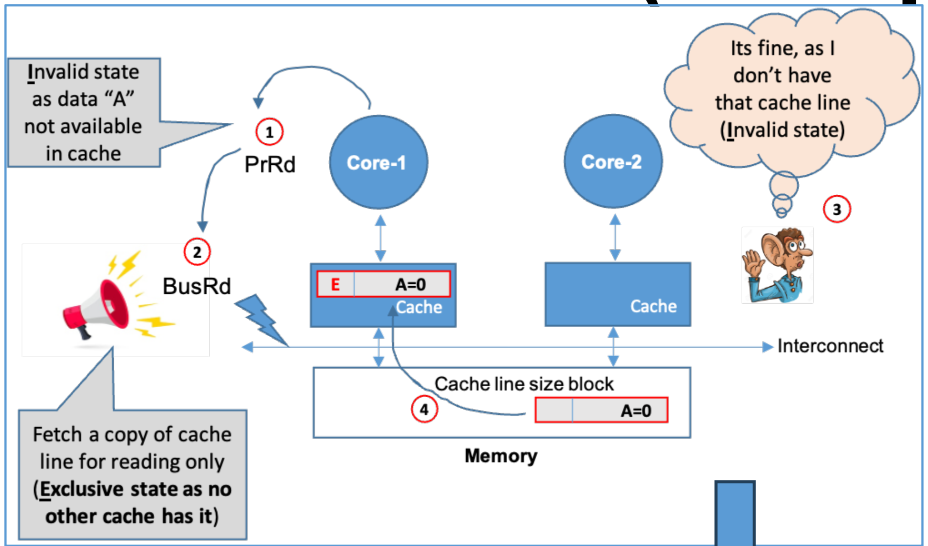
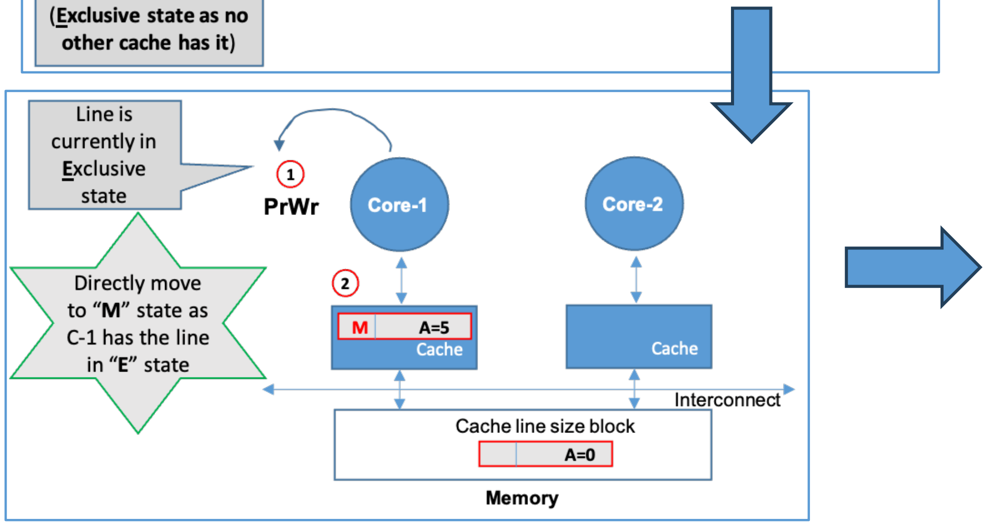
Lecture 17: False Sharing

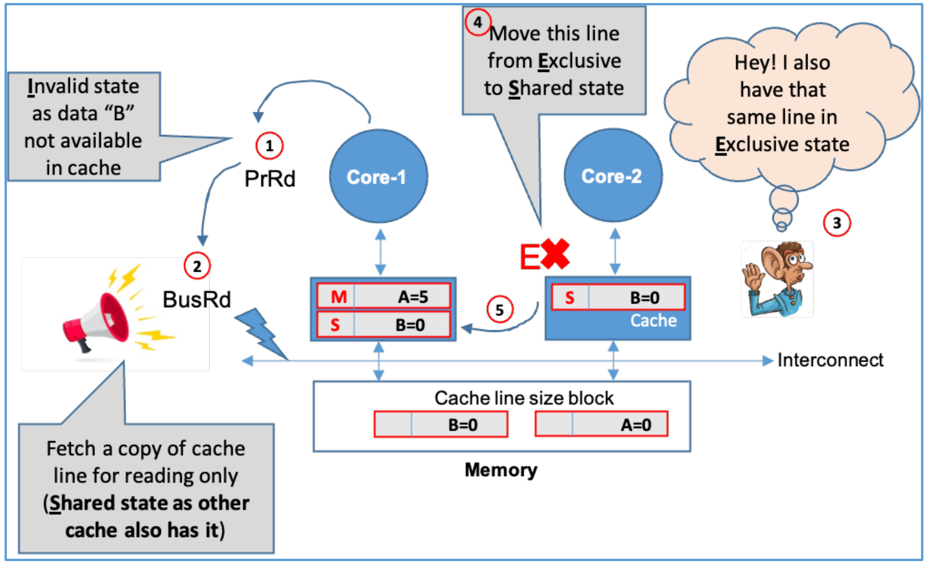
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture (Recap)

