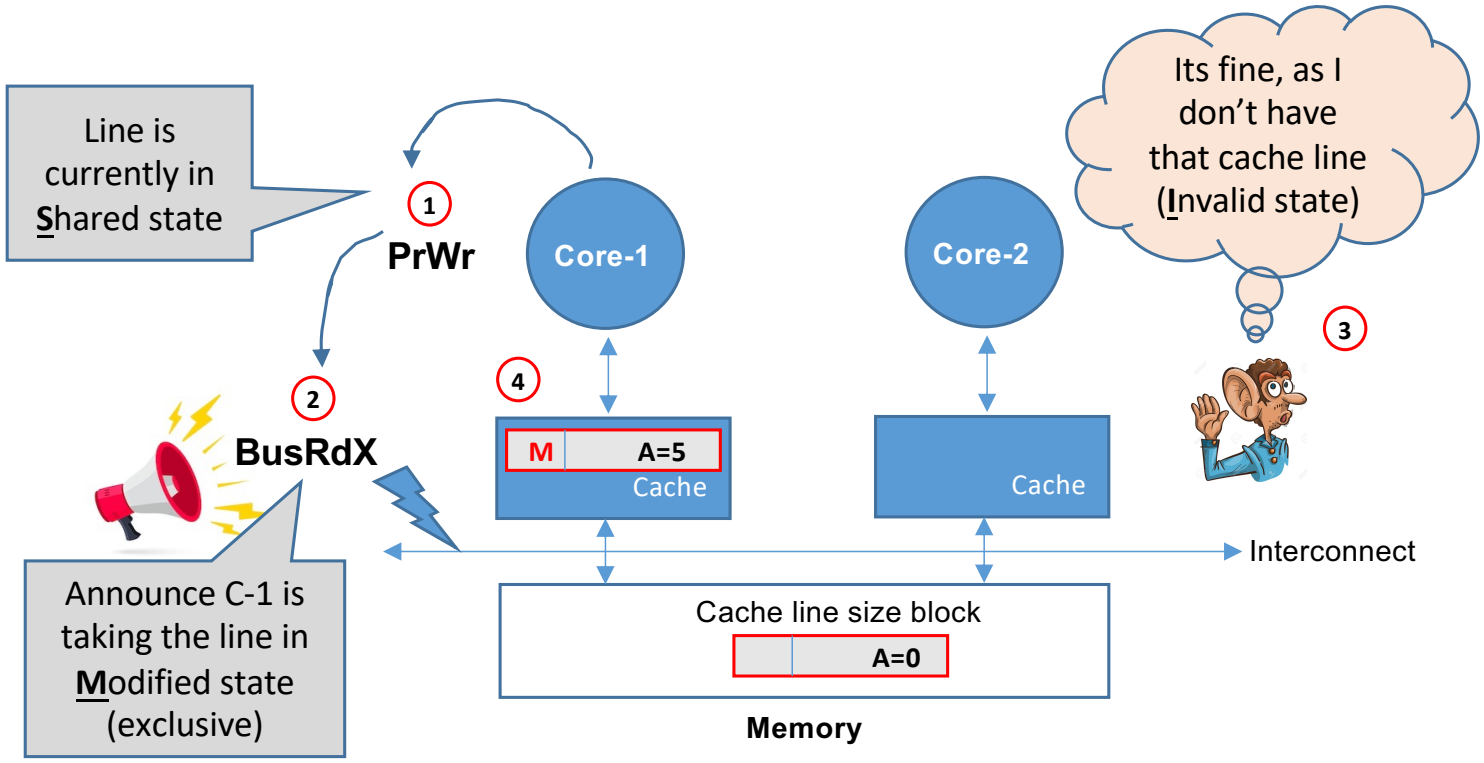
- Snooping based coherency protocol
  - Each core's cache controller broadcast any memory operations it wishes to perform before actually carrying out that operation
  - Rest of the core's cache controllers having that memory acts like a good citizen and help others to carry out their intended operation

MSI Protocol (1/5)



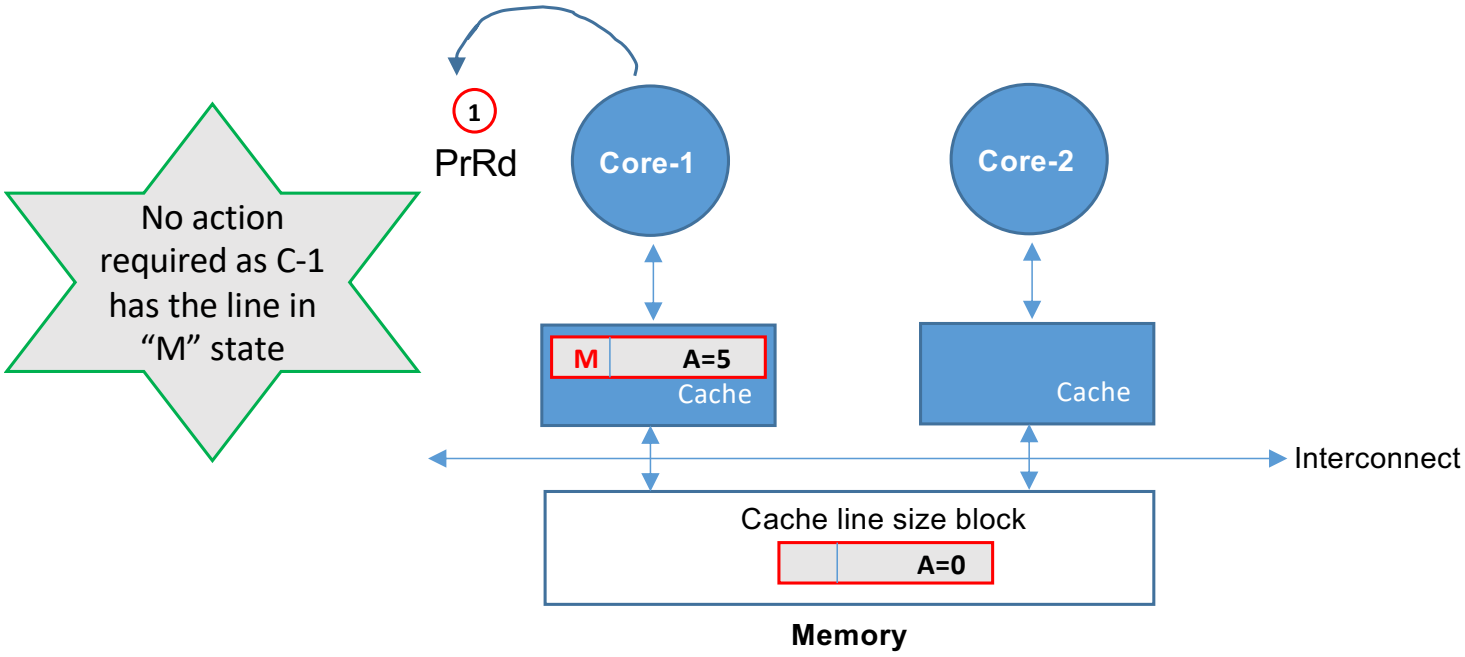
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode

MSI Protocol (2/5)



