CS636: Shared Memory Programming and POSIX Threads

Swarnendu Biswas

Semester 2018-2019-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

How can we sum up all elements in an array?

```
int array[1000] = {0, 1, 34, 2, 89, -5, 67, 8, 4, 56,
                   23, 67, 0, 9, …}
```

$$sum = \sum_{i=1}^{n} array[i]$$

Comparing Implementations

Main Thread

```
long sum = 0;

for (int i =0; i < LEN; i++) {
  sum += array[i];
}
```

Comparing Implementations

Main Thread

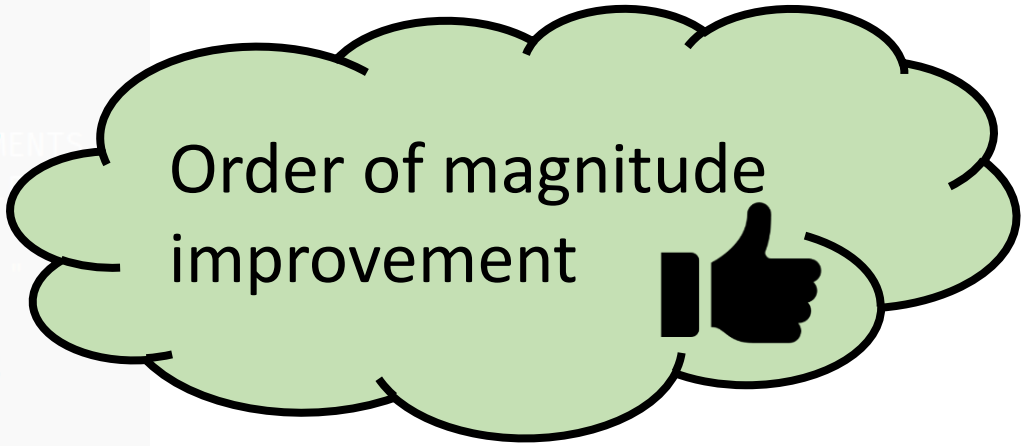
```
long sum = 0;

for (int i =0; i < LEN; i++) {
  sum += array[i];
}
```

Main Thread

```
  Spawn n threads
  long thr_sum[n] = {0}

  for (int i = 0; i < n; i++) {
    sum += thr_sum[i];
  }

Thread i

  Compute CHUNK i of array[]
  for (int i = CHUNK_START; i + CHUNK_START <
CHUNK_END; i++) {
    thr_sum[i] += array[i];
  }
```

Gains from Extra Complexity

```
1: fish  /home/swarnendu/iitk-workspace/c++-examples/src
~/i/c/src $ ./a.out
Sequential sum: 499158189 Time (ns): 2119657

Parallel sum: 499158189 Time (ns): 147934

~/i/c/src $ ./a.out
Sequential sum: 499019481 Time (ns): 2063707

Parallel sum: 499019481 Time (ns): 259234

~/i/c/src $ ./a.out
Sequential sum: 498973205 Time (ns): 2113602

Parallel sum: 498973205 Time (ns): 257328

~/i/c/src $ ./a.out
Sequential sum: 499697650 Time (ns): 2110496

Parallel sum: 499697650 Time (ns): 252351

~/i/c/src $
```



Parallel Programming Overview

Find parallelization opportunities in the problem

- Decompose the problem into parallel units

Parallel Programming Overview

Find parallelization opportunities in the problem

- Decompose the problem into parallel units

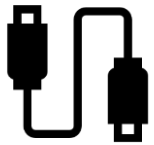
Create parallel units of execution

- Manage efficient execution of the parallel units

Parallel Programming Overview

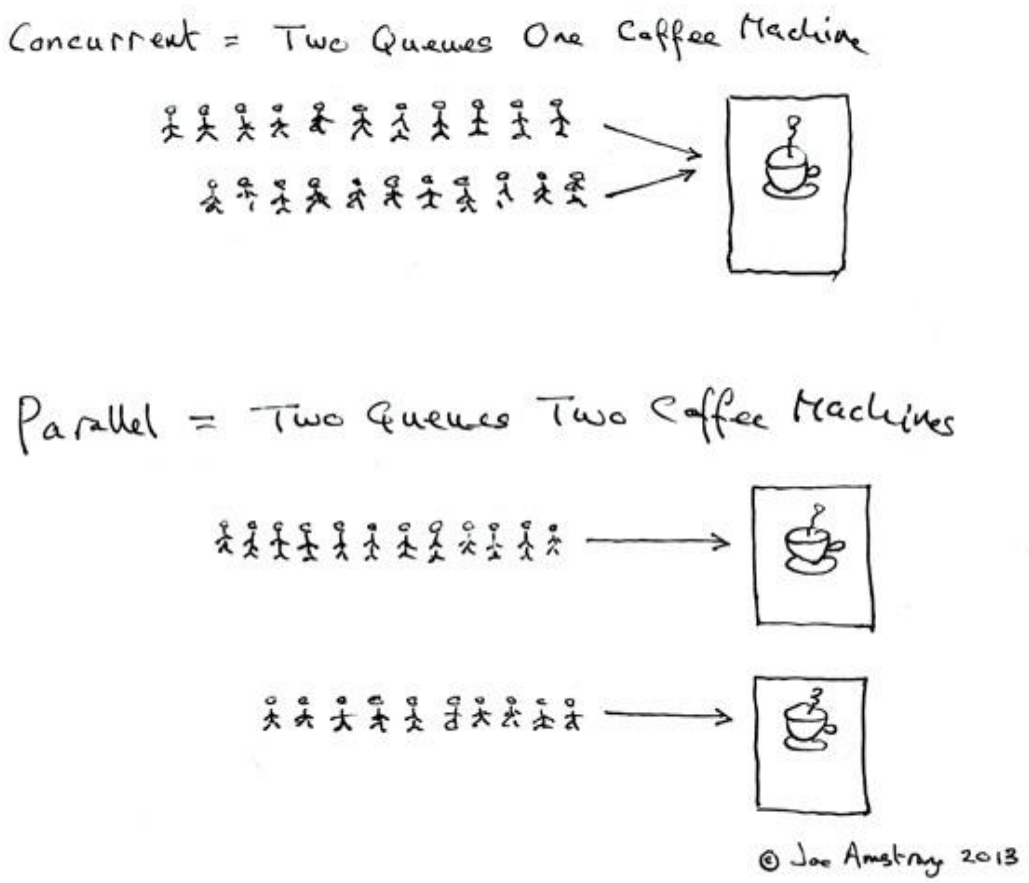
Find parallelization opportunities in the problem
- Decompose the problem into parallel units

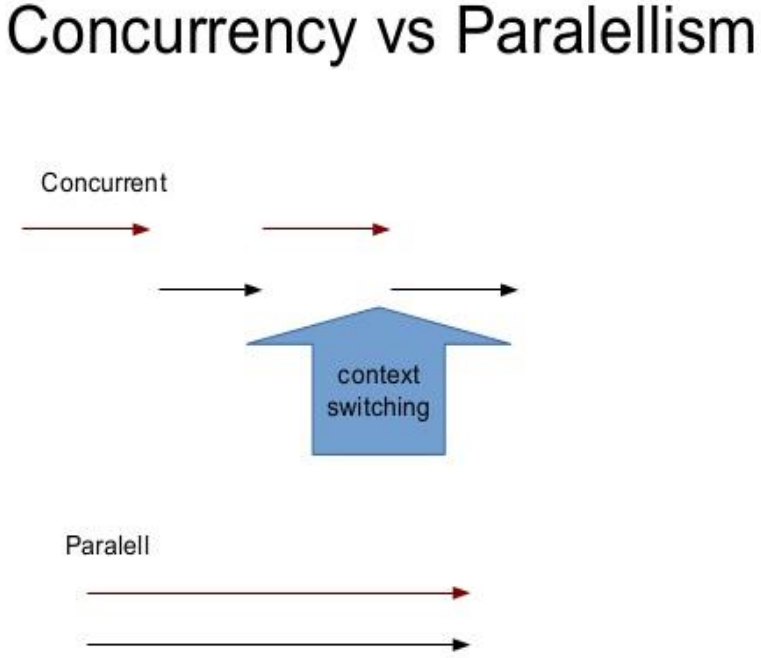
Create parallel units of execution
- Manage efficient execution of the parallel units



Problem may require inter-unit communication
- Communication between threads, cores, …

Parallelism vs Concurrency





Parallelism vs Concurrency

Parallel programming

- Use additional resources to speed up computation
- Performance perspective

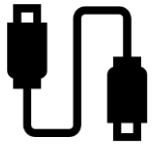
Concurrent programming

- Correct and efficient control of access to shared resources
- Correctness perspective

Distinction is not absolute

Inter-unit Communication

The problem logic will possibly require inter-unit communication



Units may be on the same processor or across processors or across nodes

What do we communicate in sequential programs?

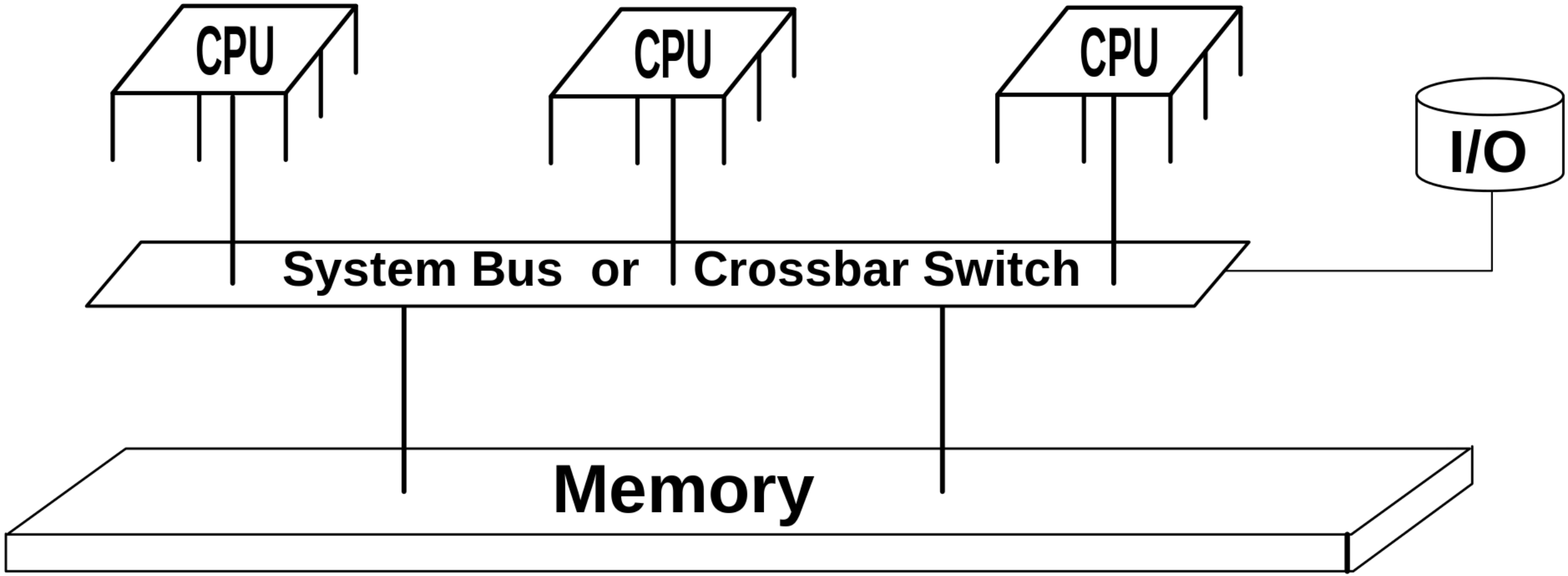
What do we communicate in sequential programs?