- MESI protocol based cache coherence for write-back private cache







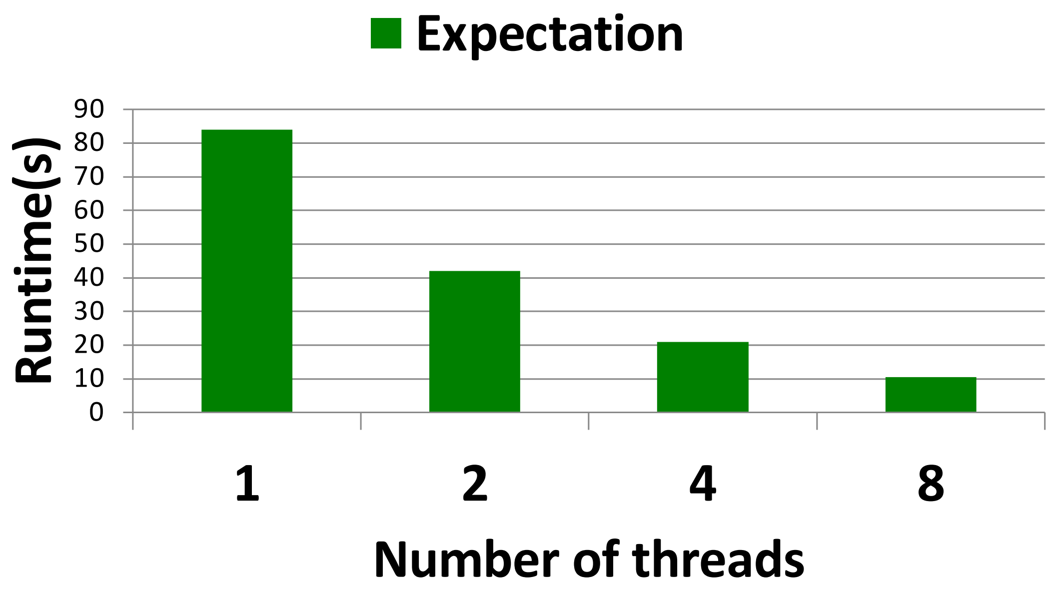
Today's Class

- False sharing
- Runtime solutions for detecting/repairing false sharing
  - Sheriff
  - Featherlight

Acknowledgement: Today's lectures slides are adapted from several conference presentation slides available online on false sharing