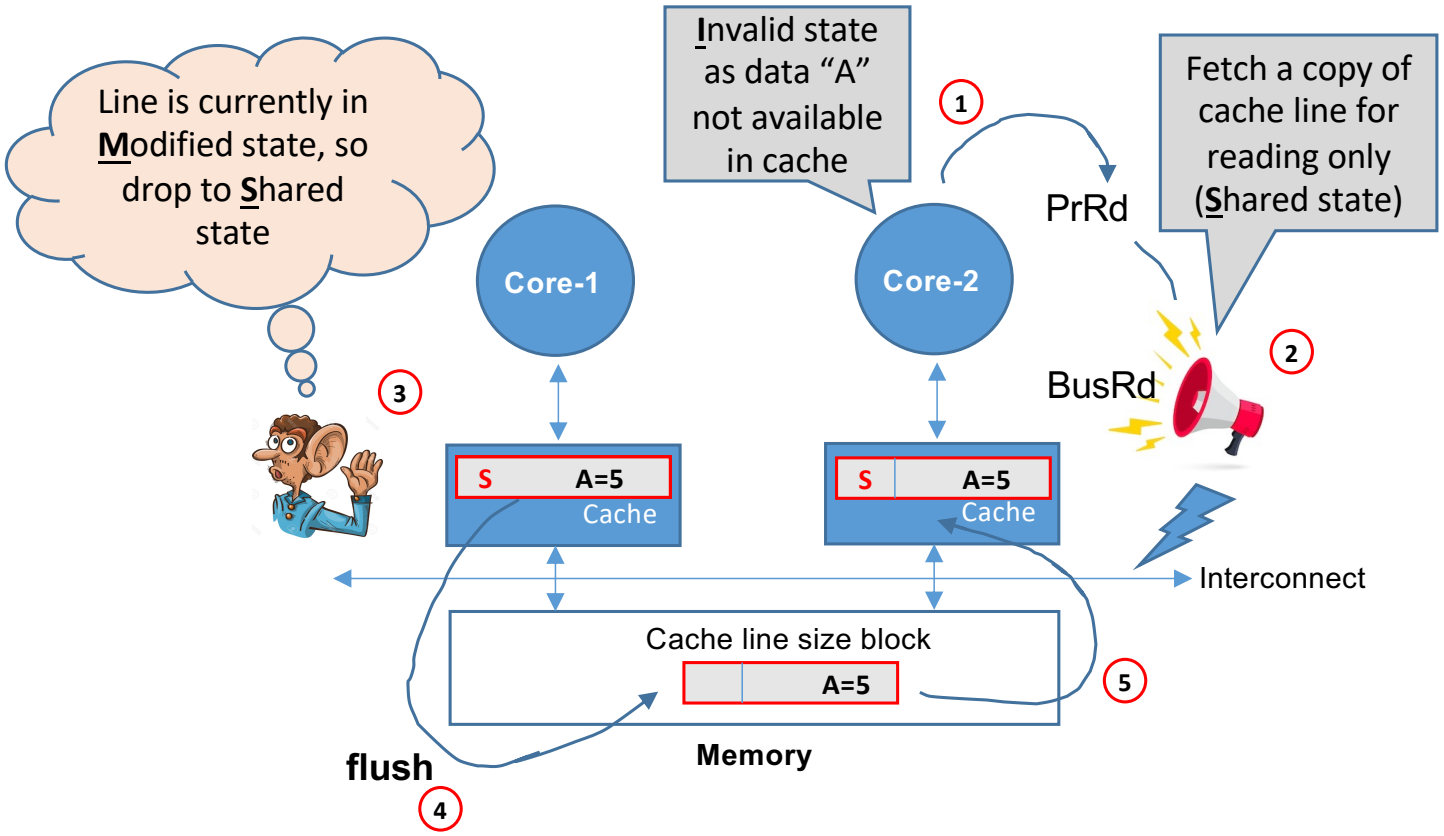
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

MSI Protocol (3/5)



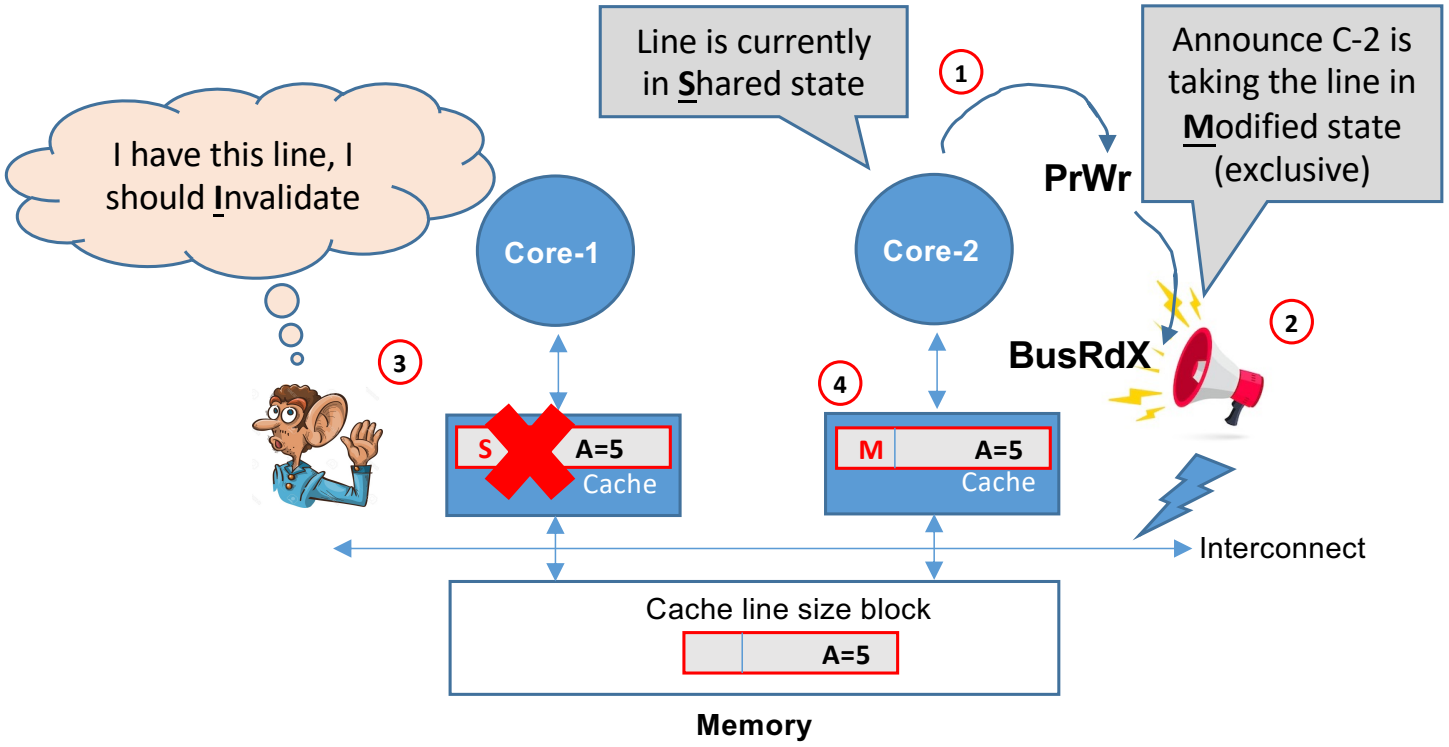
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

MSI Protocol (4/5)



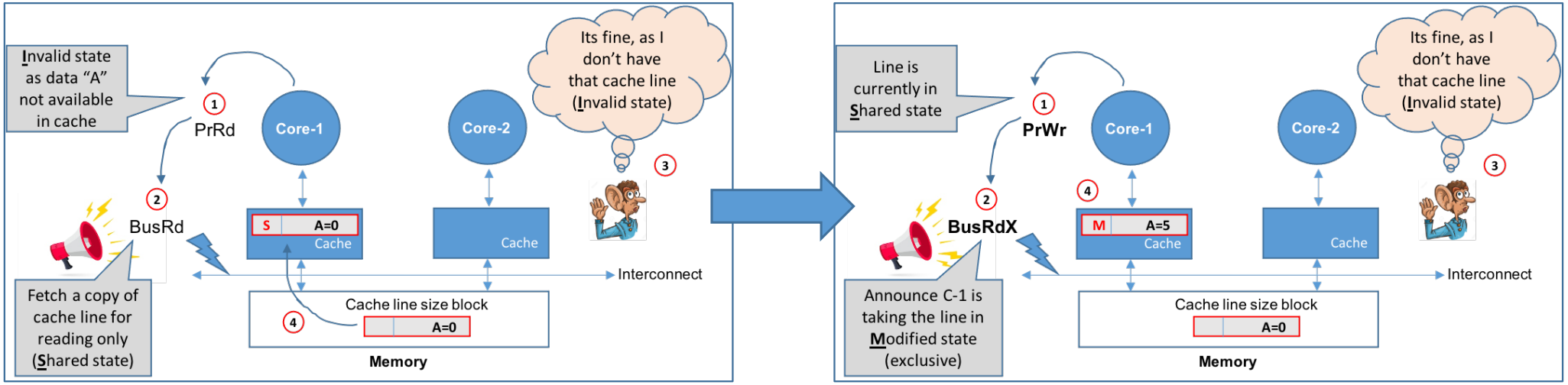
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

MSI Protocol (5/5)