- Global variables or data structures
- Function arguments and call parameters

Shared Memory Programming

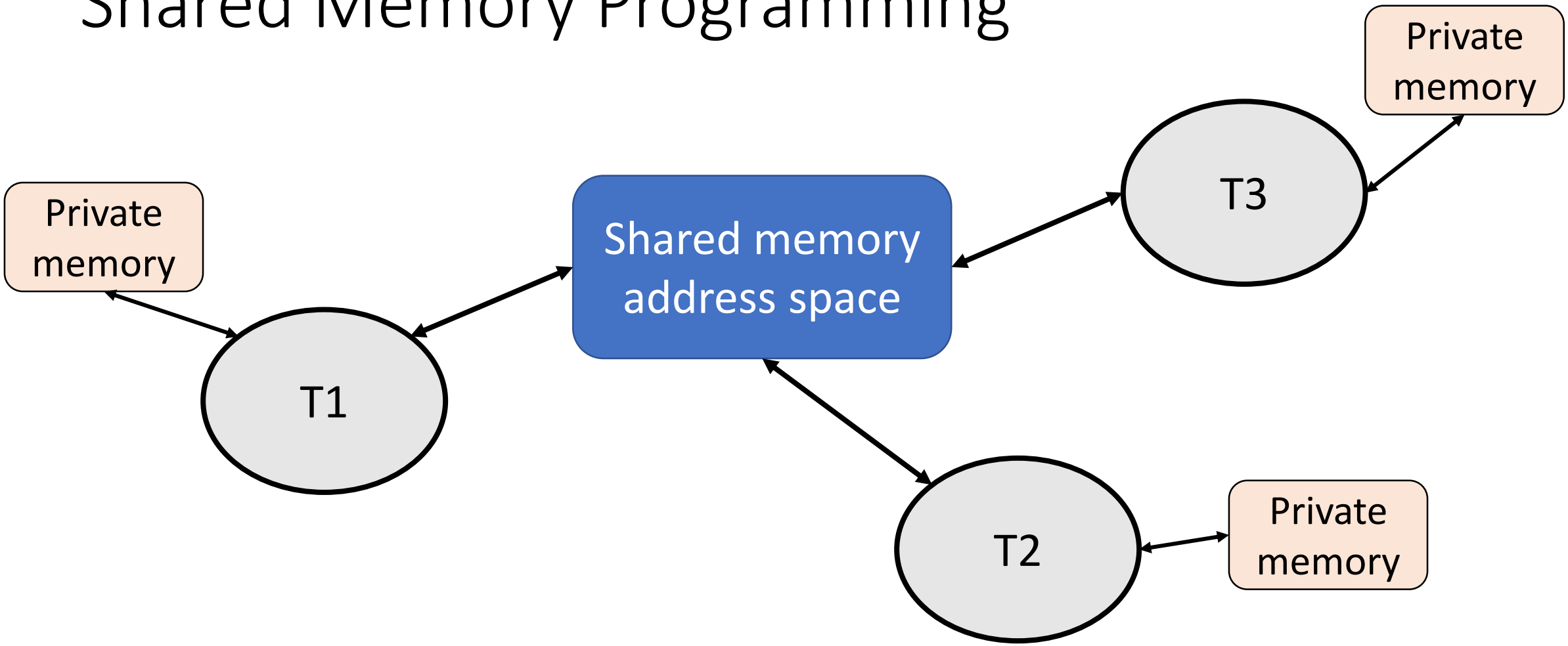
- Single address space shared by multiple processes
  - Can also be threads: static or dynamic
  - A memory location can be accessed by any process

- Relatively easy to program, avoids redundant data



Wikipedia.

Shared Memory Programming



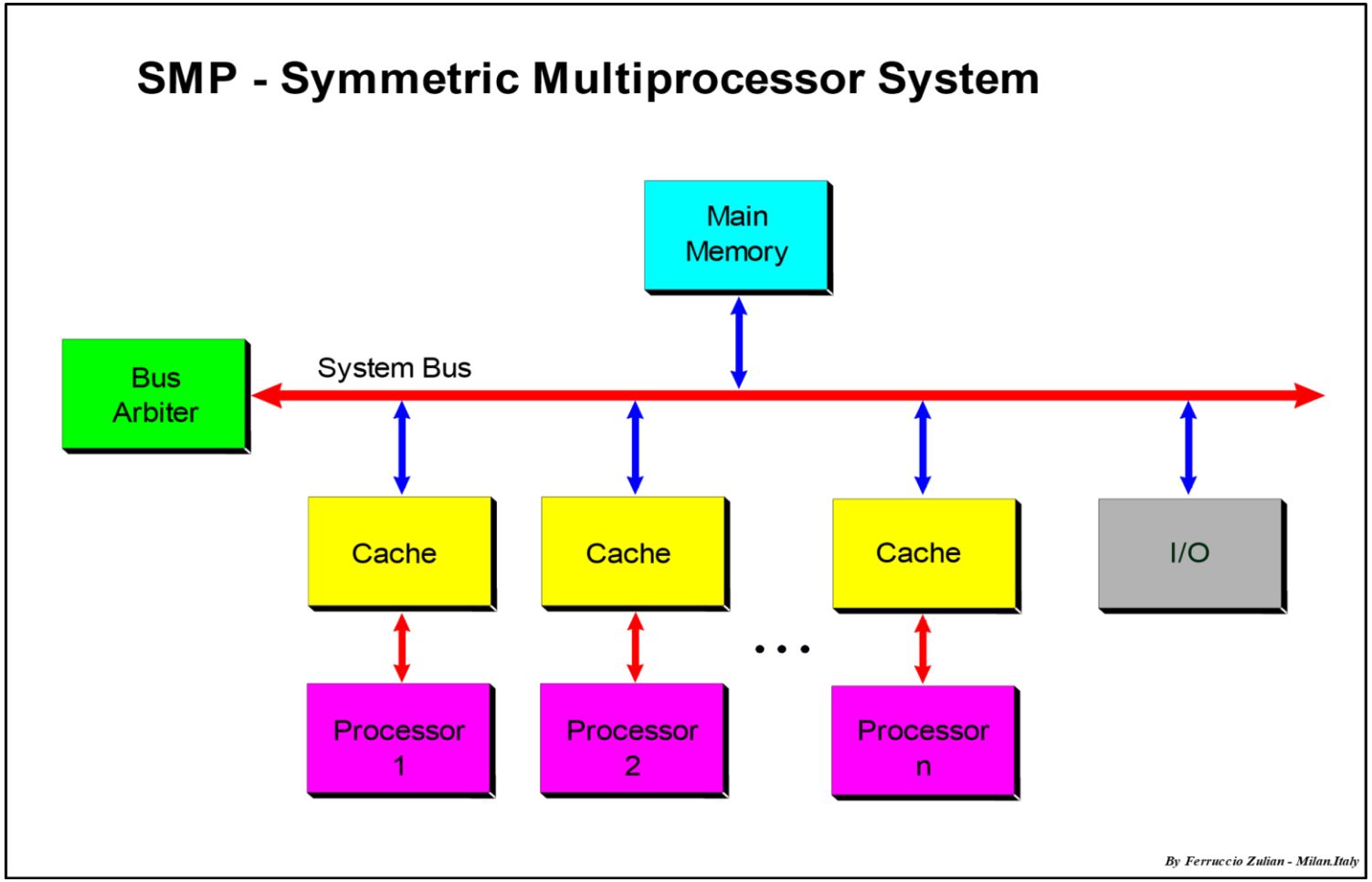
Shared Memory Programming

- Threads also have a private memory

- All threads access the shared address space
- Synchronization is required to access shared resources
  - Can otherwise lead to pernicious bugs

Implementing Shared Memory

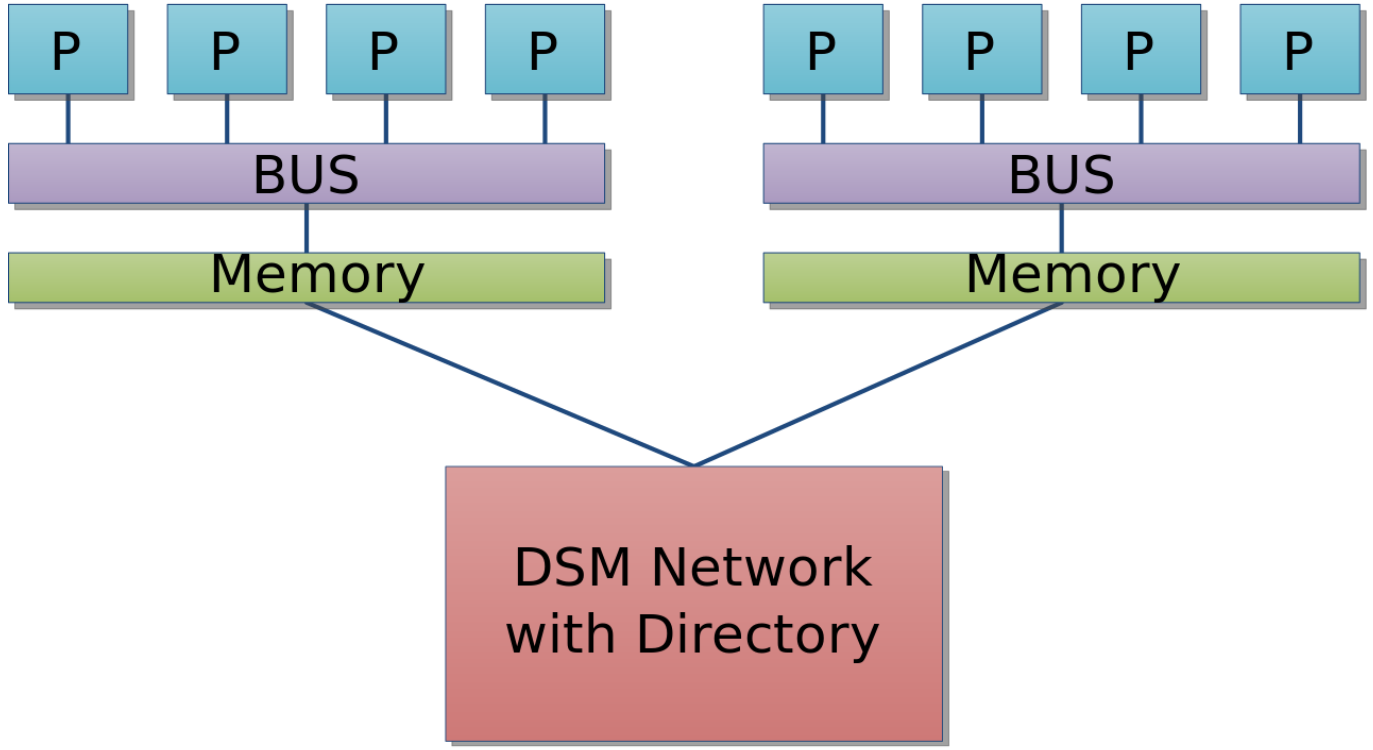
- Uniform memory access



Wikipedia.