Parallel Updates

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

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

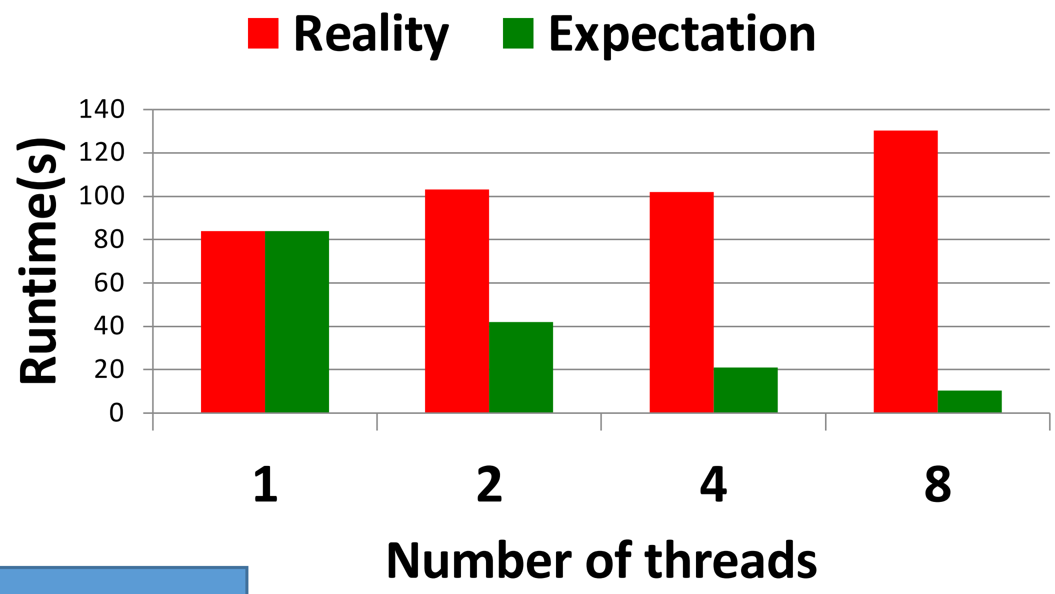


Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

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

False Sharing

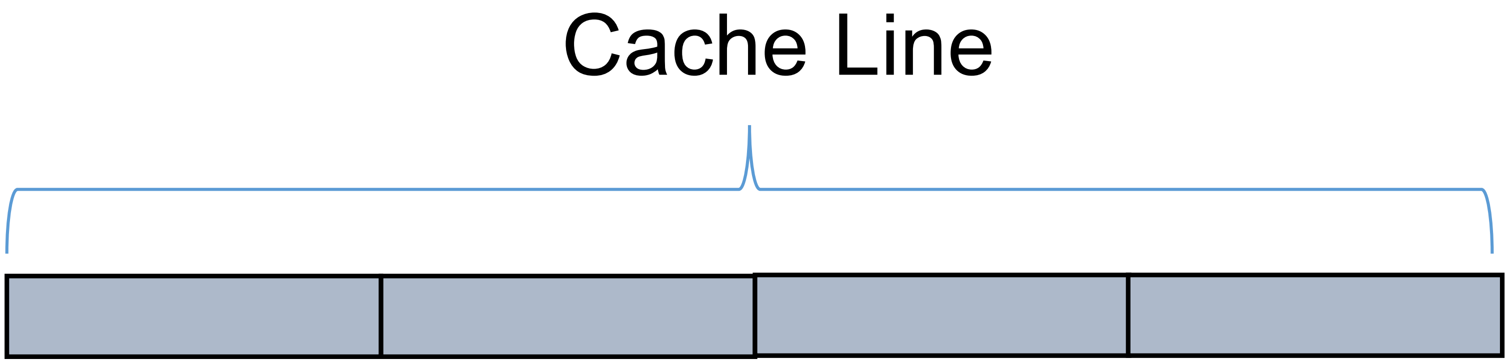
```
int count[8]; //Global array

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



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

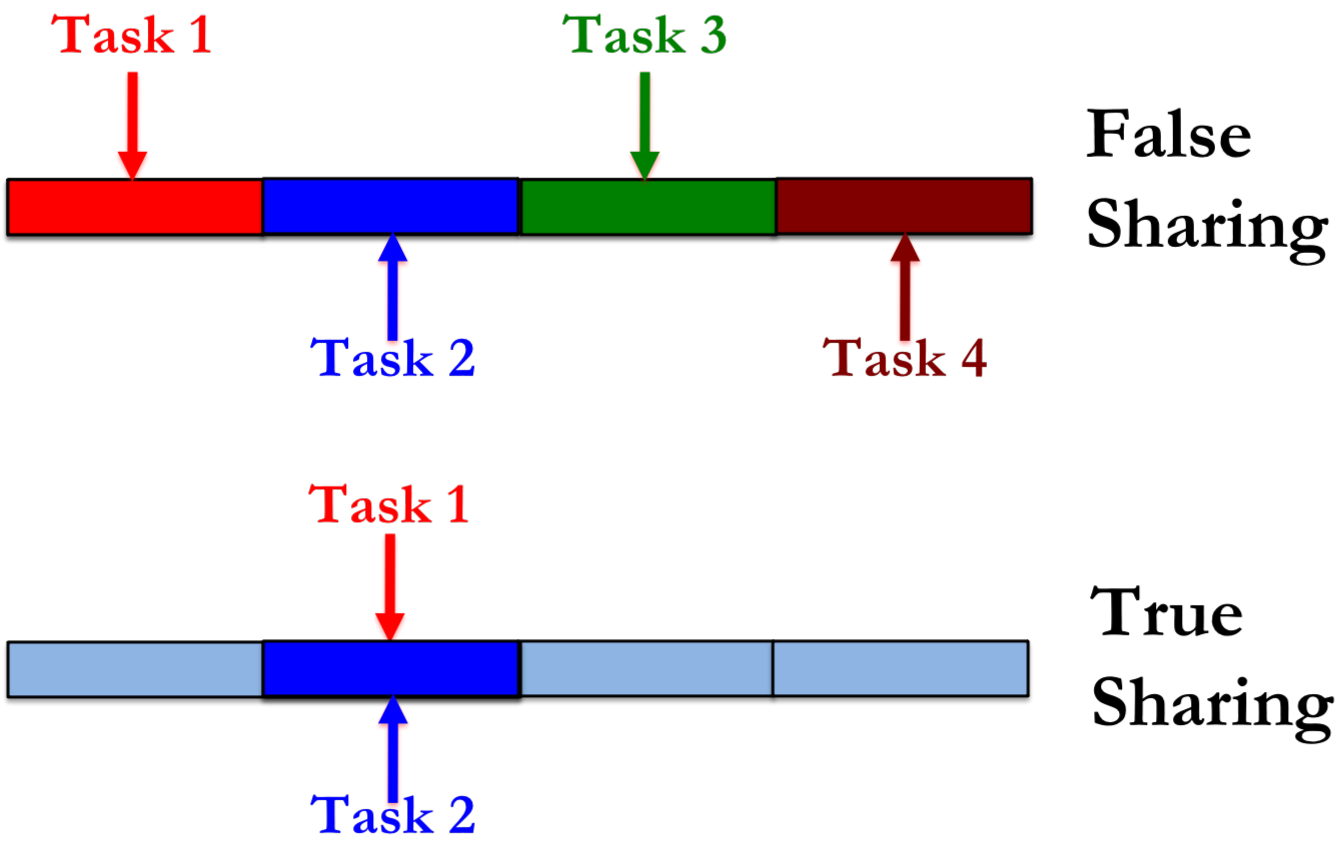