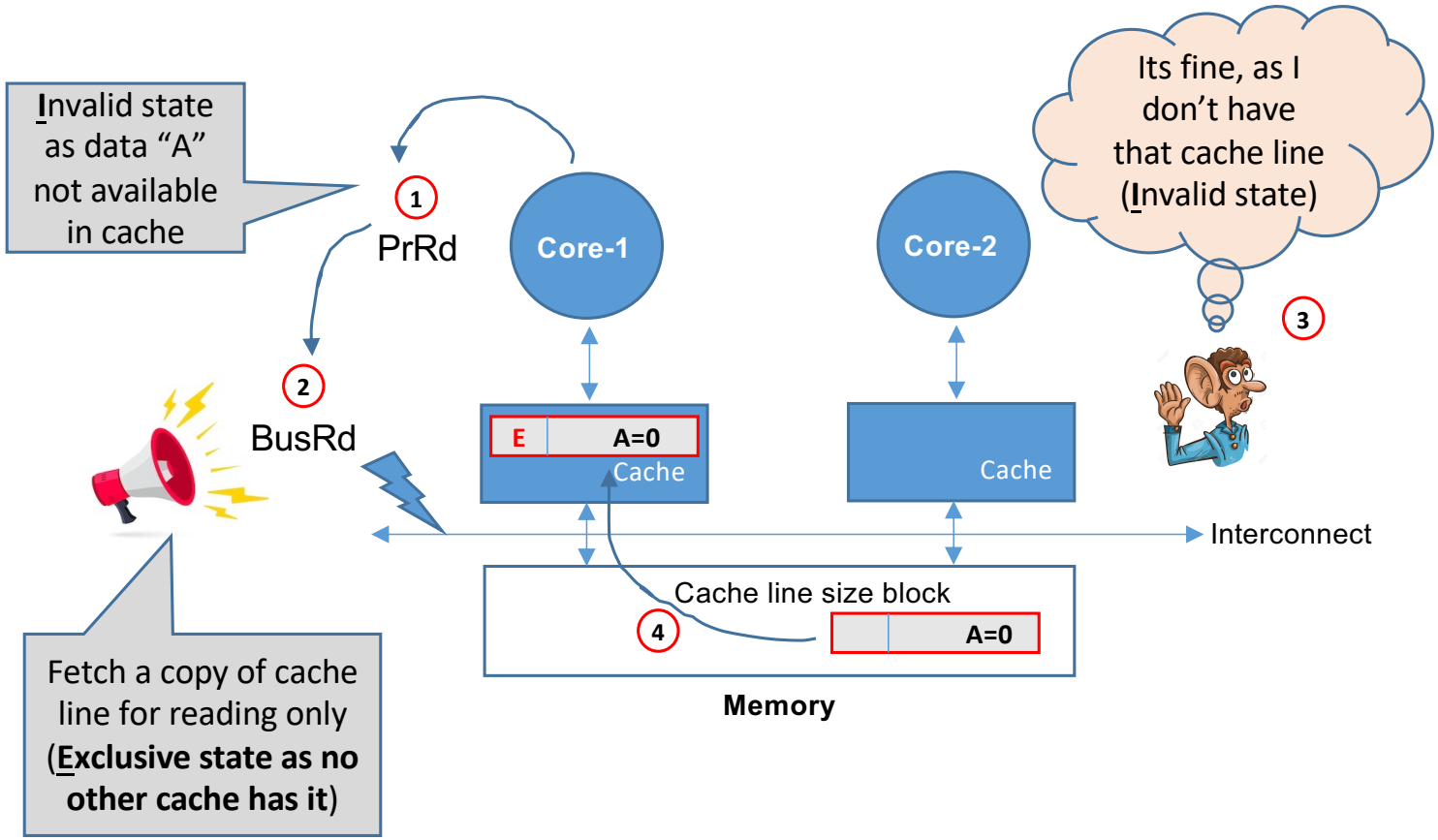
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

What is Wrong with MSI?



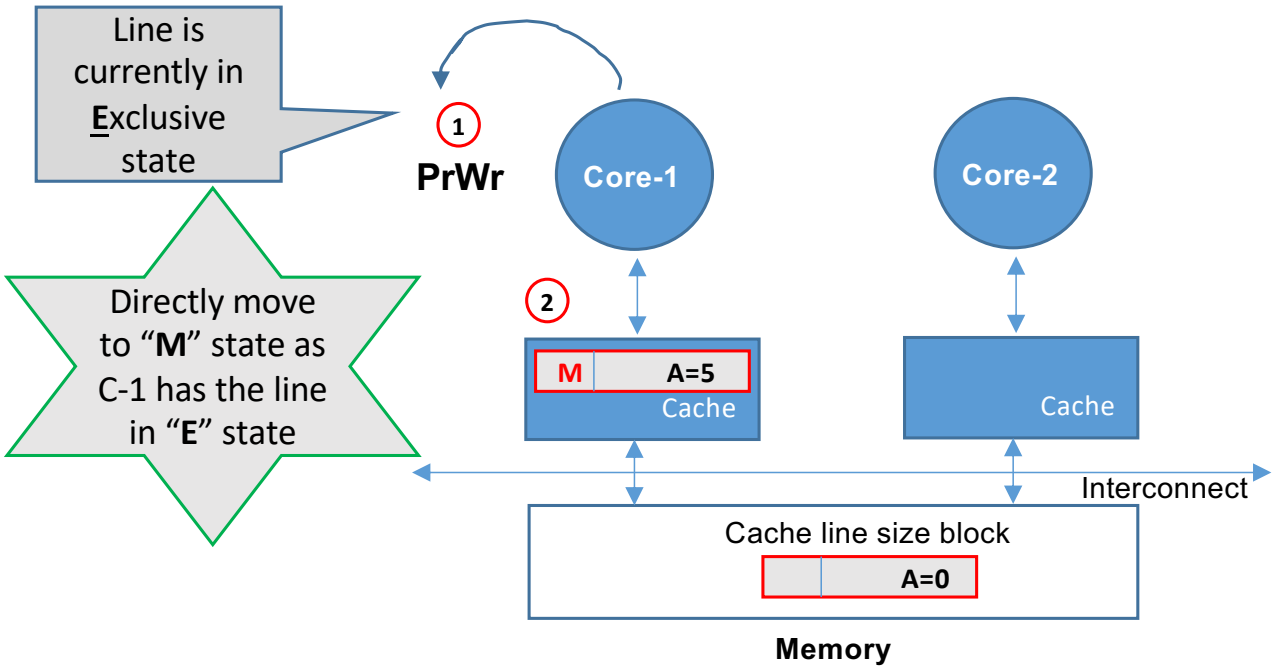
- Core-1 reads a data, and then wishes to modify it
  - The line is only in Core-1's, but it's cache controller still has to perform BusRdX operation for moving the line from "S" state to "M" state
    - Redundant traffic over the interconnect

MESI Protocol (1/4)



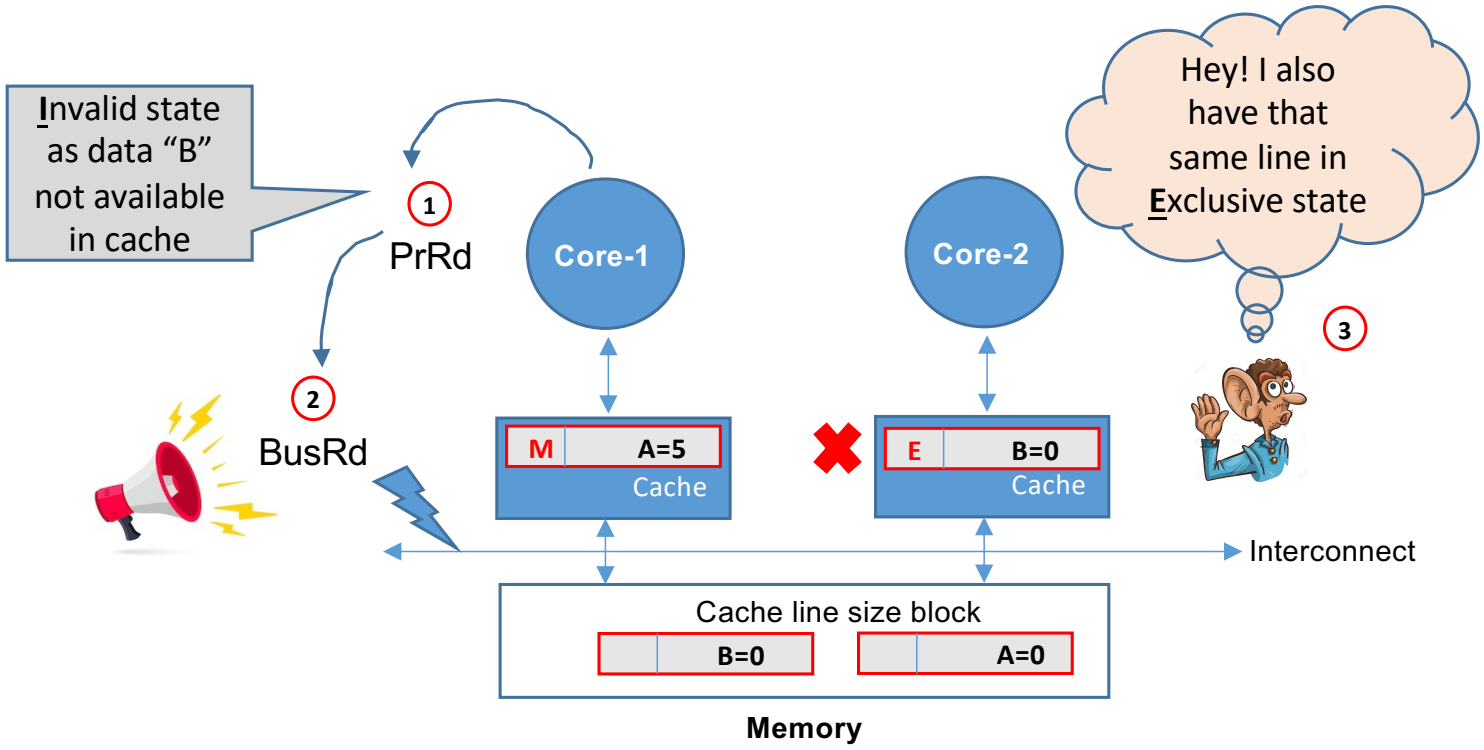
- An additional state **E**
  - Exclusive clean
  - Implies no other cache has a copy of this line