Implementing Shared Memory

- Non-uniform memory access



Challenges with Shared Memory

Challenges with Shared Memory

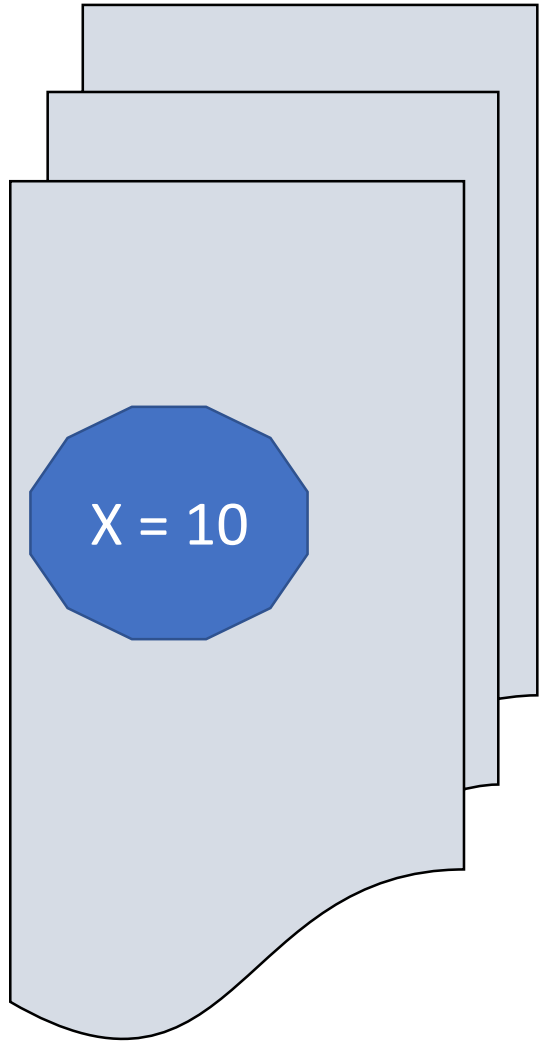
- Need support for cache coherence

Sequence of Operations

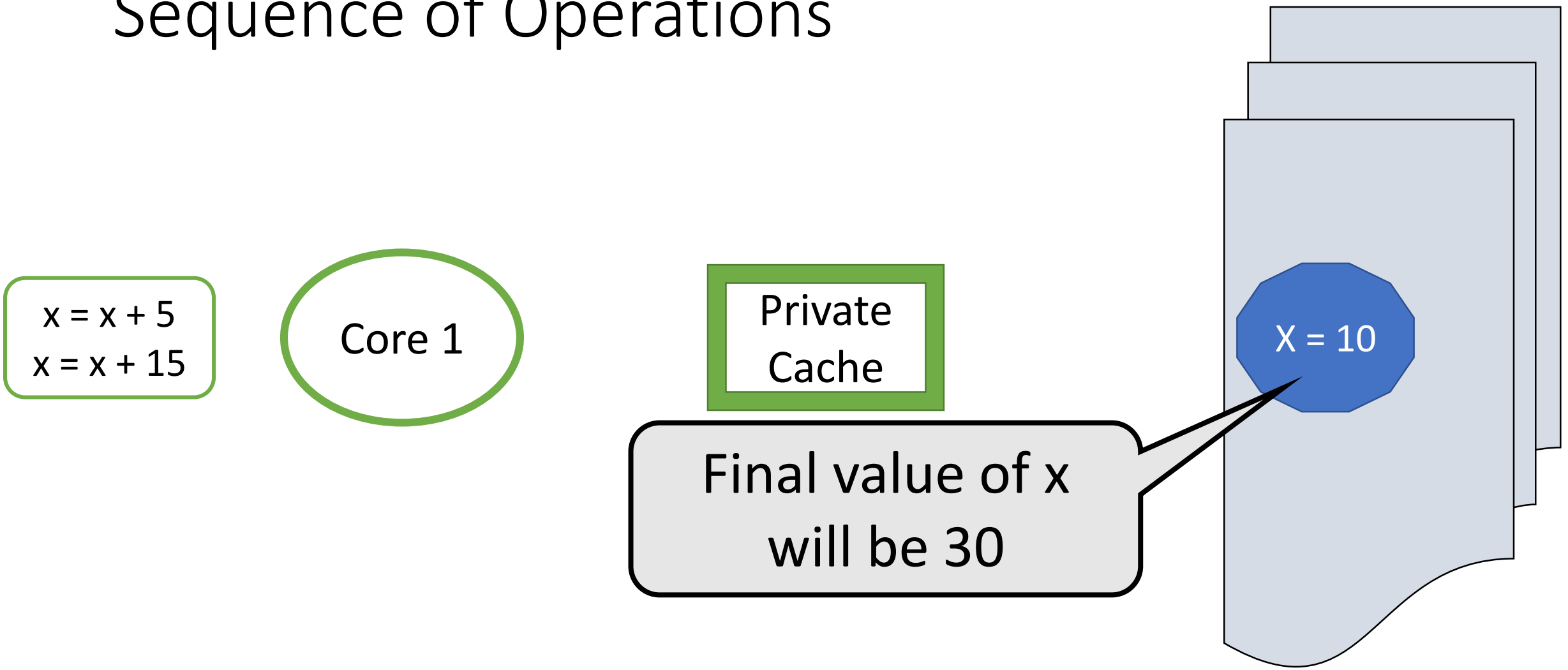
x = x + 5
x = x + 15

Core 1

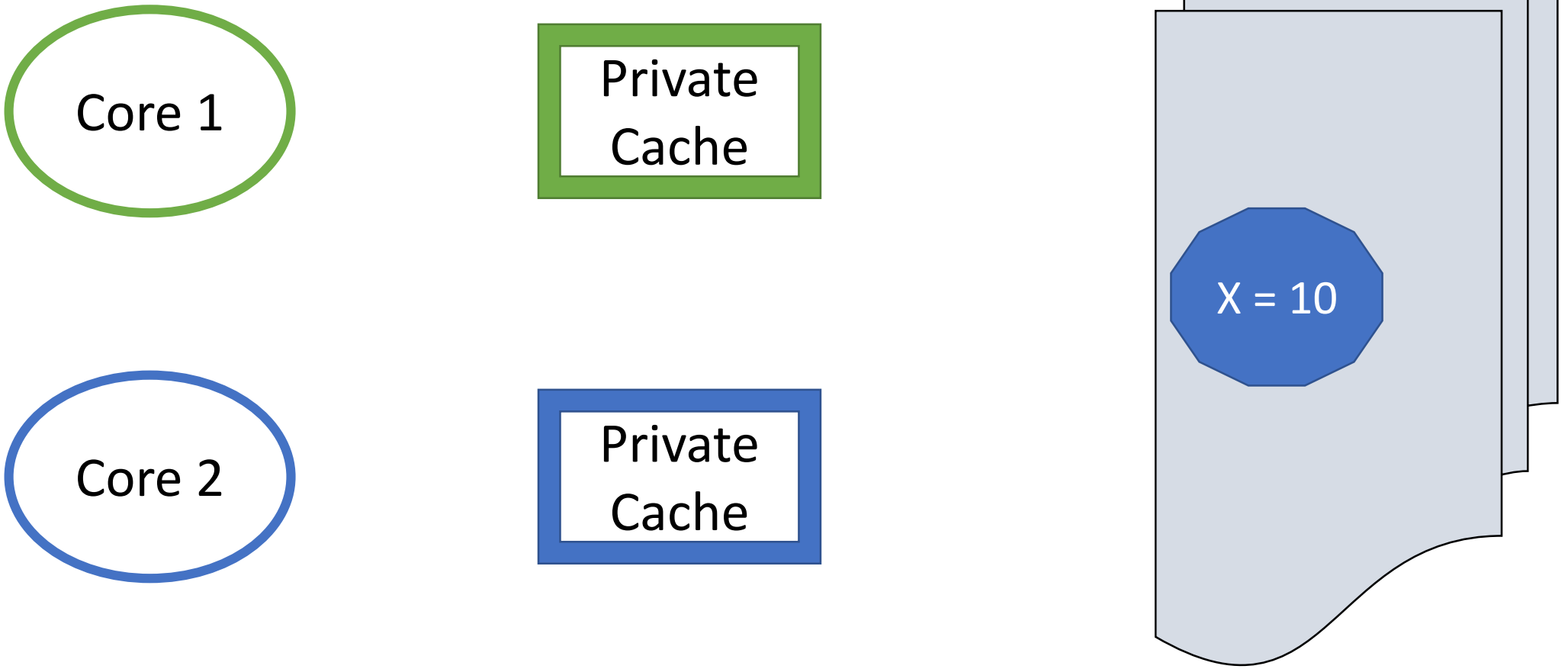
Private Cache



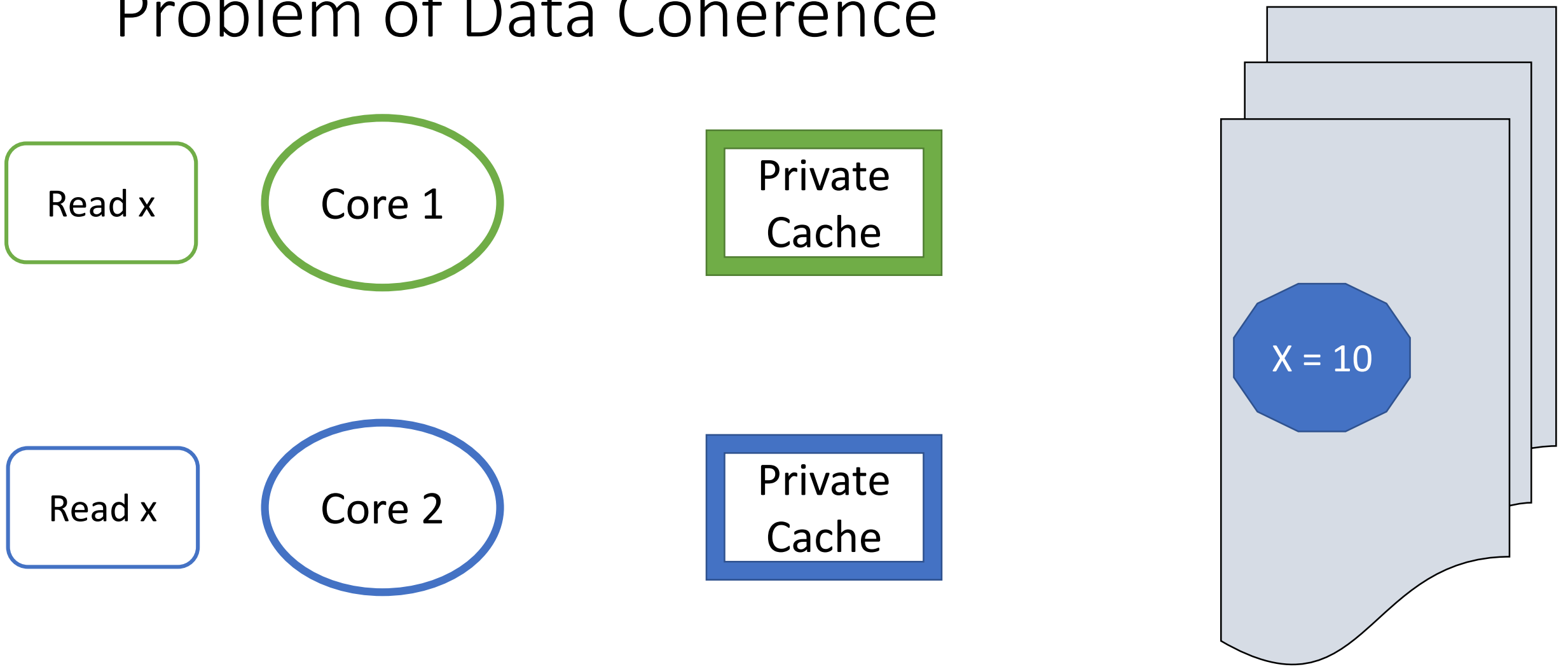
Sequence of Operations



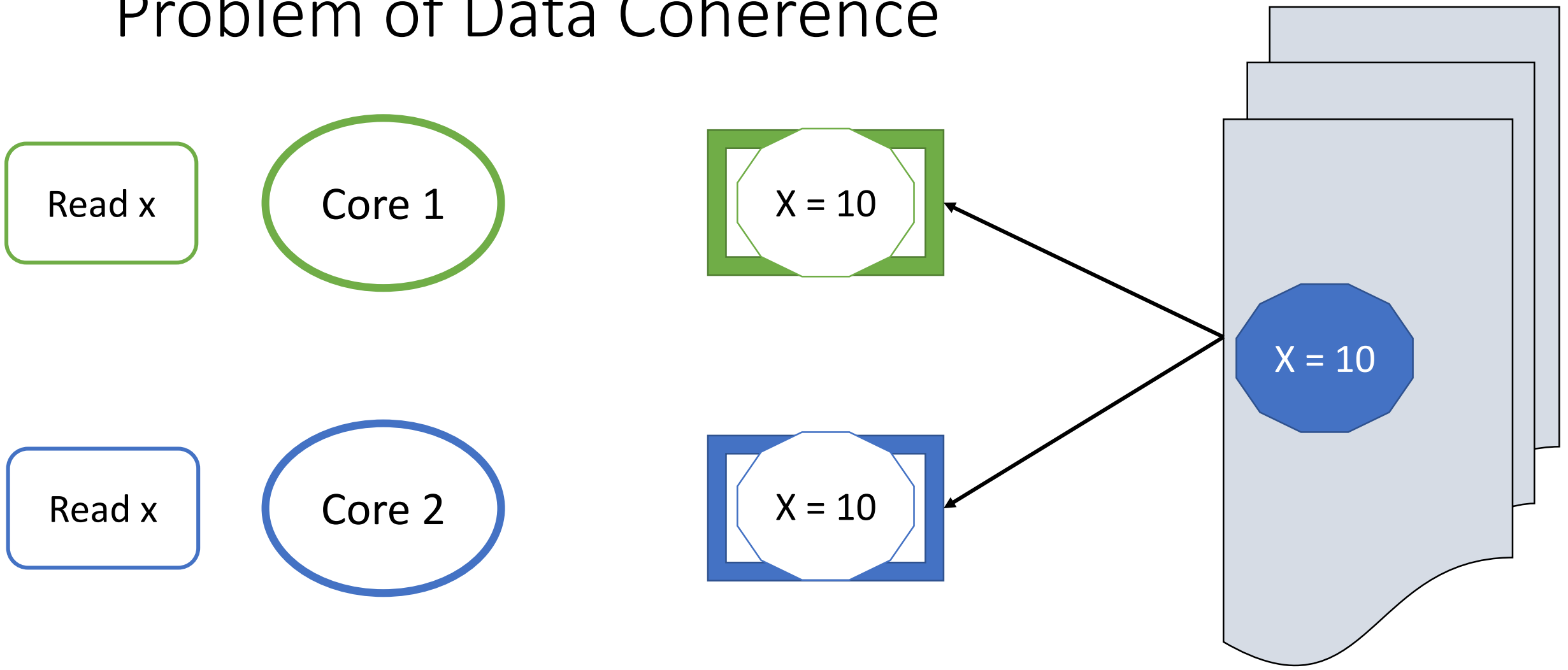
Problem of Data Coherence



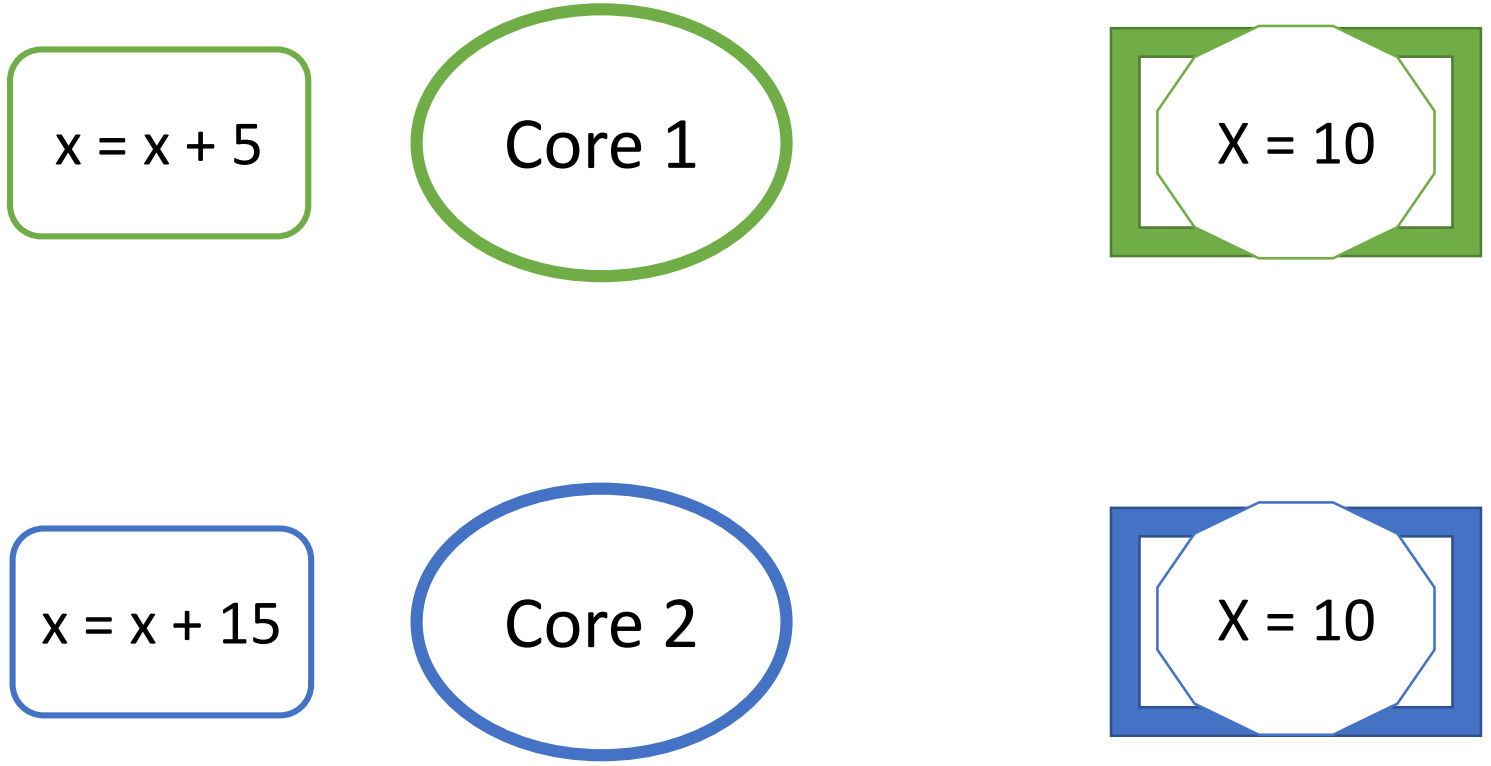
Problem of Data Coherence

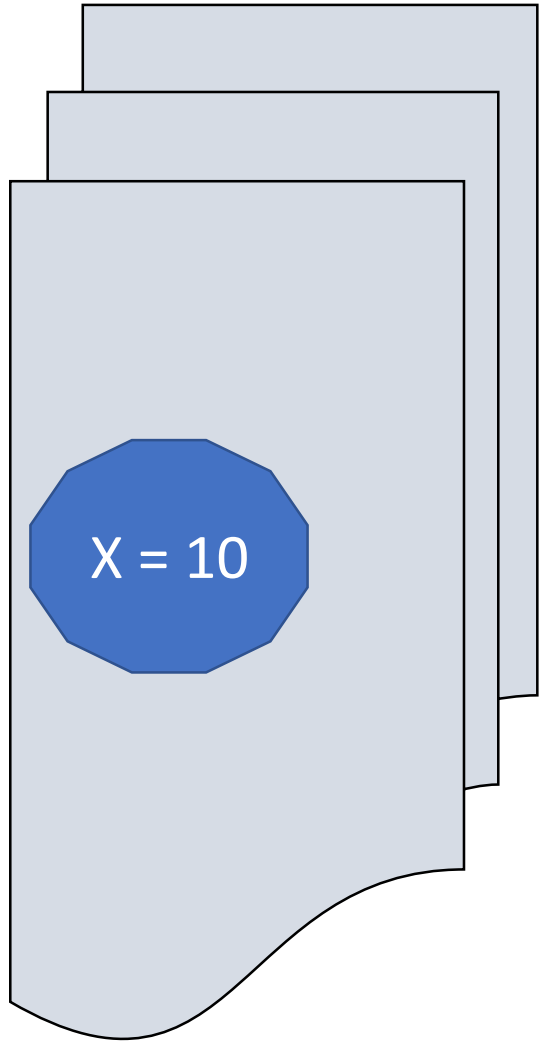


Problem of Data Coherence

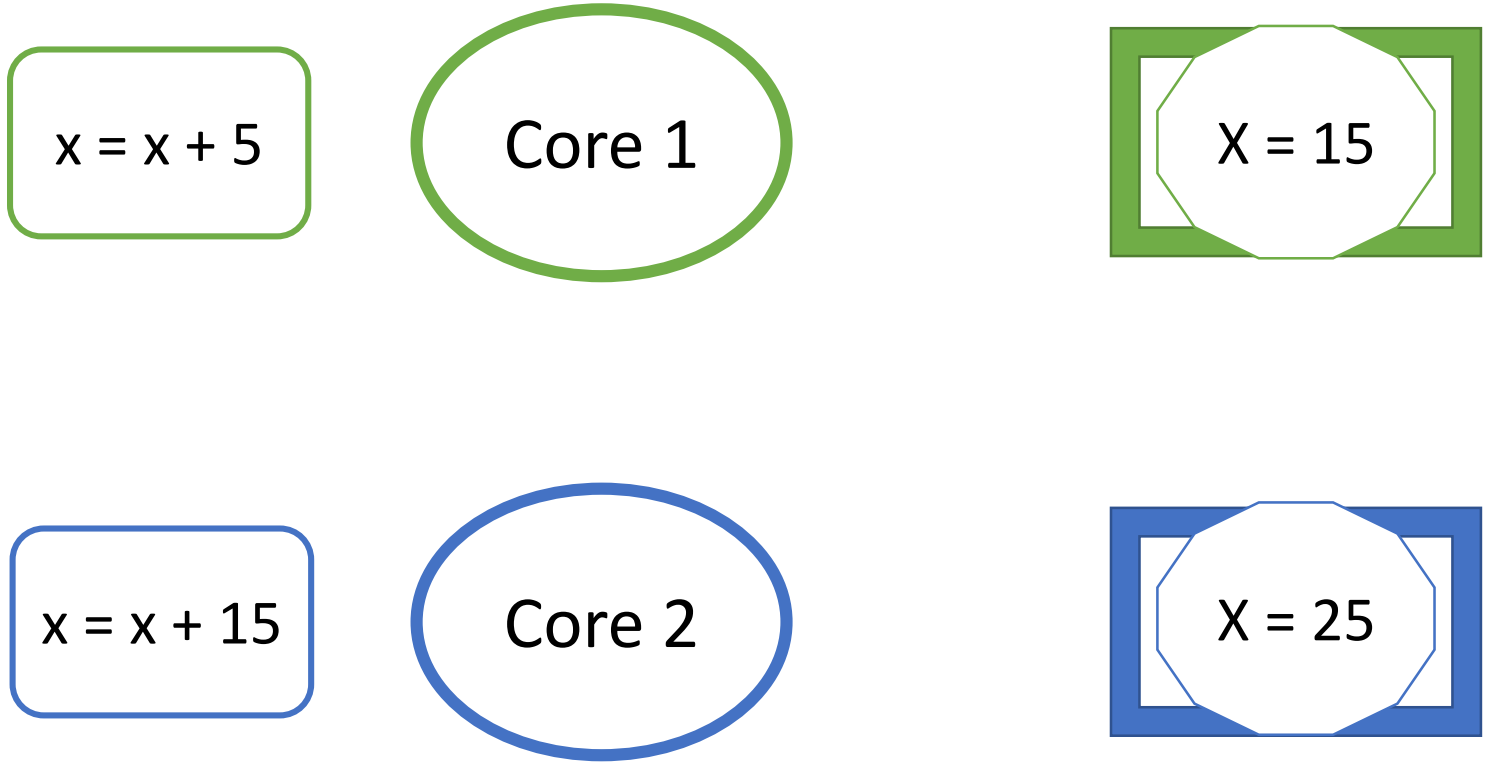