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing vs. True Sharing



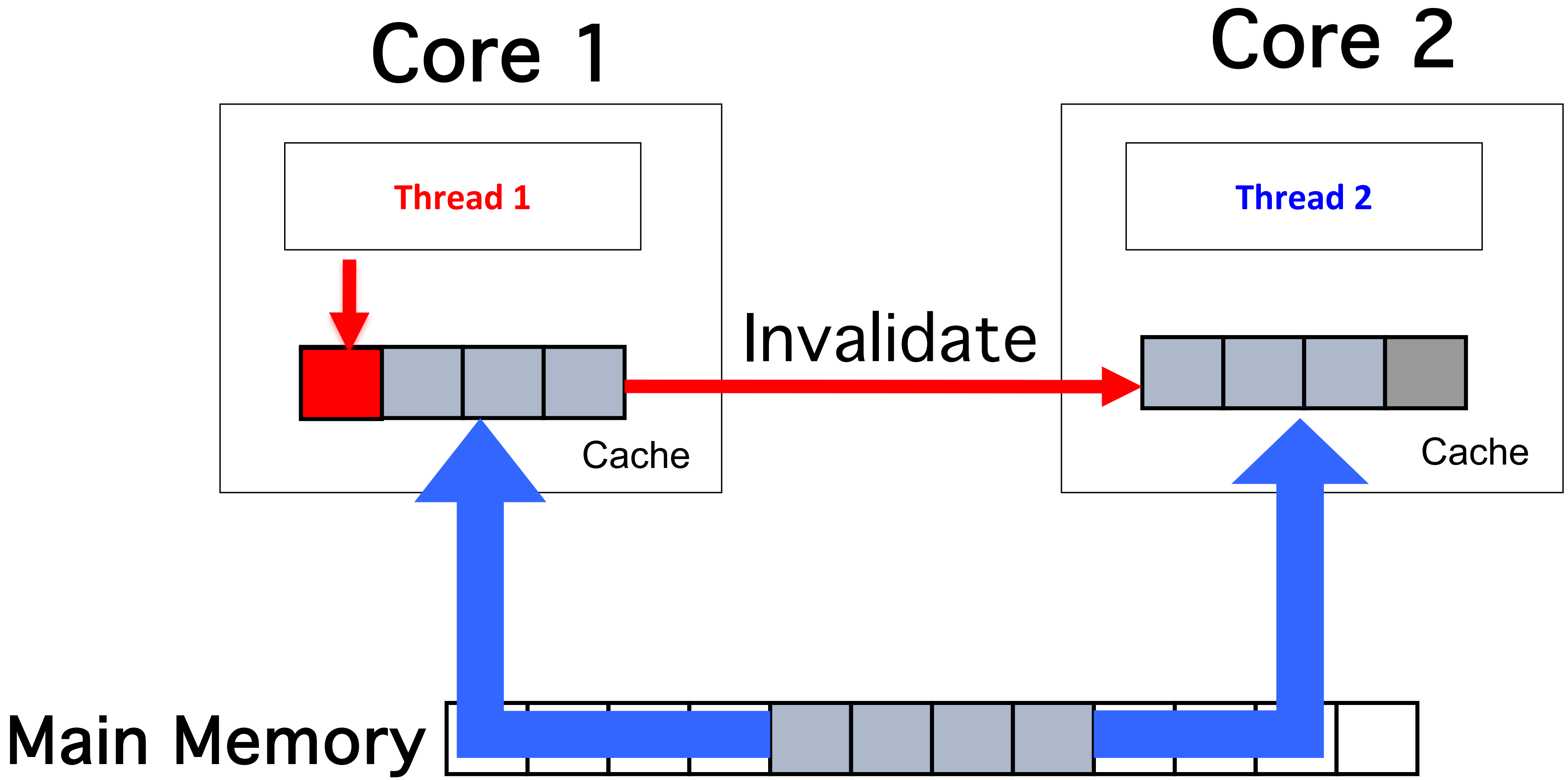
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing vs. True Sharing



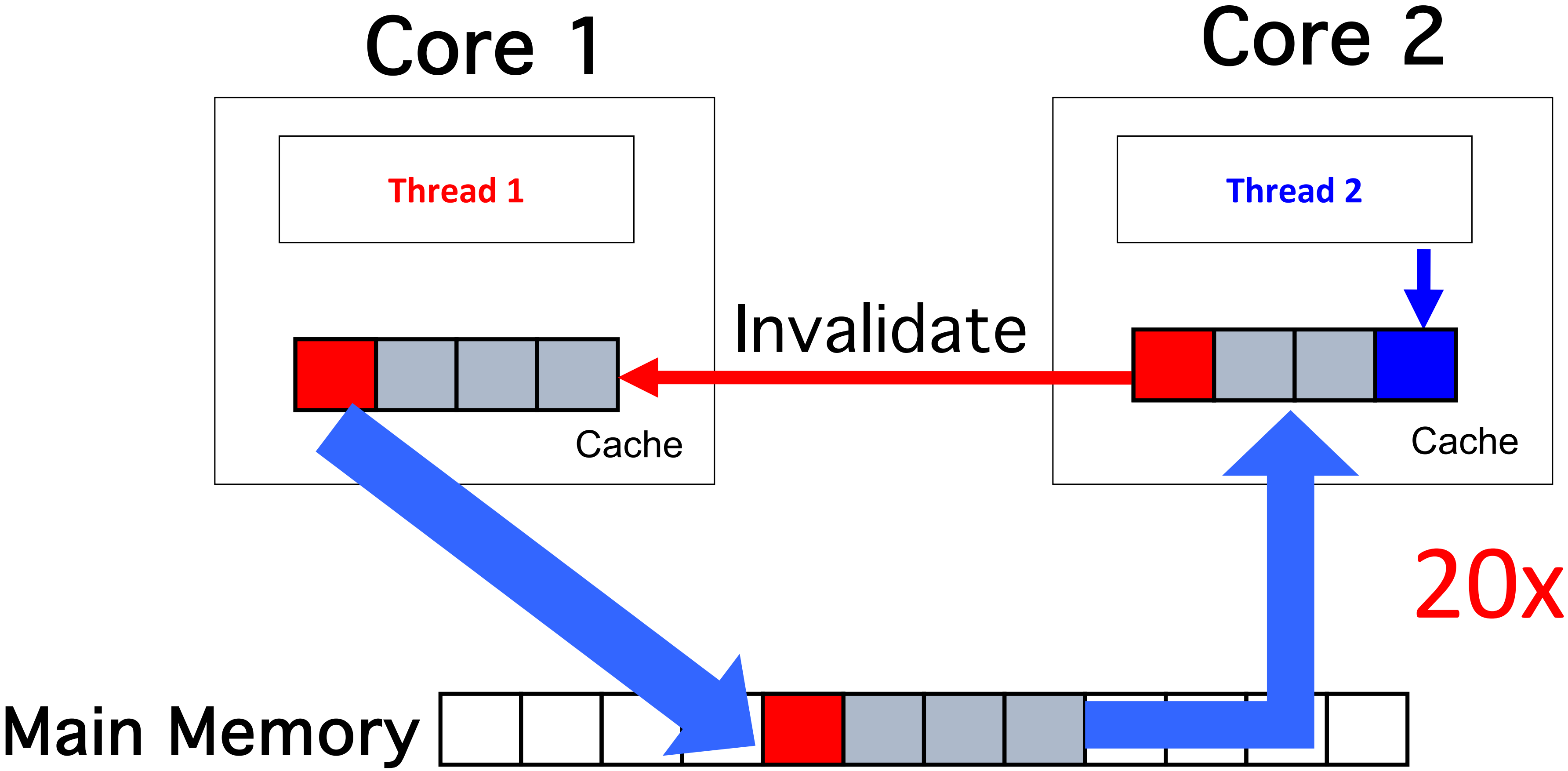
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing



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

False Sharing



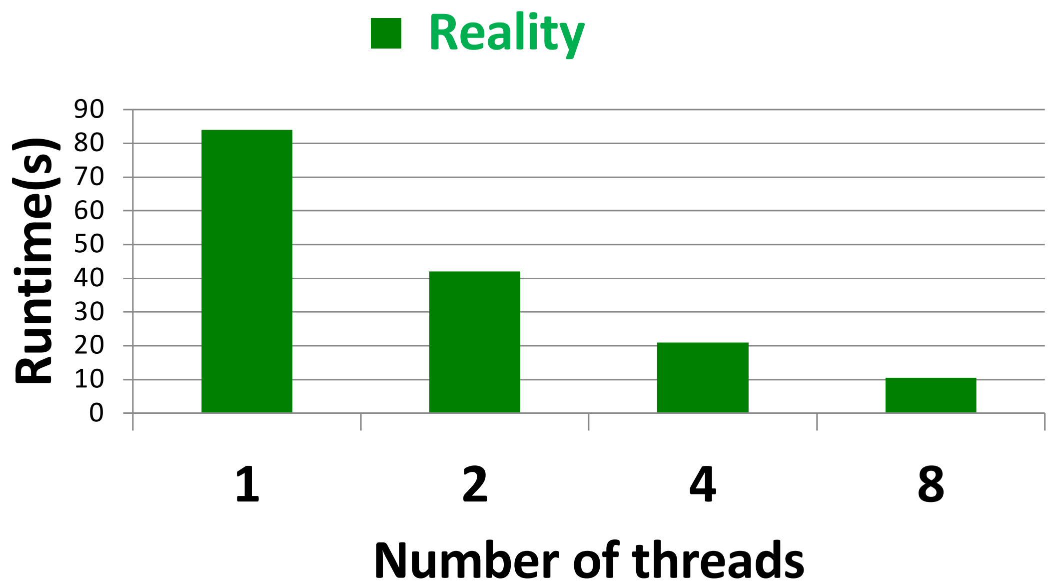
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

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?