MESI Protocol (2/4)



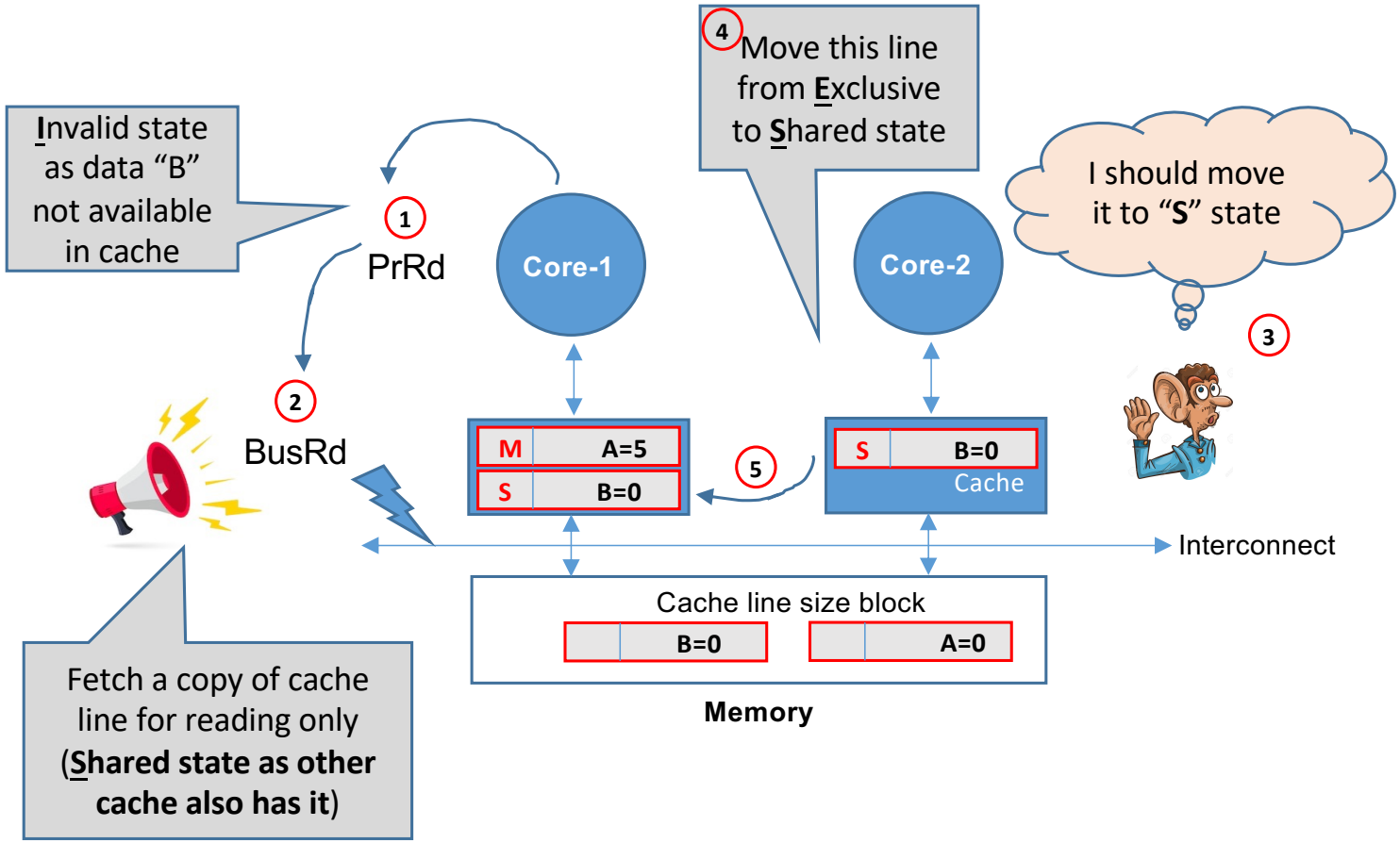
- Moving from **E** to **M** state
  - No action required to be performed on interconnect
  - Present **E** state implies the line is not in any other cache

MESI Protocol (3/4)



- 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 cannot retain it in **E** state anymore

MESI Protocol (4/4)



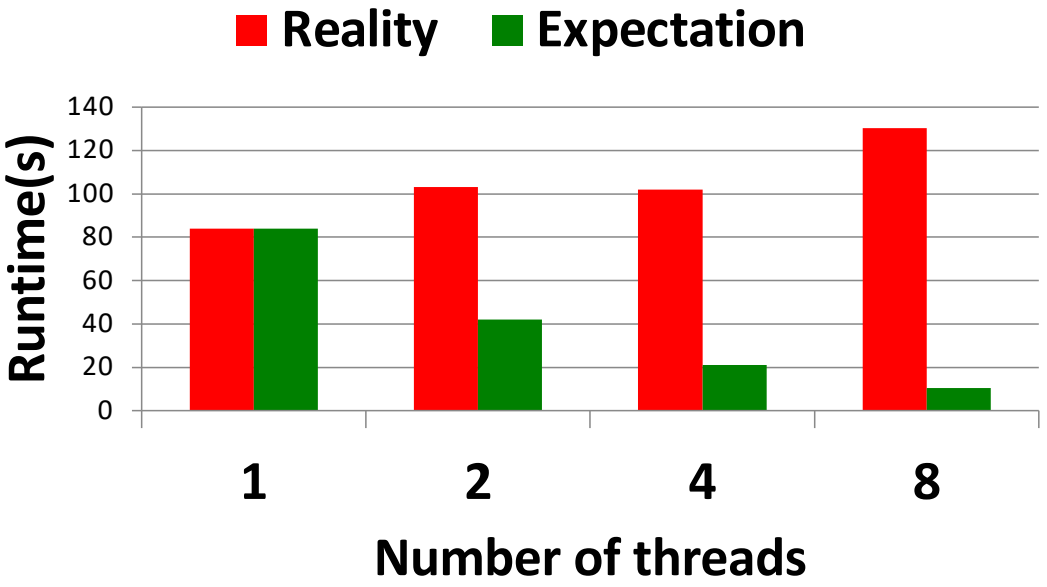
- C-1 wants to read a line (B) which C-2 also has (E state)
  - C-1 cannot have it in **E** state, as C-2 also wants to own it for read purpose
  - C-2 drops it from E to S state
  - C-1 has the line in S state

Today's Class

- Cache coherency
  - MSI protocol
  - MESI protocol
- False sharing
- Writing cache friendly code

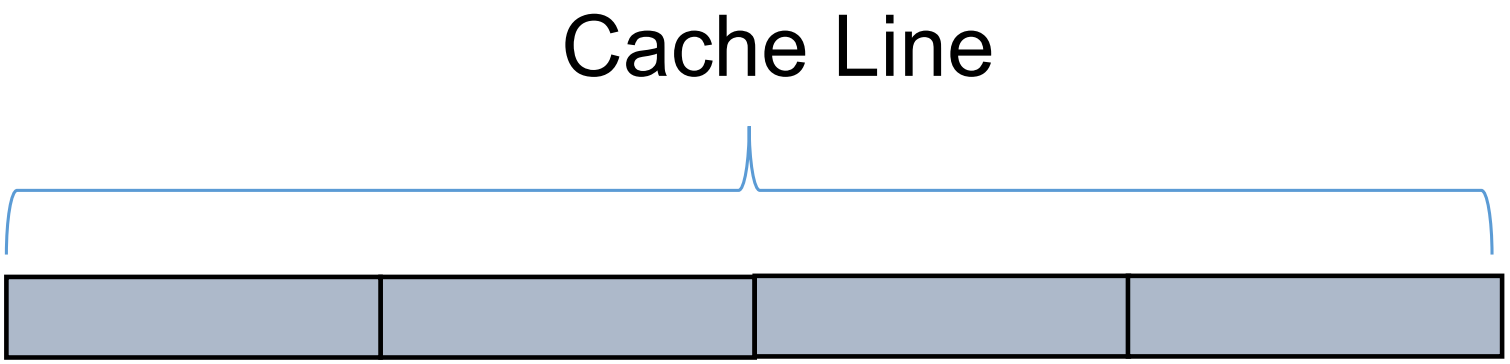
False Sharing

```
int count[8]; //Global array

thread_func(int id) {
    for(i = 0; i < M; i++)
        count[id]++;
}
```