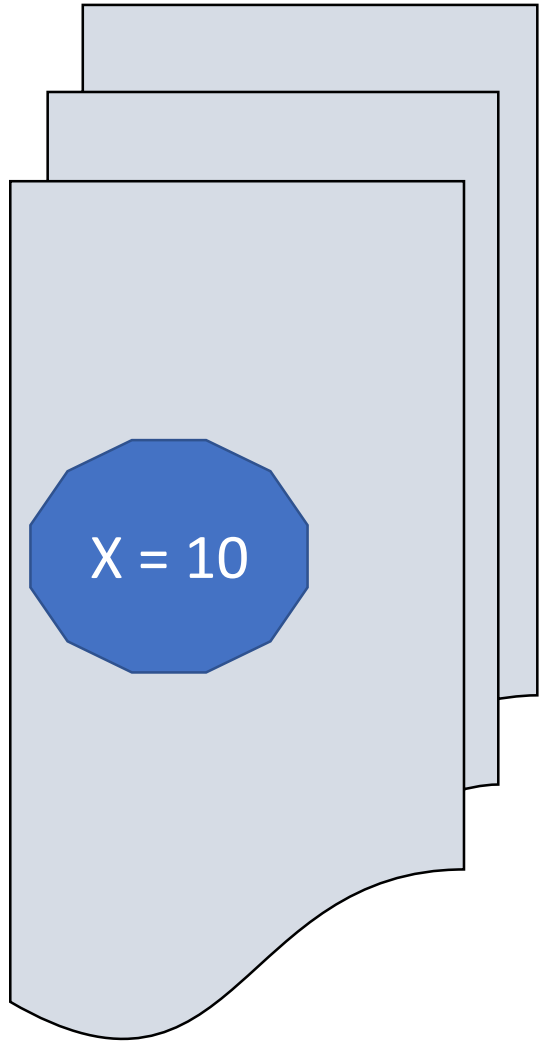
Problem of Data Coherence



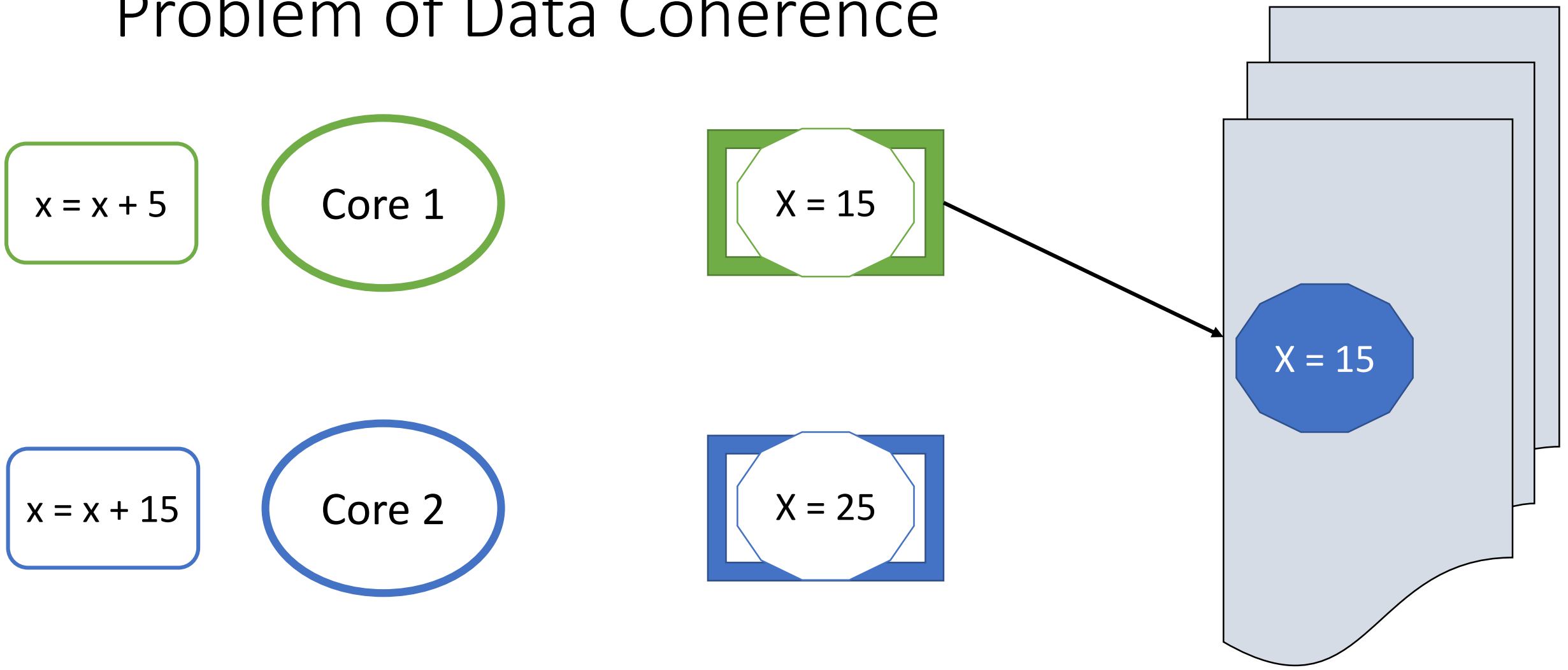


Problem of Data Coherence





Problem of Data Coherence



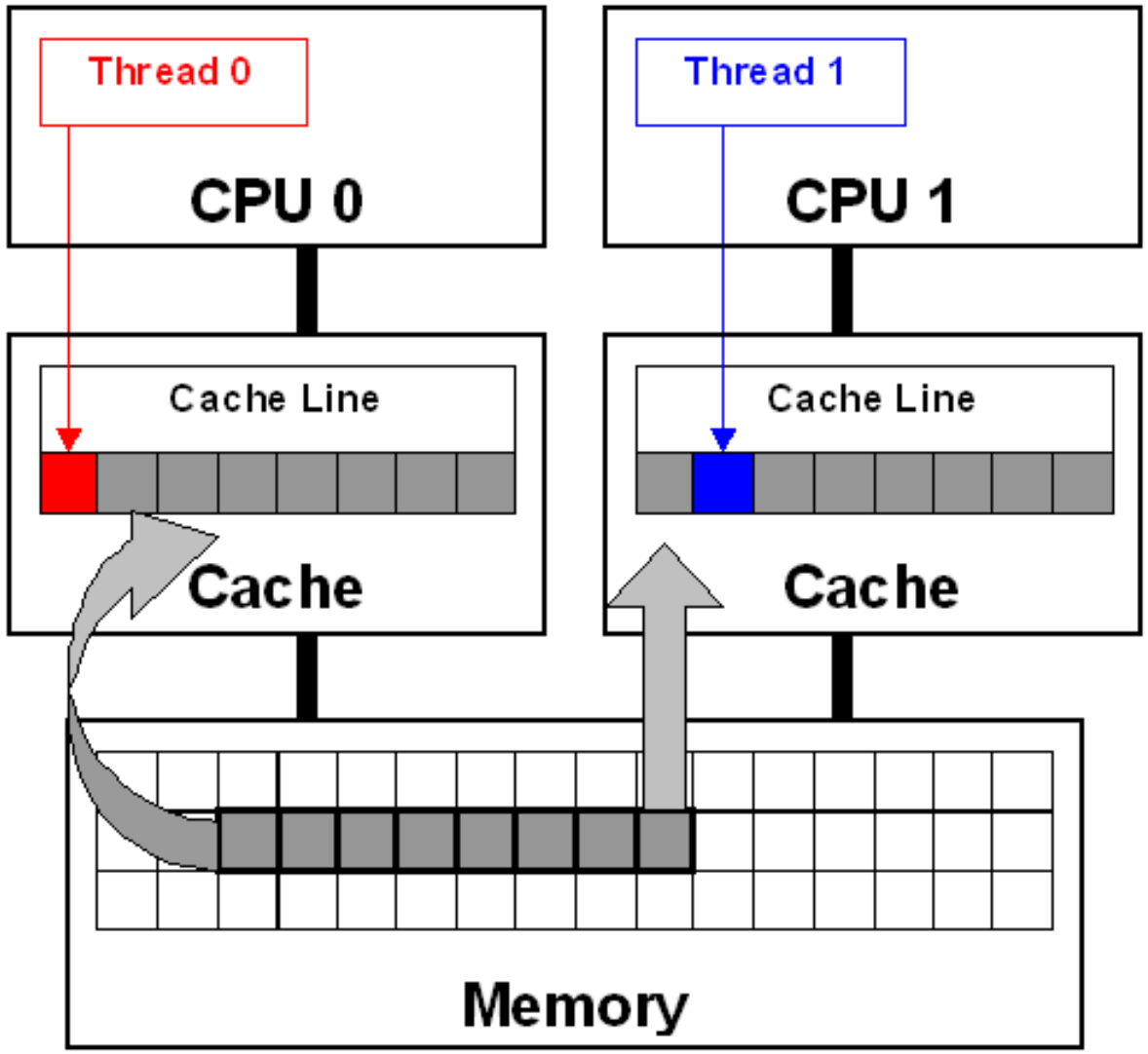
Problem of Data Coherence

Challenges with Shared Memory

- Access conflicts
  - Several threads can try to access the same shared location
  - Other performance hazards – false sharing
  - Coherence operations can become a bottleneck

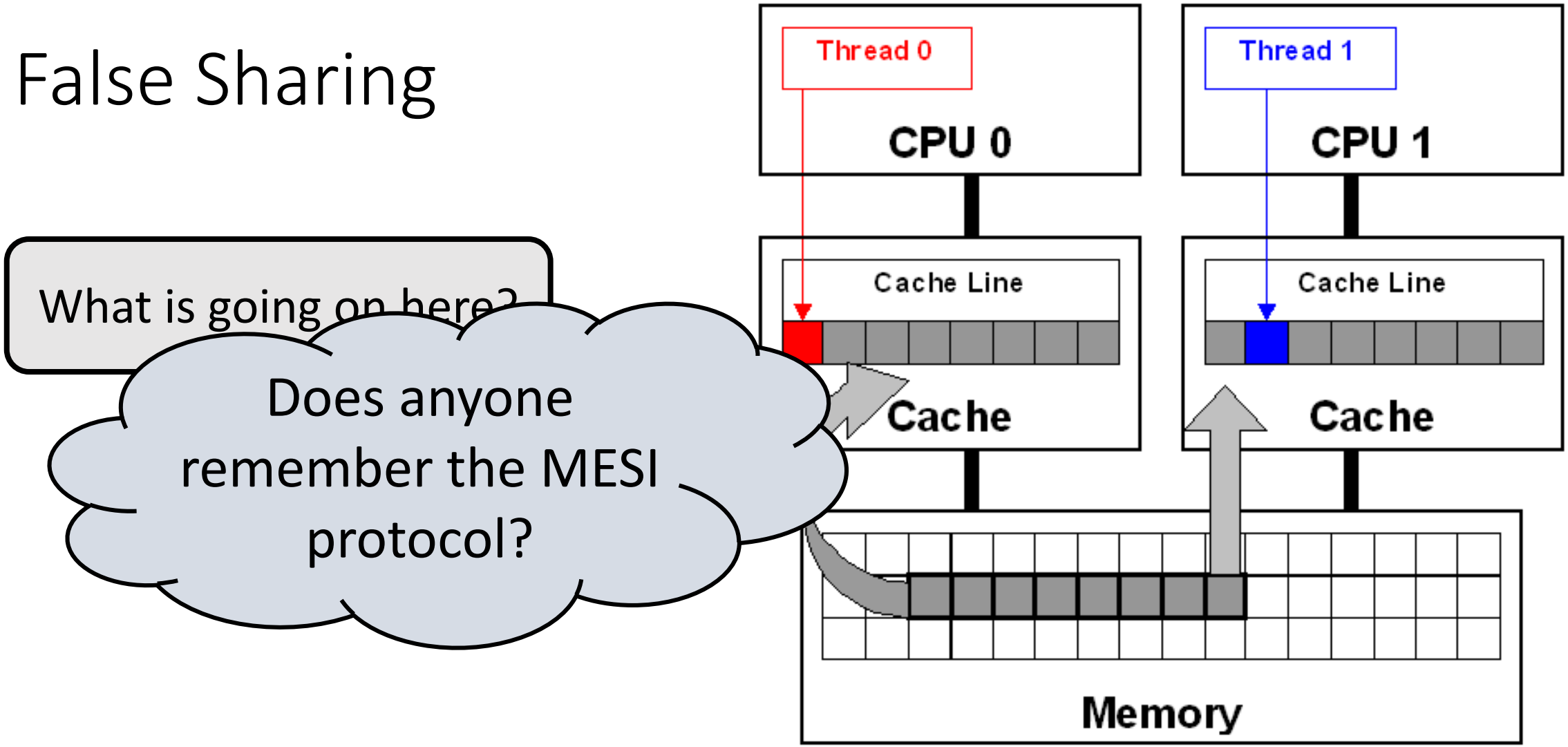
False Sharing

What is going on here?



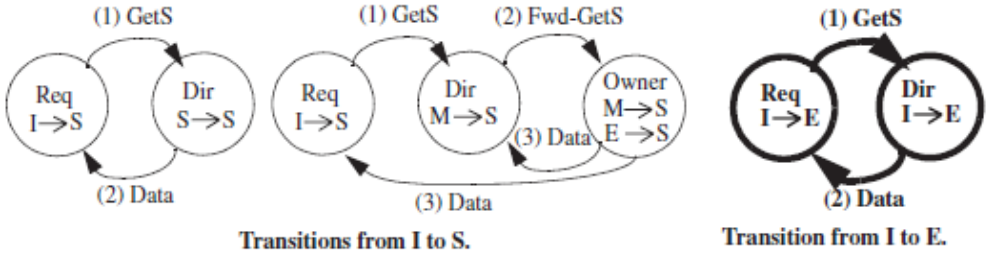
Intel. Avoiding and Identifying False Sharing Among Threads.

False Sharing



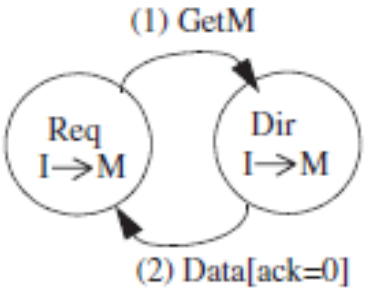