```
/* global variables */
int sum[2];
int main() {
  /* heap allocations */
  int a = new int[size]; //initialized
  T1 = new thread([=]() {
    for(int i=0; i<size/2; i++) sum[0]+=a[i];
  });
  T2 = new thread([=]() {
    for(int i=size/2; i<size; i++) sum[1]+=a[i];
  });
  T1.join(); T2.join();
  //Cleanups
}
```

False sharing

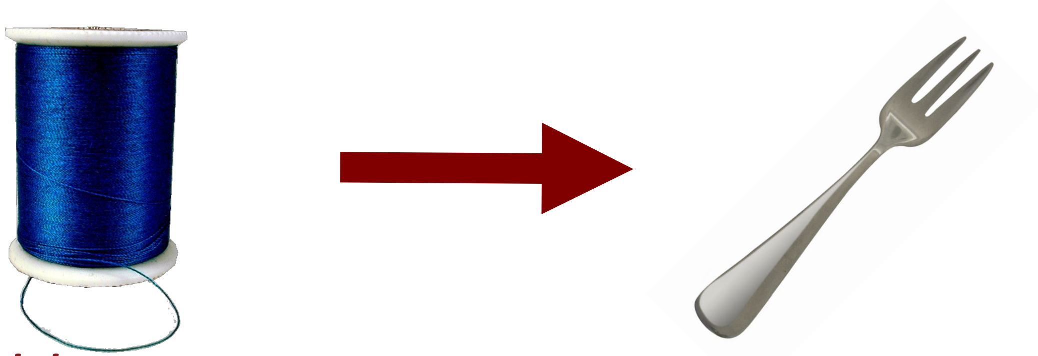
False sharing

Walkthrough of Sheriff Execution

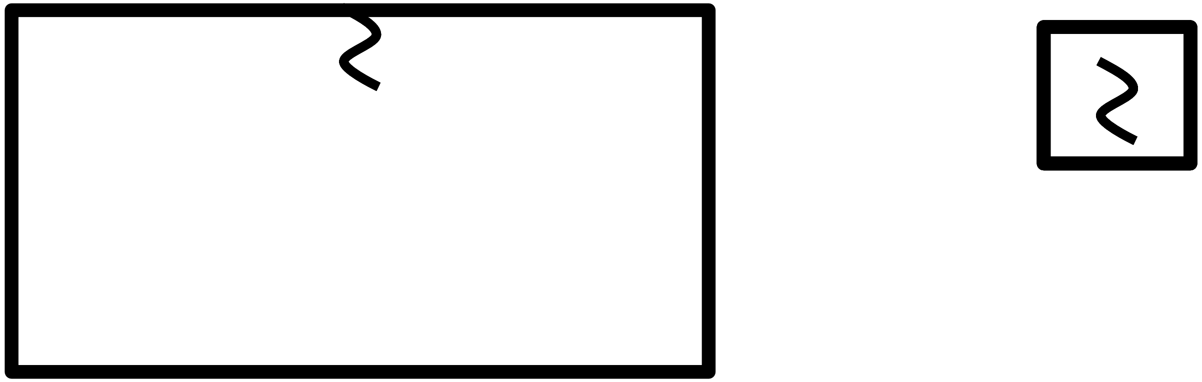
Process creation – creating processes instead of threads