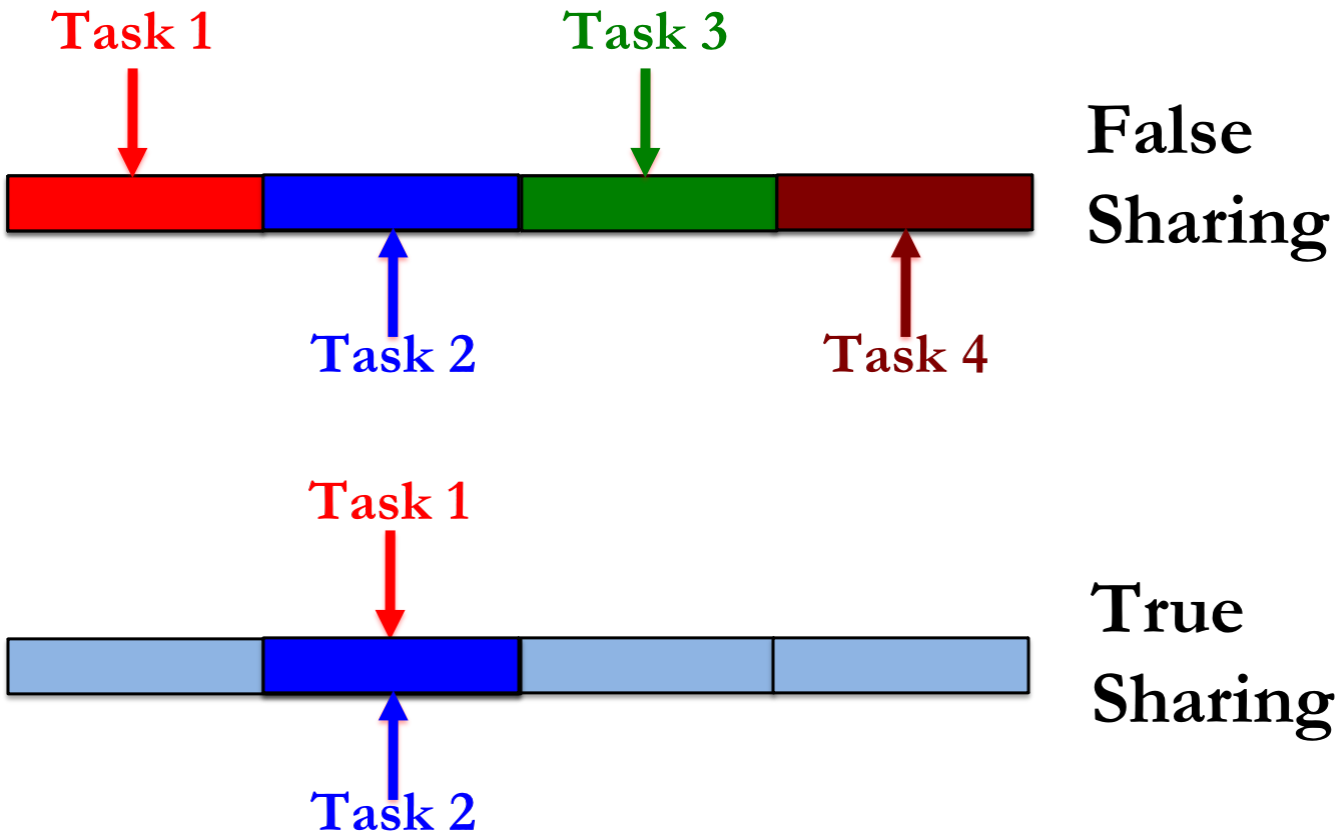
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing vs. True Sharing



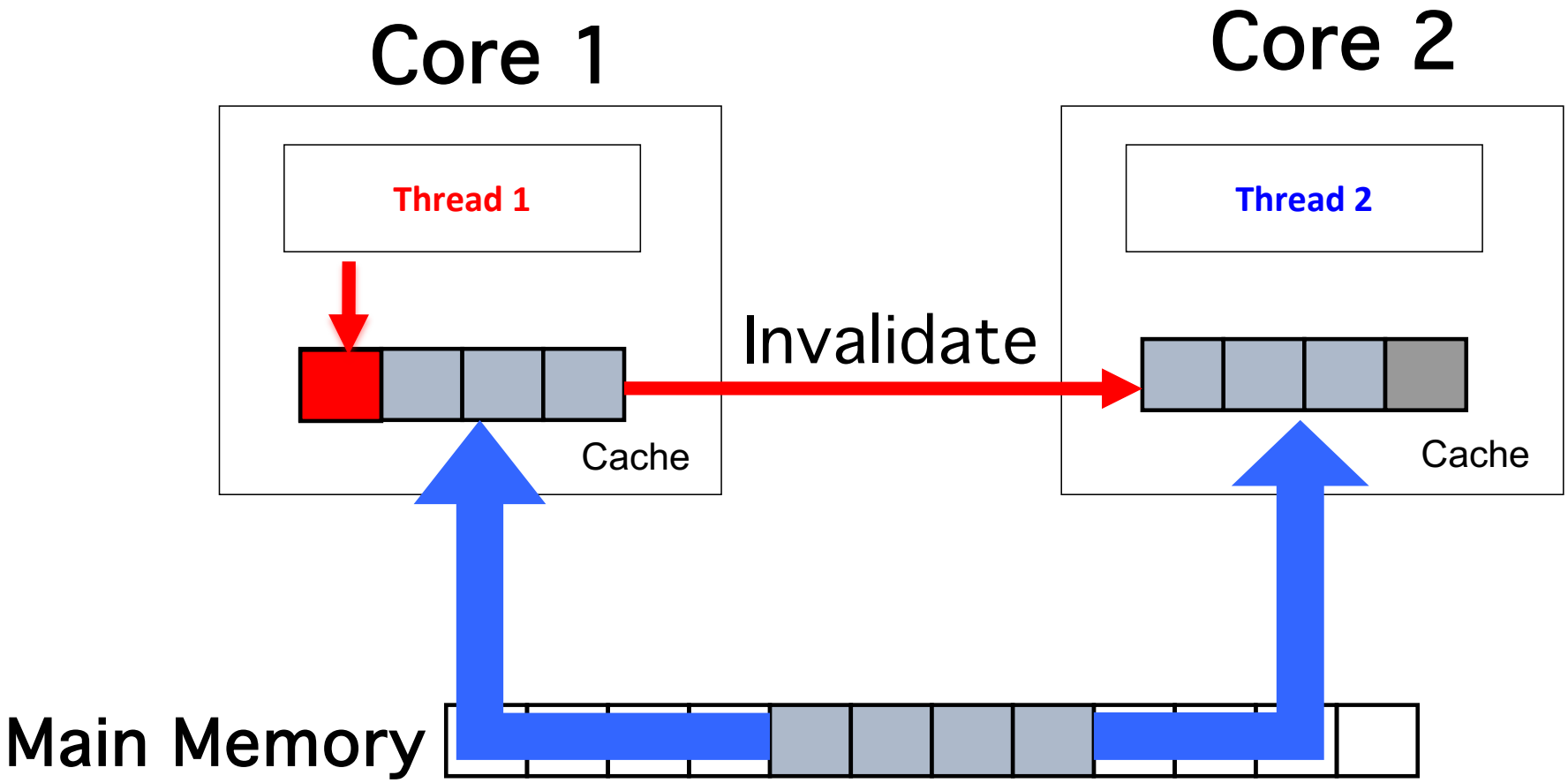
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing vs. True Sharing