Intel. Avoiding and Identifying False Sharing Among Threads.

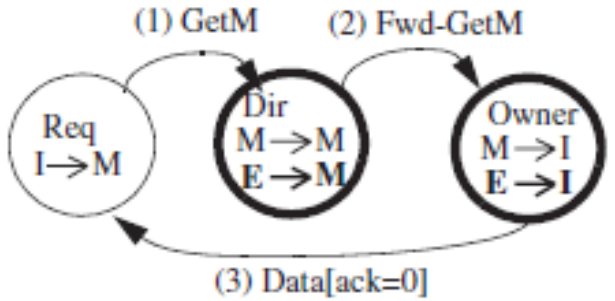
State Transitions in MESI

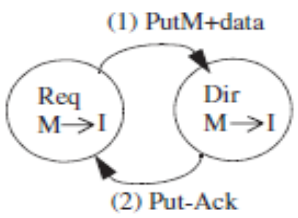


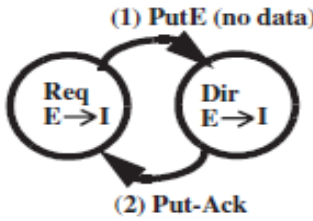
Transitions from I to S.

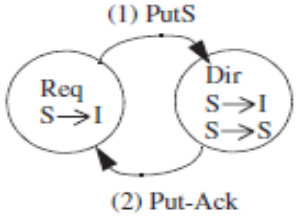
Transition from I to E.



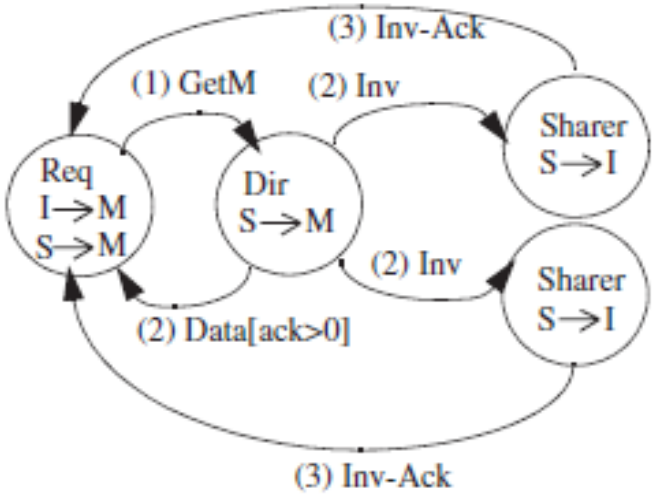








Transition from M or E or S to I



The only sharer might be the requestor, in which case no Invalidation messages are sent and the Data message from the Dir to Req has an AckCount of zero.

Transitions from I or S to M. Transition from E to M is silent.

A Primer on Memory Consistency and Cache Coherence.