```
/* global variables */
int sum[2];
int main() {
  /* heap allocations */
  int a = new int[size]; //initialized
  T1 = new thread([=]() {
    for(int i=0; i<size/2; i++) sum[0]+=a[i];
  });
  T2 = new thread([=]() {
    for(int i=size/2; i<size; i++) sum[1]+=a[i];
  });
  T1.join(); T2.join();
  //Cleanups
}
```

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)

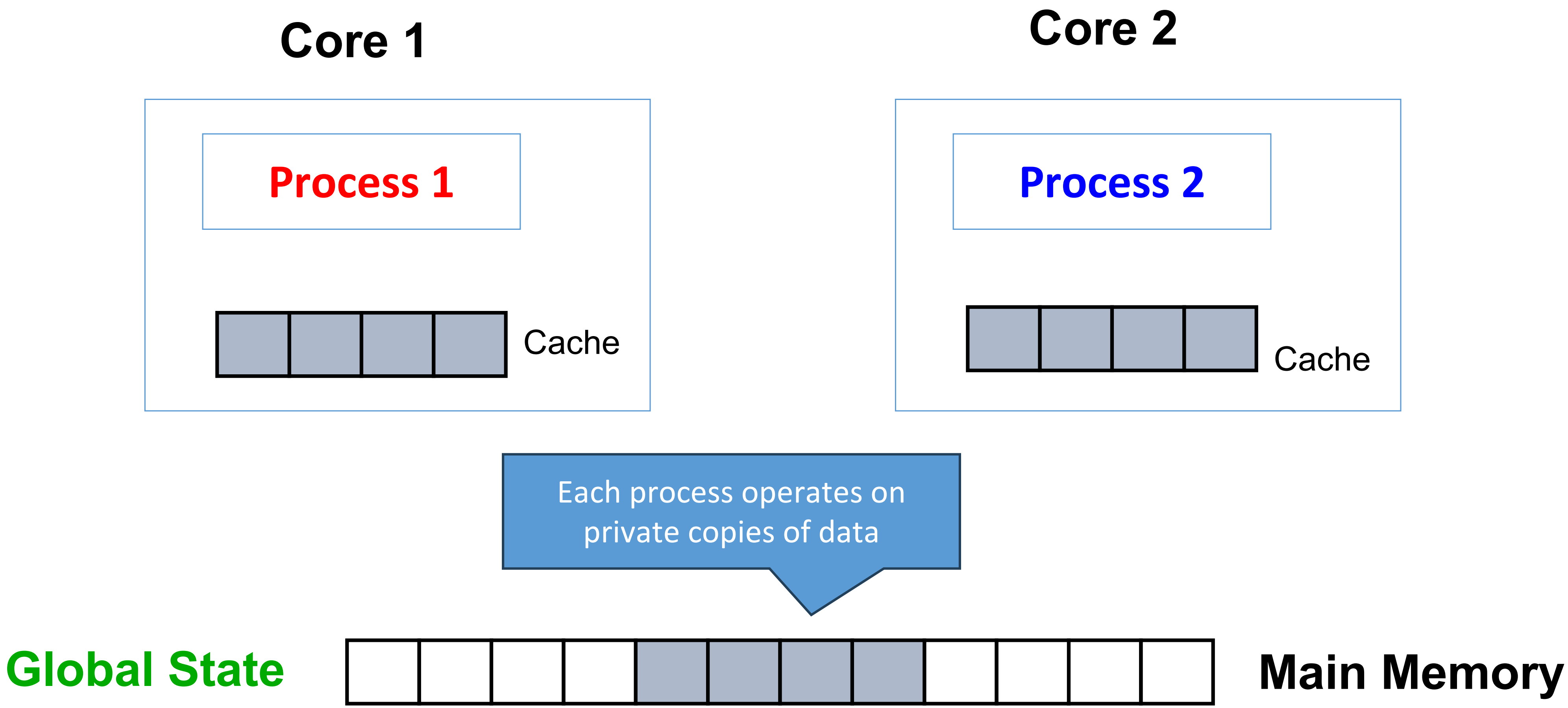
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Walkthrough of Sheriff Execution

Initialization – creating mapping of global and heap variables for processes