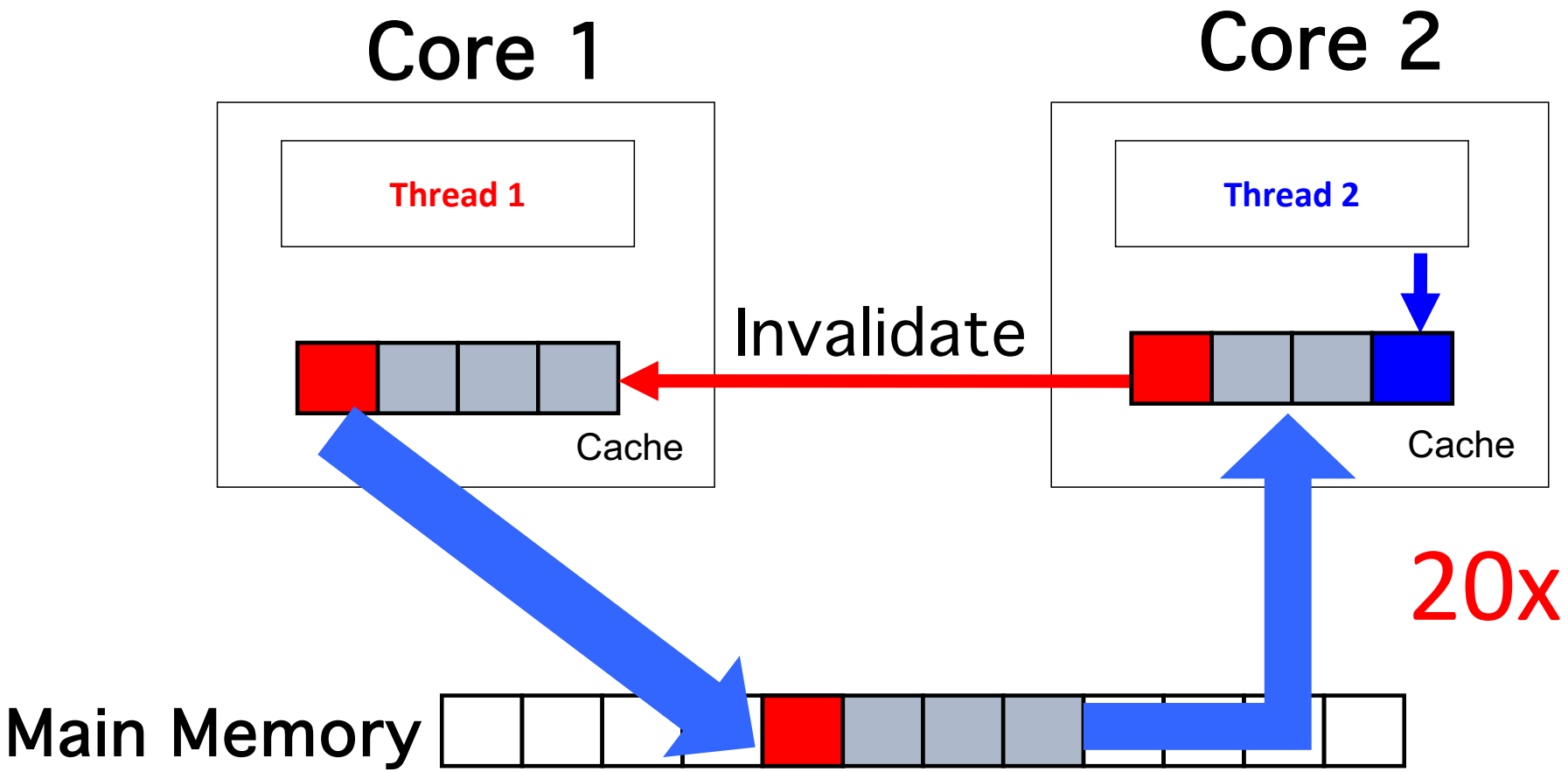
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing



Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing



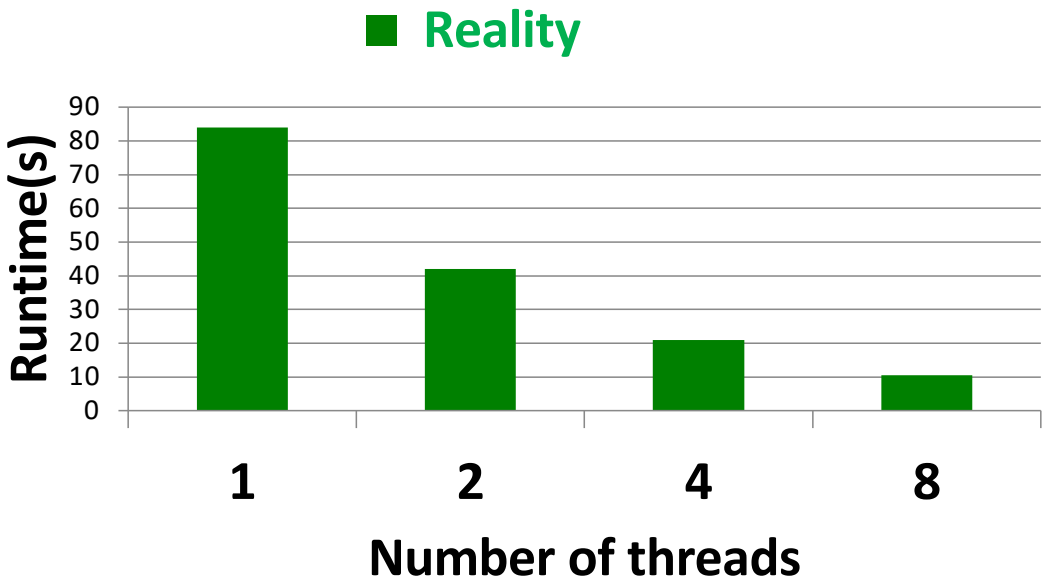
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Resource Contention at Cache Line Level

Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Parallel Updates – How to Fix it Manually?

```
int count[8]; //Global array

thread_func(int id) {
    for(i = 0; i < M; i++)
        count[id]++;
}
```



Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing is Everywhere

```
me = 1;
you = 1; // globals
```

```
me = new Foo;
you = new Bar; // heap
```

```
class X {
  int me;
  int you;
}; // fields
```

```
arr[me] = 12;
arr[you] = 13; // array indices
```

Two different threads
T1 & T2 are involved

Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Detecting False Sharing

- False sharing on a cache line implies that particular cache line will incur large number of cache invalidations
- A runtime approach can track memory access using hardware performance counters

Automatically Fixing the False Sharing

- Core idea
  - Identify and fix memory locations that could lead to false sharing
  - Approaches
    - Using compilation technique
      - Use static analysis of the application and identify the locations of potential false sharing
      - Can emit memory padding to avoid false sharing
      - Limited to eliminating false sharing for programs where the size and location of data elements can be determined statically
    - Use runtime technique
      - Overcomes the above limitations
      - Runtime checks might inflate the execution time
      - Let us try to understand one of the well-known runtime implementations

        *(**Sheriff**: https://github.com/plasma-umass/sheriff/tree/master)*

Detour – Process Creation

```
int read_var[1024];   //page aligned (4Kb size)
int write_var[1024];  //page aligned (4Kb size)

int main() {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
        write_var[0] = 2 * read_var[0];
    } else {
        printf("I am the parent Shell\n");
        write_var[0] = 4 * read_var[0];
    }
    printf("write_var value = %d\n", write_var[0]);
    ....
    return 0;
}
```

- fork is a system call used for creating a new process
- Called once, but returns twice!
  - Return value in child process is zero, whereas child's process PID is returned in parent process
- It creates a replica of the parent process
  - Copy-on-Write (COW) – Initially, both parent and child process have read-only access to parent's address space. Whichever process attempts a write on a memory page in parent's address space, it would get a copy of that page (lazy copy)

What could be our Approach?

```
1.  /* global variables */
2.  int sum[2];
3.  int main() {
4.    /* heap allocations */
5.    int a = new int[size]; //initialized
6.    T1 = new thread([=]() {
7.      for(int i=0; i<size/2; i++) sum[0]+=a[i];
8.    });
9.    T2 = new thread([=]() {
10.     for(int i=size/2; i<size; i++) sum[1]+=a[i];
11.   });
12.   T1.join(); T2.join();
13.   //Cleanups
14. }
```

False sharing

False sharing

Walkthrough of Sheriff Execution

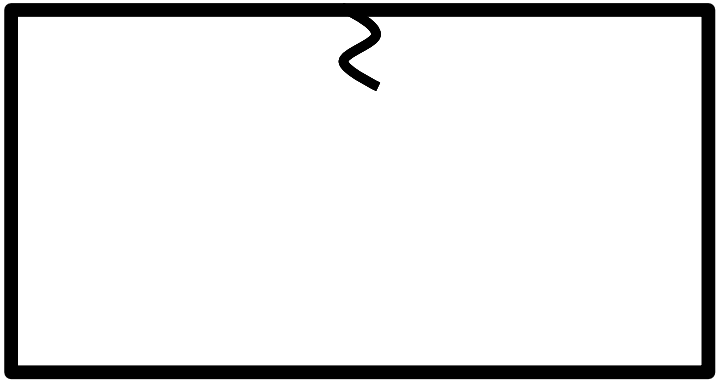
Process creation – creating processes instead of threads