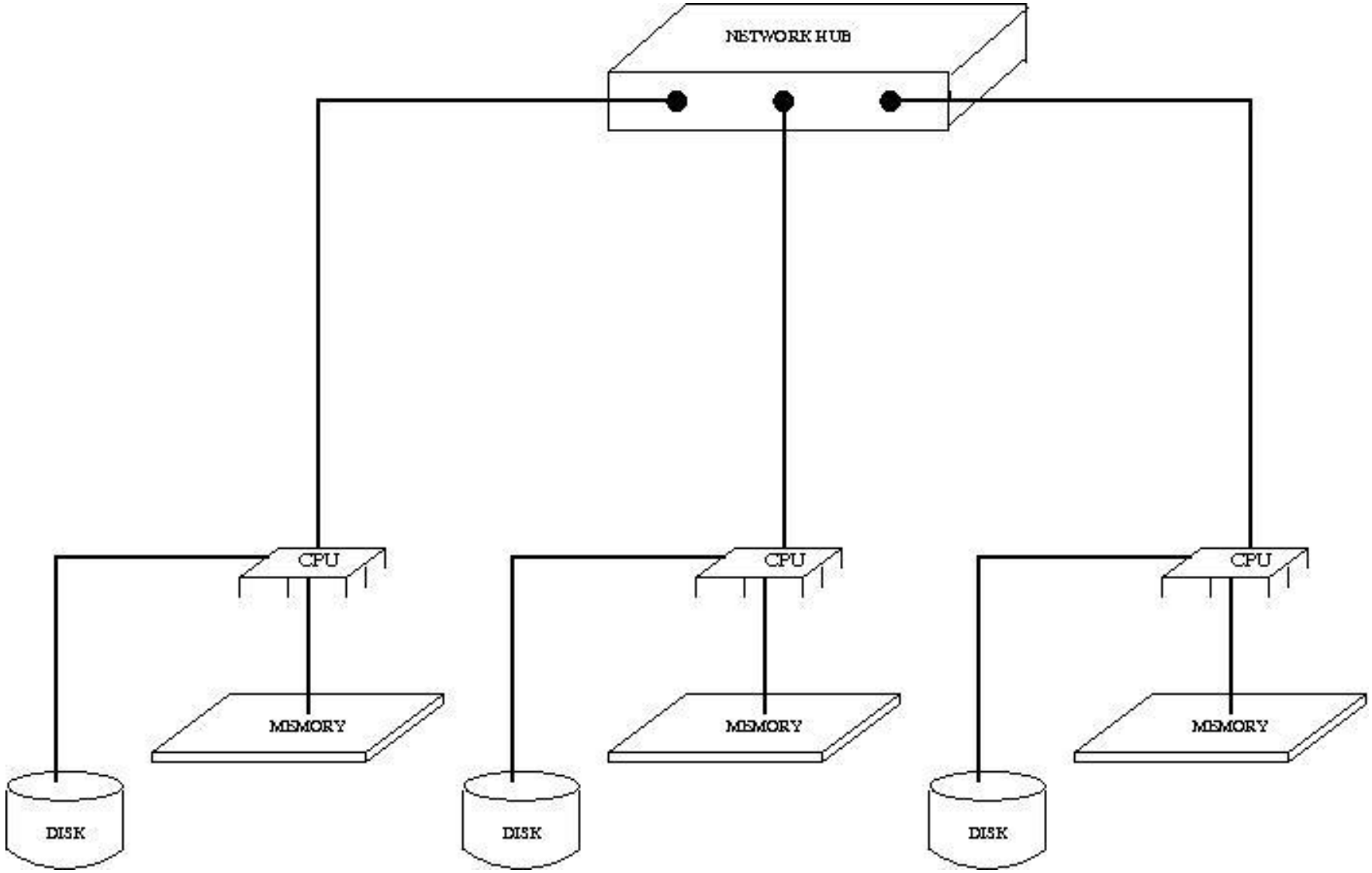
Software Support for Shared Memory

- Unix-like systems
  - POSIX shared memory – `shm_open()`, `shmget()`, `shmctl()`
  - `mmap()`

Distributed Memory Programming

- The problem size may not fit on a single machine
  - Graph analytics
  - Obvious step: Go distributed!

- Distributed computing model
  - Launch multiple processes on multiple systems
  - Processes carry out work
  - Processes may communicate through message passing
  - Processes coordinate either through message passing or synchronization



Wikipedia.

Challenges with Distributed Memory

Often, communication turns out to be the primary bottleneck

- How do you partition the data between different nodes?
- Network topology is very important for scalability

Since communication is explicit, therefore it **excludes** race conditions.

Be clear with uses!

- Parallel computing
  - Multiple tasks in a program cooperate to solve a problem efficiently
- Concurrent programming
  - Multiple tasks in a program can be in progress at the same time
- Distributed computing
  - A program needs to cooperate with other programs to solve a problem

Yonghong Yan.

POSIX Threads

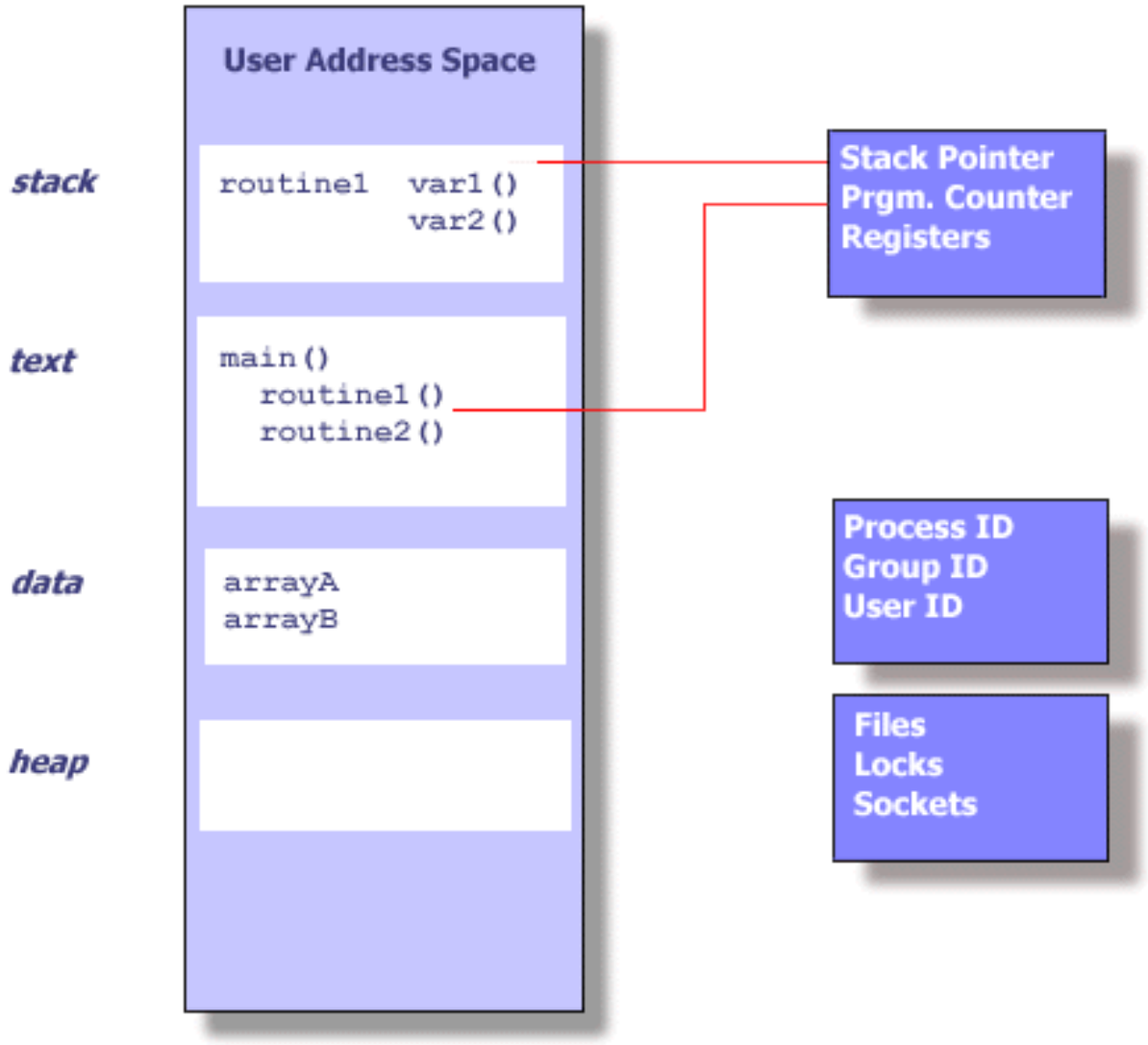
Multithreading with C/C++

C/C++ languages do not provide built-in support for threads

Several thread libraries have been proposed

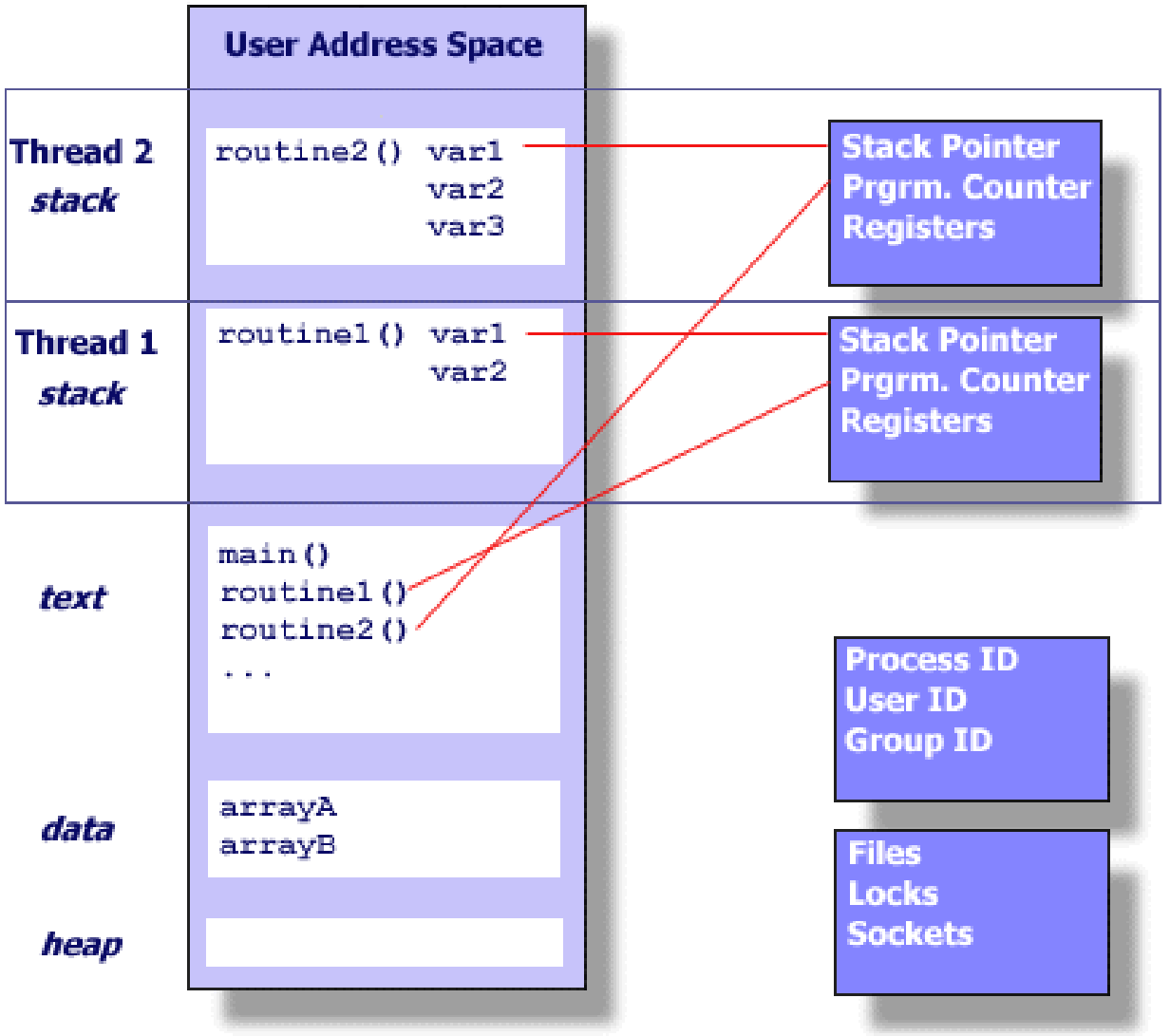
- Pthreads
- OpenMP
- Intel TBB

Unix Process



Blaise Barney, LLNL. POSIX Threads Programming.

Threads in Unix



Blaise Barney, LLNL. POSIX Threads Programming.

What are Threads?

- Software analog of cores
  - Each thread has its own PC, SP, registers, etc
  - All threads share the process heap and the global data structures
- Runtime system schedules threads to cores
  - If there are more threads than cores, the runtime will time-slice threads on to the cores

POSIX Threads (Pthreads)

- POSIX: Portable Operation System Interface for Unix
  - Standardized programming interface by IEEE POSIX 1003.1c for Unix-like systems
- **Pthreads**: POSIX threading interface
  - Provides system calls to create and manage threads
  - Contains ~100 subroutines

When to use Pthreads?