```
/* global variables */
int sum[2];
int main() {
  /* heap allocations */
  int a = new int[size]; //initialized
  T1 = new thread([=]() {
    for(int i=0; i<size/2; i++) sum[0]+=a[i];
  });
  T2 = new thread([=]() {
    for(int i=size/2; i<size; i++) sum[1]+=a[i];
  });
  T1.join(); T2.join();
  //Cleanups
}
```

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

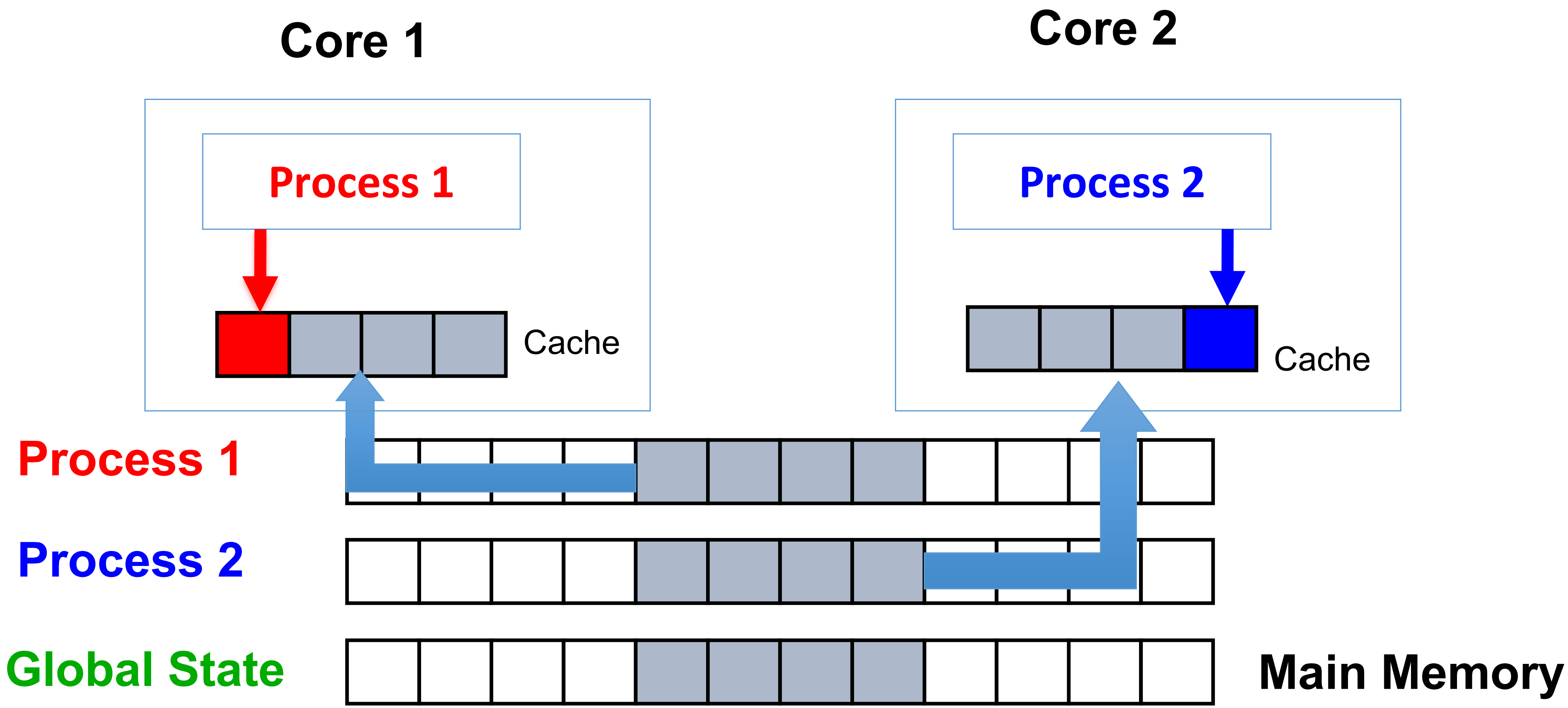
- Advantages of converting threads into processes
  - Enables the use of per-thread page protection, allowing Sheriff to track memory accesses by different threads (processes)
  - Each thread's (process) memory access are isolated, hence they would not update the same cache line
    - No false sharing!
- 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

```
/* global variables */
int sum[2];
int main() {
  /* heap allocations */
  int a = new int[size]; //initialized
  T1 = new thread([=]() {
    for(int i=0; i<size/2; i++) sum[0]+=a[i];
  });
  T2 = new thread([=]() {
    for(int i=size/2; i<size; i++) sum[1]+=a[i];
  });
  T1.join(); T2.join();
  //Cleanups
}
```

Execution – copies of memory pages per process for local updates

Sheriff Execution: Execution



Walkthrough of Sheriff Execution