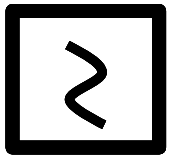
```
1.  /* global variables */
2.  int sum[2];
3.  int main() {
4.    /* heap allocations */
5.    int a = new int[size]; //initialized
6.    T1 = new thread([=]() {
7.      for(int i=0; i<size/2; i++) sum[0]+=a[i];
8.    });
9.    T2 = new thread([=]() {
10.     for(int i=size/2; i<size; i++) sum[1]+=a[i];
11.   });
12.   T1.join(); T2.join();
13.   //Cleanups
14. }
```

Sheriff Execution: Process Creation

shared address space

disjoint address spaces





- In Linux, both pthreads and processes are essentially a KLT, and are created using the same API (do_fork)

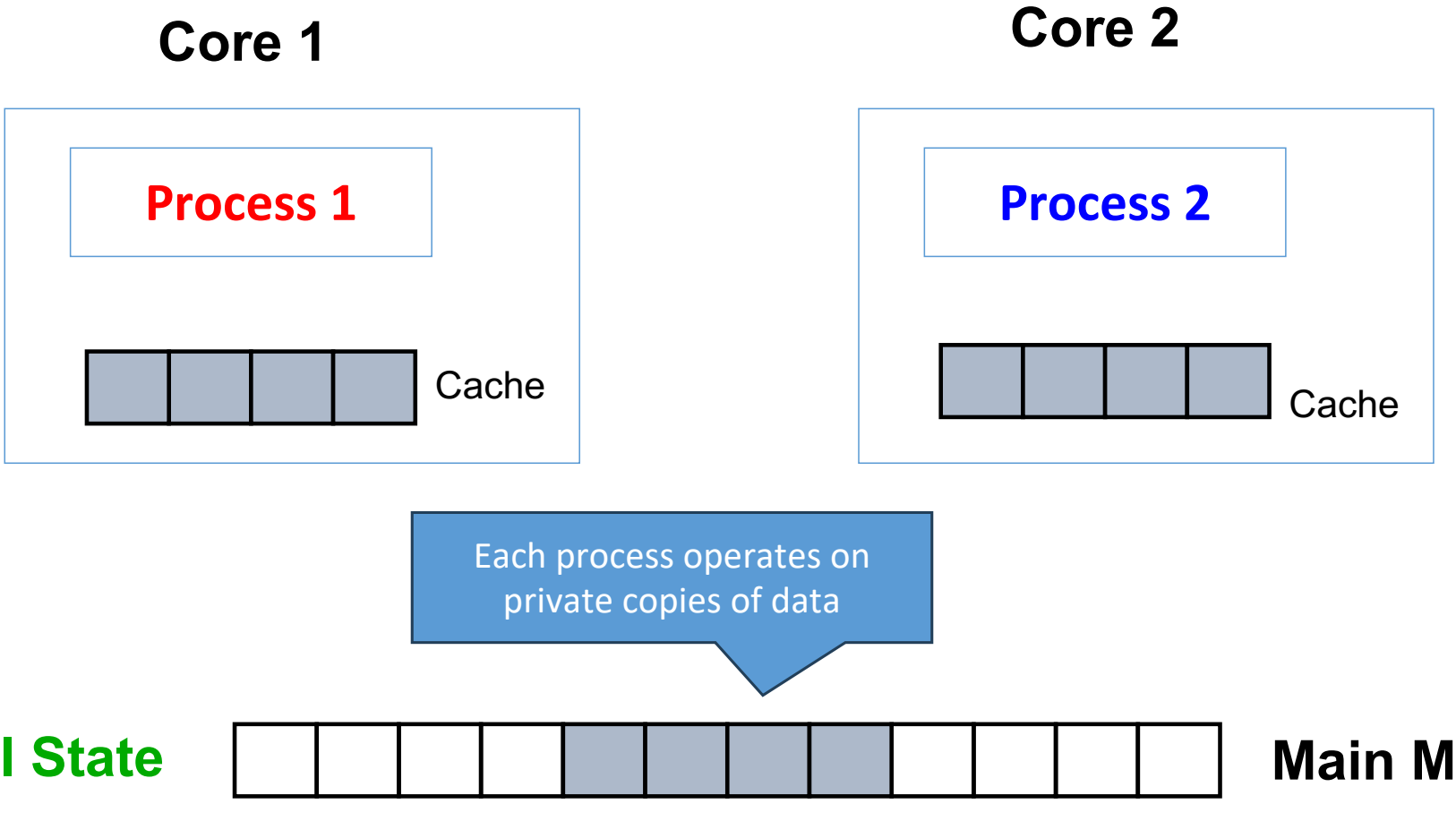
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Walkthrough of Sheriff Execution

Initialization – creating mapping of global and heap variables for processes

```
1.  /* global variables */
2.  int sum[2];
3.  int main() {
4.    /* heap allocations */
5.    int a = new int[size]; //initialized
6.    T1 = new thread([=]() {
7.      for(int i=0; i<size/2; i++) sum[0]+=a[i];
8.    });
9.    T2 = new thread([=]() {
10.     for(int i=size/2; i<size; i++) sum[1]+=a[i];
11.   });
12.   T1.join(); T2.join();
13.   //Cleanups
14. }
```

Sheriff Execution: Initialization



Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Sheriff Execution: Initialization

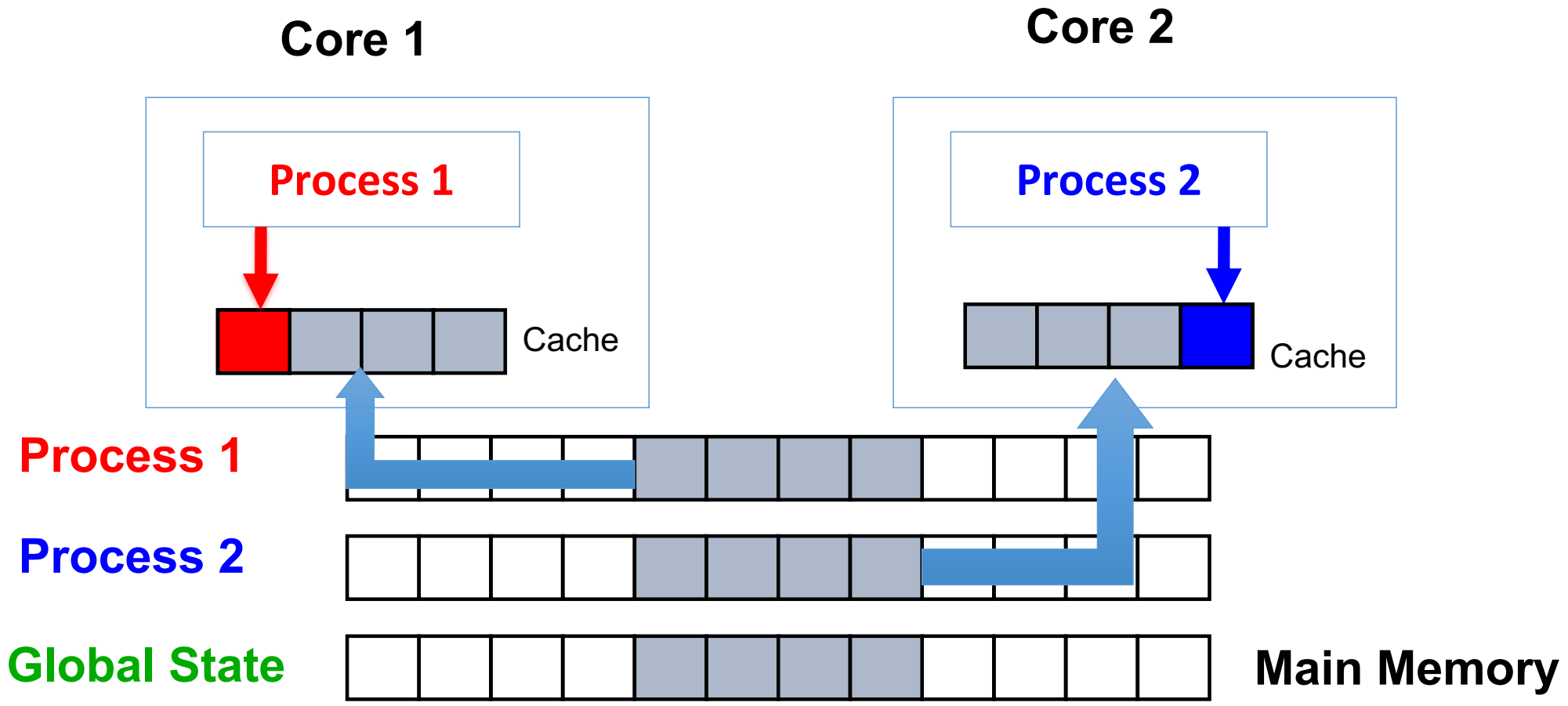
- Advantages of converting threads into processes
  - Enables the use of per-thread page protection, allowing Sheriff to track memory accesses by different threads (processes)
  - Each thread's (process) memory access are isolated, hence they would not update the same cache line
    - No false sharing!
- Memory mapped files are used to share global and heaps across different processes
- Twin copies of the pages for storing the global and heaps
  - Shared mapping for holding shared states
    - Pages storing these shared states are marked copy-on-write
  - Private mapping for per-process updates
    - Private copy of of the above shared pages are created whenever a process would attempt to update a page for the first time