- Pthreads provide good performance on shared-memory single-node systems
  - Compare with MPI on a single node
- Ideal for shared-memory parallel programming
- HPC: # threads == # cores

Pthread routines

Call Prefix	Functional Group
pthread_	Thread management
pthread_attr_	Thread attributes objects
pthread_mutex	Mutexes
pthread_mutexattr_	Mutex attribute objects
pthread_cond_	Condition Variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Compile Pthread Programs

- GNU GCC
  - `gcc/g++ <options> <file_name(s)> -lpthread`

- Clang
  - `clang/clang++ <options> <file_name(s)> -lpthread`

Creating Threads

- Program begins execution with the **MAIN** thread

```
#include <pthread.h>


int pthread_create(pthread_t* thread_handle,
                   const pthread_attr_t* attribute,
                   void* (*thread_function) (void*),
                   void* arg);
```

Thread Creation Example

```
errcode = pthread_create(&tid, &attribute, &thread_function,
                         &fun_args);
```

- A pthread with handle "`tid`" is created
- Thread will execute the code defined in `thread_function` with optional arguments captured in `fun_args`
- `attribute` captures different thread features
  - Default values are used if you pass `NULL`
- `errcode` will be nonzero if thread creation fails

```
#include <cstdint>
#include <iostream>
#include <pthread.h>

#define NUM_THREADS 1

void *thr_func(void *thread_id) {
  uint32_t id = (intptr_t)thread_id;
  std::cout << "Hello World from Thread " << id << "\n";
  pthread_exit(NULL);
}
```

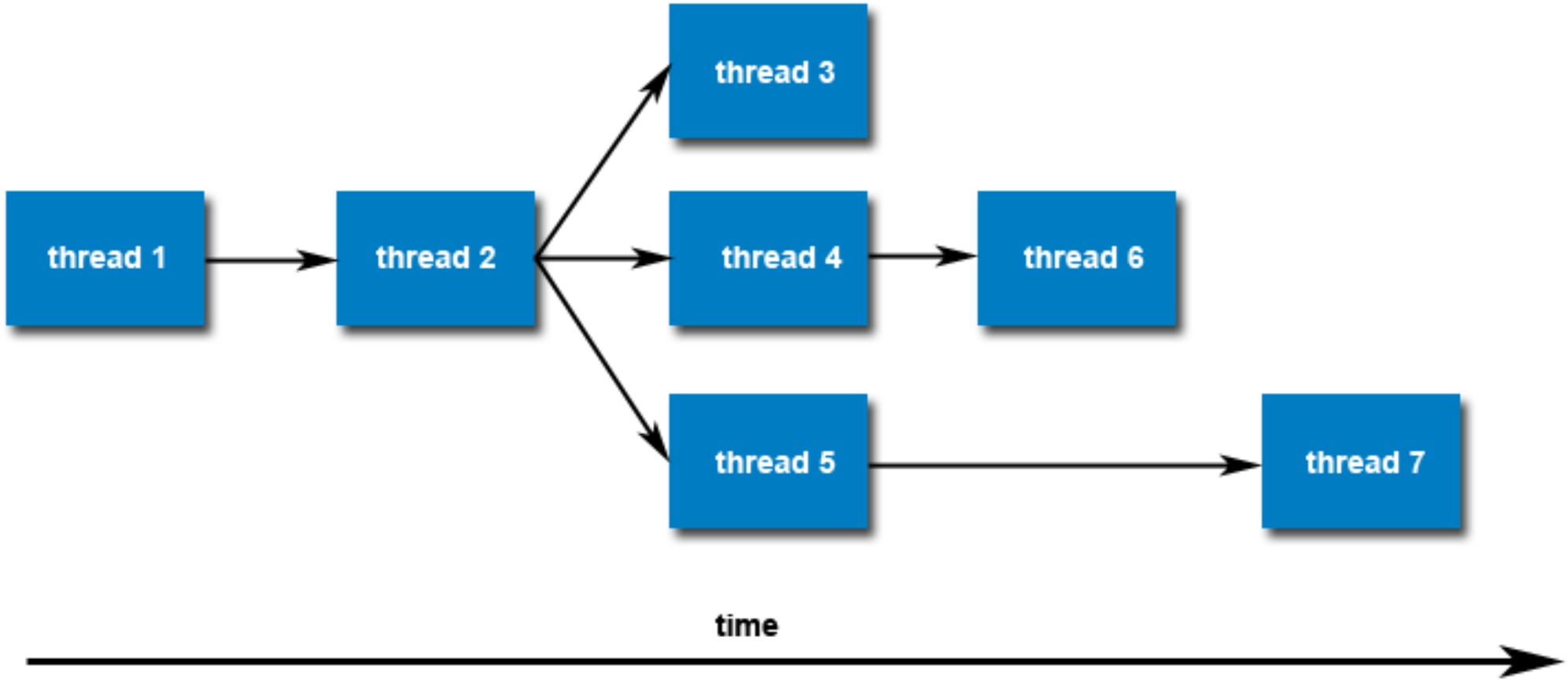
```
int main() {
  pthread_t threads[NUM_THREADS];
  int errcode;
  uint32_t id;
  for (id = 0; id < NUM_THREADS; id++) {
    std::cout << "In main: creating thread: " << id << "\n";
    errcode = pthread_create(&threads[id], NULL, thr_func, (void *)(intptr_t)id);
    if (errcode) {
      std::cout << "ERROR: return code from pthread_create() is " << errcode
<< "\n";
      exit(-1);
    }
  }

  pthread_exit(NULL);
}
```

```
#include <cstdint>
#include <iostream>
#include <pthread.h>

#define NU

void *thr
  uint32_
  std::cou
Thread " <
  pthread_
}
```

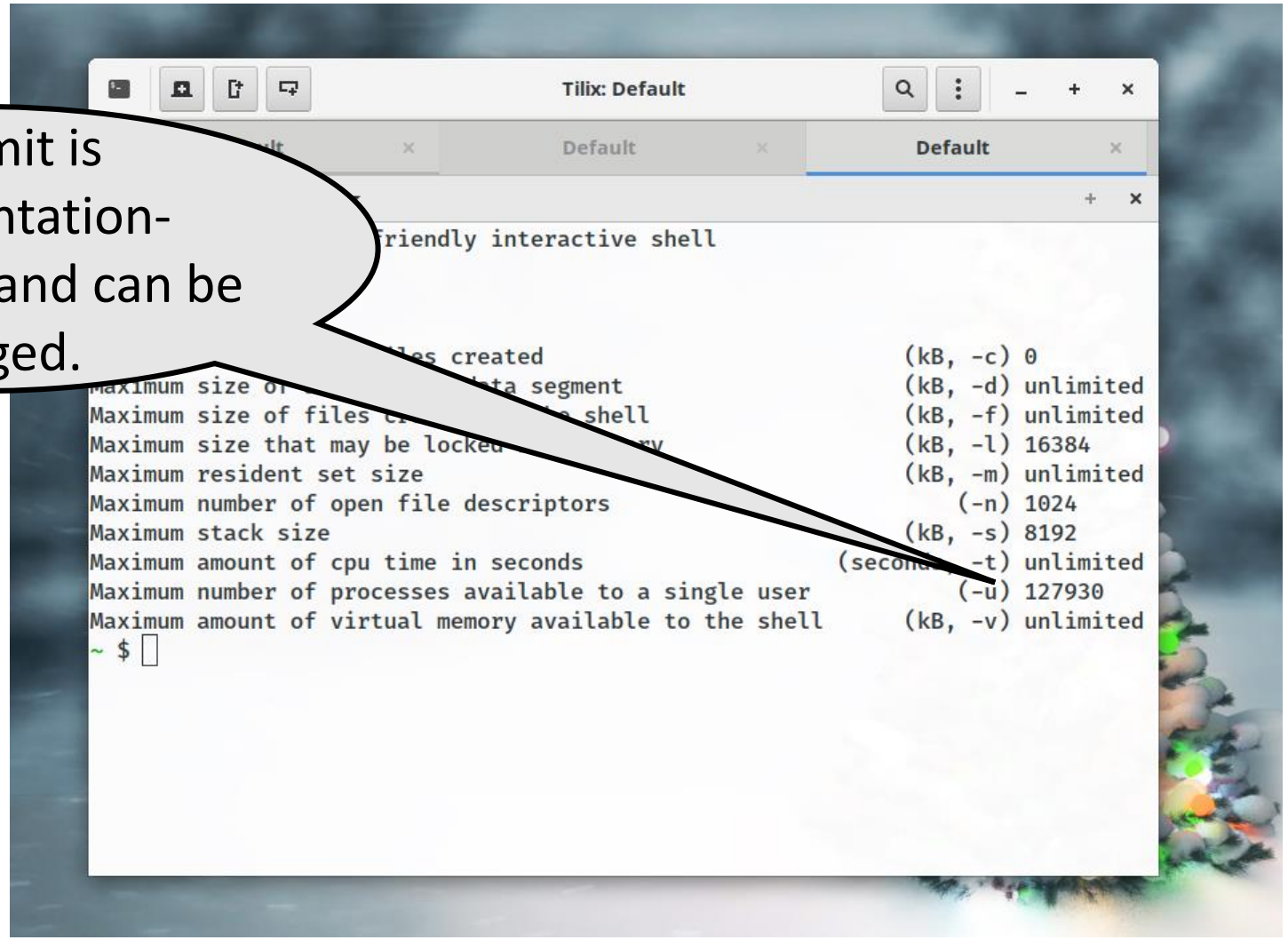
```
int main() {
  pthread_t threads[NUM_THREADS];
  int errcode;
  uint32_t id;
                                    " << id <<
                                    , (void
                                    n
  }

  pthread_exit(NULL);
}
```

```
1: fish  /home/swarnendu/iitk-workspace/c++-examples/src
~/i/c/src $ g++ pthread_helloworld.cpp -lpthread
~/i/c/src $ ./a.out
In main: creating thread: 0
Hello World from Thread 0
~/i/c/src $
```

No Implied Hierarchy Between Threads



Number of Pthreads



Terminating Threads

- A thread is terminated with

```
void pthread_exit(void* retval);
```

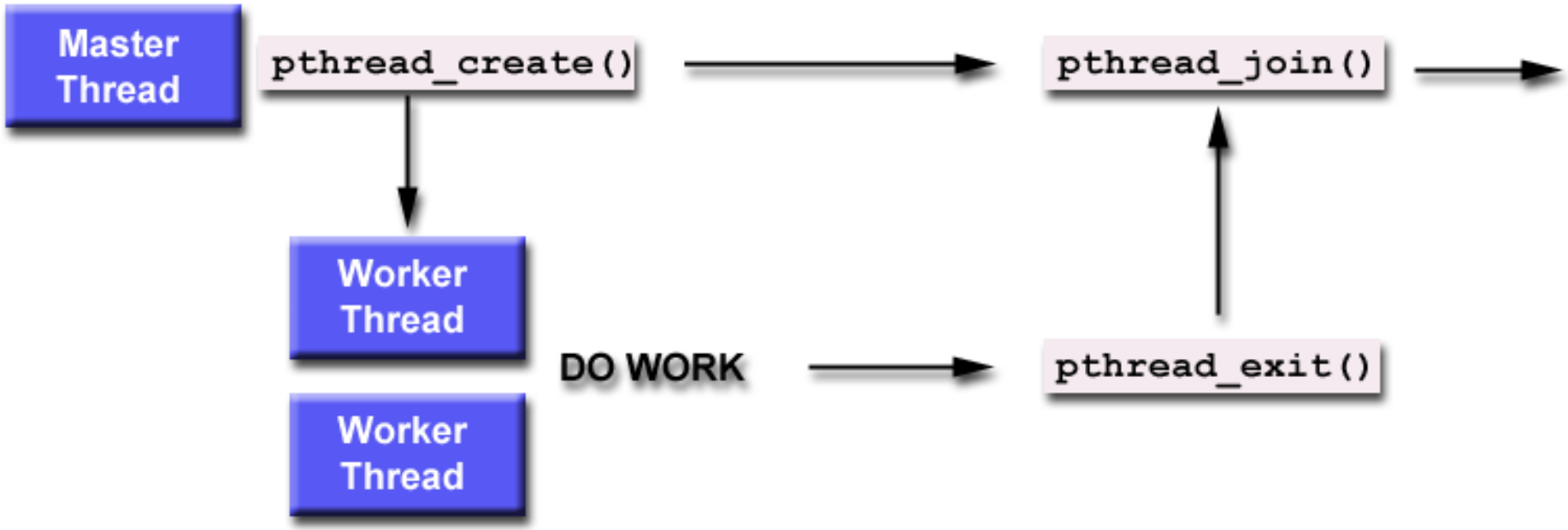
- Process-shared resources (e.g., mutexes, file descriptors) are not released
- Process terminates after the last thread terminates
  - Like calling `exit()`
  - Shared resources are released
- Child threads will continue to run if called from main thread

Other Ways to Terminate

- Thread completes executing `thr_func`
- Thread is canceled by another thread via `pthread_cancel()`
- Entire process is terminated by `exit()`
- If main thread finishes first without calling `pthread_exit()` explicitly

Joining Threads

```
int pthread_join(pthread_t thread, void ** value_ptr);
```



Subtle Issues to Keep in Mind

- Only threads that are created as “joinable” can be joined
  - If a thread is created as “detached”, it can never be joined
- A joining thread can match one `pthread_join()` call
  - It is a logical error to attempt multiple joins on the same thread
- If a thread requires joining, it is recommended to explicitly mark it as **joinable**.