```
/* global variables */
int sum[2];
int main() {
  /* heap allocations */
  int a = new int[size]; //initialized
  T1 = new thread([=]() {
    for(int i=0; i<size/2; i++) sum[0]+=a[i];
  });
  T2 = new thread([=]() {
    for(int i=size/2; i<size; i++) sum[1]+=a[i];
  });
  T1.join(); T2.join();
  //Cleanups
}
```

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

Sheriff Execution: Synchronization

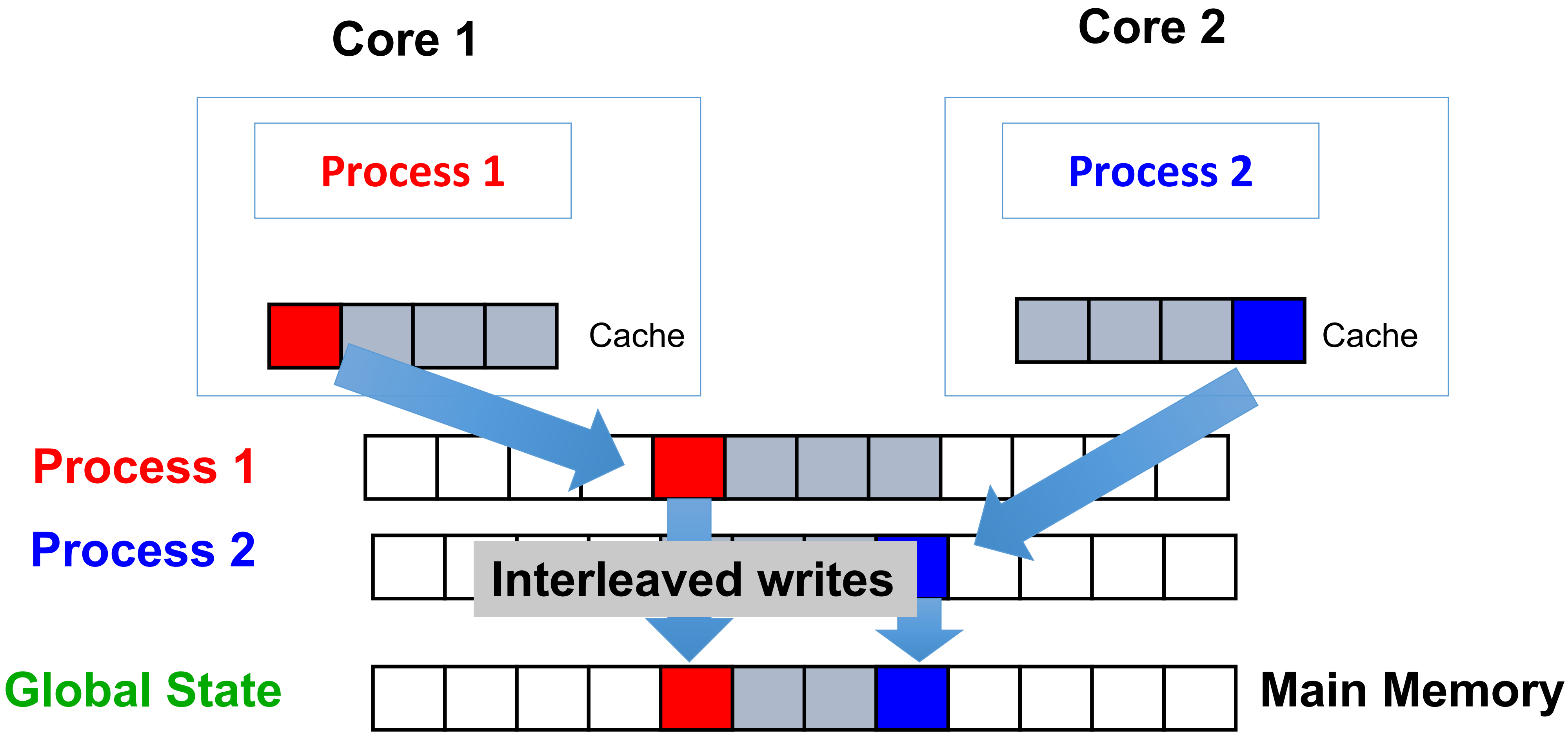
- There are two different types of synchronization points
  - Thread termination
  - End of the critical section (mutex unlock), barriers, etc.
- At each synchronization point, Sheriff commits changes from private pages to the shared pages
  - It commits only the differences between the twin and the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

Sheriff Execution: Synchronization

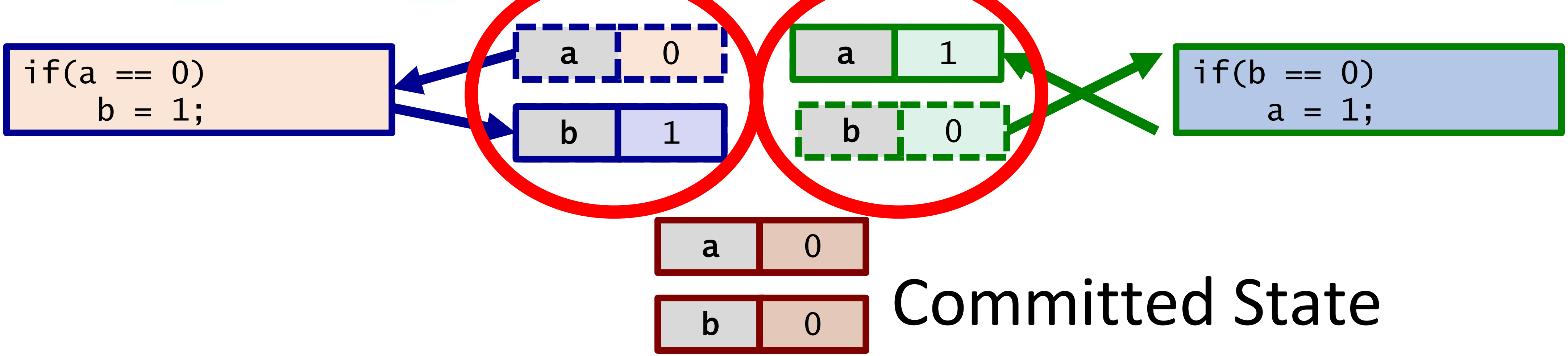
Snapshot and diffing the local changes

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

Sheriff Execution: Synchronization



Sheriff Execution: Synchronization



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

Reading Materials

- Sheriff
  - https://people.umass.edu/tongping/pubs/sheriff-oopsla11.pdf

Next Lecture

- Power management in multicore processors
- **Quiz-4**
    - **Syllabus: Lectures 14-17**