Walkthrough of Sheriff Execution

Execution – copies of memory pages per process for local updates

```
1.  /* global variables */
2.  int sum[2];
3.  int main() {
4.    /* heap allocations */
5.    int a = new int[size]; //initialized
6.    T1 = new thread([=]() {
7.      for(int i=0; i<size/2; i++) sum[0]+=a[i];
8.    });
9.    T2 = new thread([=]() {
10.     for(int i=size/2; i<size; i++) sum[1]+=a[i];
11.   });
12.   T1.join(); T2.join();
13.   //Cleanups
14. }
```

Sheriff Execution: Execution



Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Walkthrough of Sheriff Execution

```
1.  /* global variables */
2.  int sum[2];
3.  int main() {
4.    /* heap allocations */
5.    int a = new int[size]; //initialized
6.    T1 = new thread([=]() {
7.      for(int i=0; i<size/2; i++) sum[0]+=a[i];
8.    });
9.    T2 = new thread([=]() {
10.     for(int i=size/2; i<size; i++) sum[1]+=a[i];
11.   });
12.   T1.join(); T2.join();
13.   //Cleanups
14. }
```

Synchronization – merging diffs in per-process pages into the global copy

Sheriff Execution: Synchronization

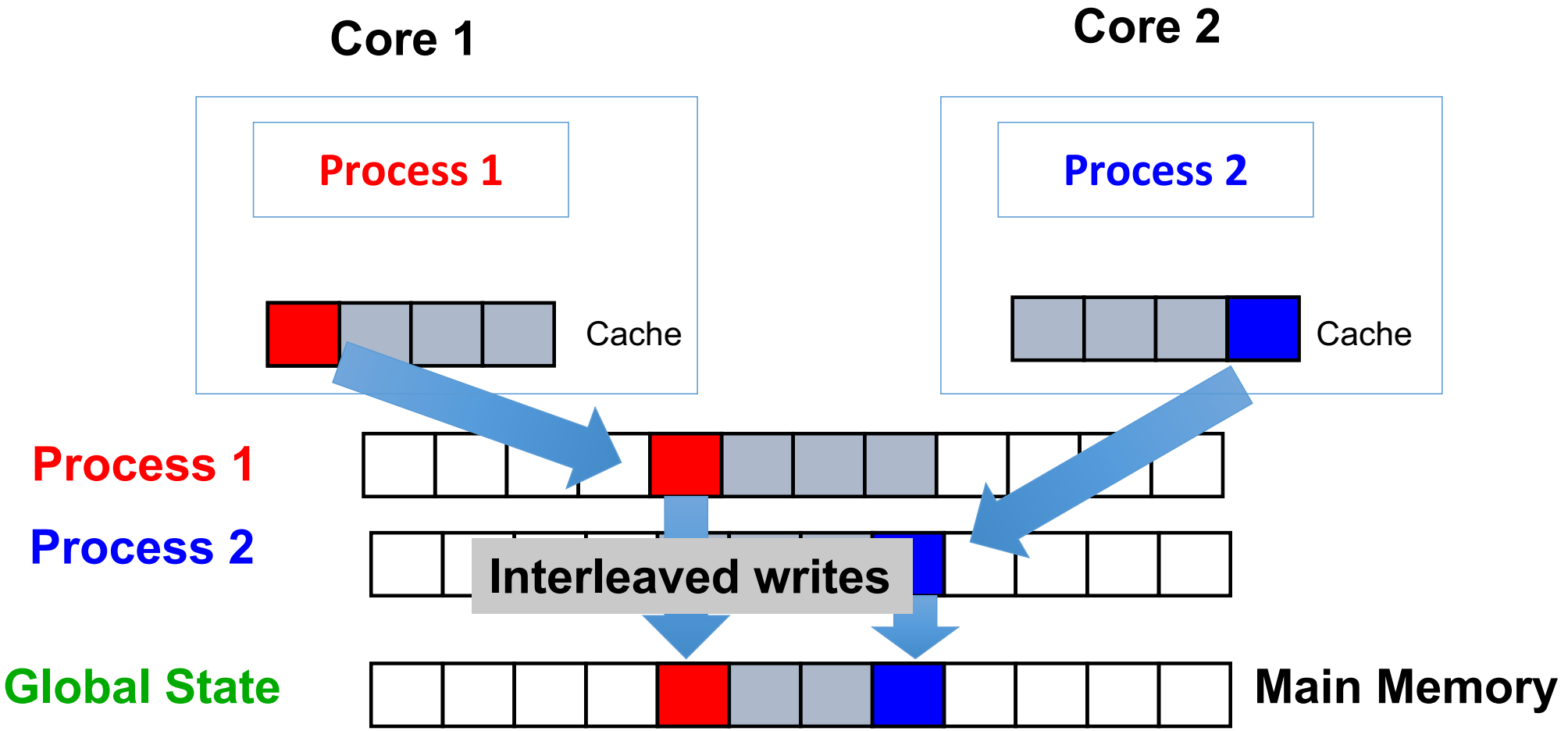
- There are two different types of synchronization points
  - Thread termination
  - End of the critical section (mutex unlock), barriers, etc.
- At each synchronization point, Sheriff commits changes from private pages to the shared pages
  - It commits only the differences between the twin and the modified pages

Sheriff Execution: Synchronization

Snapshot and diffing the local changes

Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Sheriff Execution: Synchronization



Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Sheriff Execution: Synchronization

Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Today's Class

- Cache coherency
  - MSI protocol
  - MESI protocol
- False sharing
- Writing cache friendly code

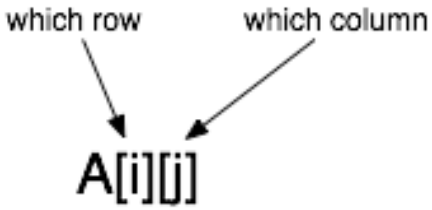
Writing Cache Friendly Code

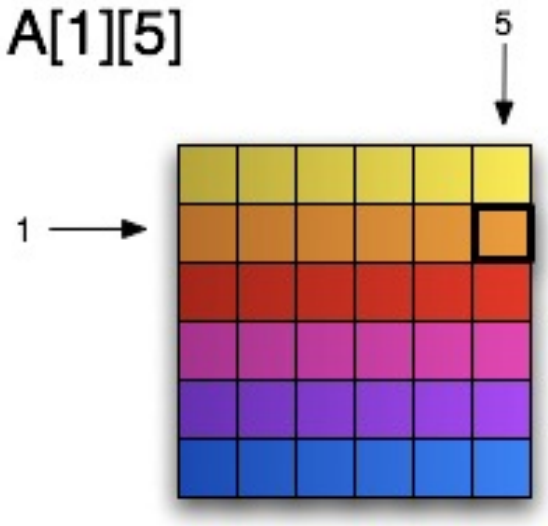
Two major rules:

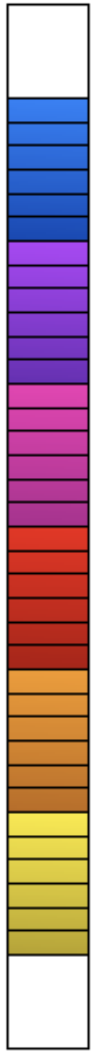
- Repeated references to data are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
  - Stride-1: i++
  - Stride-2: i+=2
  - ….

Layout of C Arrays in Memory

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







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%

Next Lecture

- Miscellaneous Topics in Parallel Programming
- Quiz-5
  - Syllabus: Lectures 19-24