Other Thread Management Routines

```
pthread_t pthread_self(void);
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

```
#define NUM_THREADS 10

uint32_t counter;

struct thr_args {
  uint16_t id;
};

void *thrBody(void *arguments) {
  struct thr_args *tmp =
static_cast<struct thr_args
*>(arguments);
  for (uint32_t i = 0; i < 1000; i++) {
    counter += 1;
  }
  pthread_exit(NULL);
}
```

```
int main() {
  int i = 0;
  int error;
  pthread_t tid[NUM_THREADS];
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  struct thr_args args[NUM_THREADS] = {0};

  while (i < NUM_THREADS) {
    args[i].id = i;
    error = pthread_create(&tid[i], &attr, thrBody,
args + i);
    i++;
  }
  pthread_attr_destroy(&attr);
  cout << "Value of counter: " << counter << "\n";
  // Join with child threads
  pthread_exit(NULL);
}
```

```
#define NUM_THREADS 10

uint32_t counter;

struct thr_args {
  uint16_t id;
};

void *thrBody(void *argum
  struct thr_args *tmp =
static_cast<struct thr_ar
*>(arguments);
  for (uint32_t i = 0; i
    counter += 1;
  }
  pthread_exit(NULL);
}
```

```
~/i/c/src $ g++ pthread_datarace.cpp -lpthread
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 9569
~/i/c/src $ ./a.out
Value of counter: 9218
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 9636
```

```
READS];

tr);
NUM_THREADS] = {0};

S) {

eate(&tid[i], &attr, thrBody,

&attr);
nter: " << counter << "\n";
reads
```

```
#define NUM_THREADS 10

uint32_t counter;

struct thr_args {
  uint16_t id;
};

void *thrBody(void *argum
  struct thr_args *tmp =
static_cast<struct thr_ar
*>(arguments);
  for (uint32_t i = 0; i
    counter += 1;
  }
  pthread_exit(NULL);
}
```

```
~/i/c/src $ g++ pthread_datarace.cpp -lpthread
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 9569
~/i/c/src $ ./a.out
Value of counter: 9218
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 9636
```

```
#define NUM_THREADS 10

uint32_t counter;

struct thr_args {
  uint16_t id;
};

void *thrBody(void *arg
  struct thr_ar
static_cast<s
*>(arguments);
  for (uint32_t i
    counter += 1;
  }
  pthread_exit(NULL);
}
```

```
~/i/c/src $ g++ pthread_datarace.cpp -lpthread
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 9569
~/i/c/src $ ./a.out
Value of counter: 9218
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 9636
```

Data race which results in an atomicity violation

Mutual Exclusion

Mutual exclusion (locks)

- Synchronize access to a shared data structure
- Cannot prevent bad behavior if other threads do not take locks

Checkout `pthread_mutex_` …

```
…
lock l =
alloc_init()
…
acq(l)
access data
rel(l)
…
acq(l)
access data
rel(l)
…
```

Creating Mutexes

```
int pthread_mutex_init(pthread_mutex_t *restrict
mutex, const pthread_mutexattr_t *restrict attr);
```

- Mutex variables must be initialized before use
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
  - `pthread_mutex_init()`

```
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

Locking and Unlocking Mutexes

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Types of Mutexes

- **NORMAL**
  - Attempt to relock a mutex by the same thread will deadlock
  - Attempt to unlock an unowned locked or unlocked mutex results in undefined behavior
- **ERRORCHECK**
  - Returns error if a thread tries to relock the same mutex
  - Attempt to unlock an unowned locked or unlocked mutex results in an error
- **RECURSIVE**
  - Allows the concept of reentrancy by maintaining a count
- **DEFAULT**
  - Wrong use results in undefined behavior

```
#define NUM_THREADS 10

uint32_t counter;
pthread_mutex_t count_mutex;

struct thr_args {
  uint16_t id;
};


void *thrBody(void *arguments) {
  pthread_mutex_lock(&count_mutex);
  for (uint32_t i = 0; i < 1000; i++) {
    counter += 1;
  }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

```
int main() {
  int i = 0;
  int error;
  pthread_t tid[NUM_THREADS];
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  struct thr_args args[NUM_THREADS] = {0};

  while (i < NUM_THREADS) {
    args[i].id = i;
    error = pthread_create(&tid[i], &attr, thrBody,
args + i);
    i++;
  }

  pthread_attr_destroy(&attr);
  cout << "Value of counter: " << counter << "\n";
  // Join with child threads
  pthread_exit(NULL);
}
```

```
#define NUM_THREADS 10

uint32_t counter;
pthread_mutex_t count_mutex;

struct thr_args {
  uint16_t id;
};

void *thrBody(void *argument
  pthread_mutex_lock(&count_
  for (uint32_t i = 0; i < 1
    counter += 1;
  }
  pthread_mutex_unlock(&cou
  pthread_exit(NULL);
}
```

```
Default
1: fish  /home/swarnendu/iitk-workspace/c++-examples/src
~/i/c/src $ g++ pthread_mutex.cpp -lpthread
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $
```

```
HREADS];
;
attr);
s[NUM_THREADS] = {0};

ADS) {

reate(&tid[i], &attr, thrBody,

y(&attr);
ounter: " << counter << "\n";
threads
```

POSIX Sempahores in Pthreads

Semaphores
- Generalize locks to allow “n” threads to access
- Useful if you have **> 1** resource units

```
#include <semaphore.h>

sem_init()
sem_wait()
sem_post()
```

```
gcc/g++ <options> <file_name(s)>
-lpthread -lrt
```

Synchronization in Pthreads

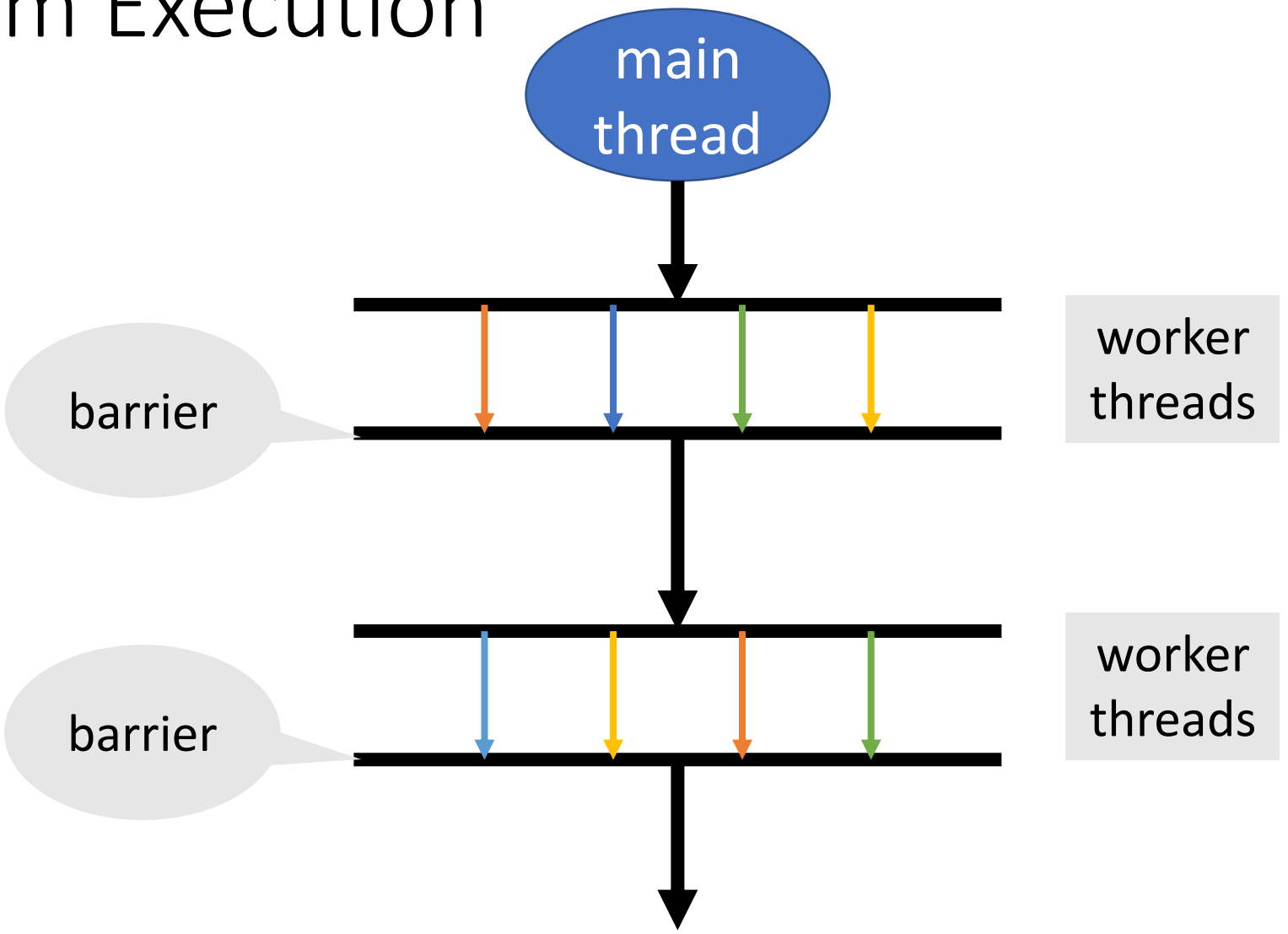
Barrier

- Form of global synchronization
- Commonly used on GPUs

```
…
dowork()
barrier
…
domorework()
barrier()
…
```

Checkout `pthread_barrier_` …

Phased Program Execution



Remember this Java Snippet?

```
Object X = null;
boolean done= false;
```

Thread T1

```
X = new Object();
done = true;
```

Thread T2

```
while (!done) {}
X.compute();
```

```
#define NUM_THREADS 2

volatile int i = 0;

void *thr1Body(void *arguments) {
  while (i == 0) {};
  cout << “Value of i has changed\n”;
  pthread_exit(NULL);
}


void *thr2Body(void *arguments) {
  sleep(1000);
  i = 42;
  pthread_exit(NULL);
}
```

```
int main() {
  pthread_t tid1, tid2;

  pthread_create(&tid1, NULL, thr1Body, NULL);
  pthread_create(&tid2, NULL, thr2Body, NULL);

  pthread_exit(NULL);
}
```

```
#define NUM_THREADS 2

volatile int i = 0;

void *thr1Body(void *arguments) {
  while (i == 0) {};
  cout << "Value of i [illegible] changed\n";
  pthread_exit(NULL);
}


void *thr2Body(void *argument
  sleep(1000);
  i = 42;
  pthread_exit(NULL);
}
```

```
int main() {
  pthread_t tid1, tid2;

  pthread_create(&tid1, NULL, thr1Body, NULL);
  pthread_create(&tid2, NULL, thr2Body, NULL);


  pthread_exit(NULL);
}
```

Busy waiting leads to wasted work

- Often used when we need to synchronize on the **data value**

```
#define NUM_THREADS 2

volatile int i = 0;

void *thr1Body(void *arguments) {
  while (i == 0) {};
  cout << "Valu
  pthread_exit
}

void *thr2Body(void *arguments) {
  sleep(1000);
  i = 42;
  pthread_exit(NULL);
}
```

```
int main() {
  pthread_t tid1, tid2;

  pthread_create(&tid1, NULL, thr1Body, NULL);
  pthread_create(&tid2, NULL, thr2Body, NULL);
```

Can you think of situations where busy waiting is actually advantageous?

Condition Variables

Signaling mechanism

- Always **used along with a mutex lock** which protects accesses to shared data

Slightly more involved usage

Checkout `pthread_cond_` …

Nuances of using Pthreads

- **Low-level abstraction**
- Pthreads scheduler may not be well-suited to manage large number of threads
  - Load balancing

- OpenMP is commonly used in scientific computing
  - Compiler extensions
  - Higher level of abstraction

- Other abstractions like Transactional Memory

References

- D. Sorin et al. – A Primer on Memory Consistency and Cache Coherence
- James Demmel and Katherine Yelick – CS 267: Shared Memory Programming: Threads and OpenMP
- Keshav Pingali – CS 377P: Programming Shared-memory Machines, UT Austin.
- Blaise Barney, LLNL. POSIX Threads Programming.