Concurrency and Parallelism
Vol.II

Tomás Gabriel

Version 1.0.0

Contents

2.5 Synchronizaiton - Competition and Cooperation

Data races: Occurs when two or more threads access a shared variable concurrently, and at least one of the accesses is a write.

Ordering violations: Ordering violations occur when the expected sequence of operations is disrupted due to concurrent execution. This can happen when operations that need to happen in a specific order (to ensure correctness) are executed out of order due to the parallel execution of threads or processes.

Synchronization is required when the progress of one or several threads depends on the behavior of other threads.

Two types of interaction require synchronization:

- **Cooperation**: Occurs when one thread can only progress after some event on another thread. This is the source of ordering violations.
- **Competition**: Occurs when threads compete to execute some statements (that access some shared resource) and only one thread at a time (or a bounded number of them) is allowed to execute them. This is the source of data races.

2.5.1 Cooperation

Synchronization Mechanisms

- **Barrier (or rendezvous)**: A set of control points, one per thread involved in the barrier, such that each thread is allowed to pass its control point only when all other threads have attained their own control points. From an operational point of view, each thread must stop until all other threads have arrived at their control point.
- **Condition variables**:
  - Waiting for a condition to be true;
  - Bound to a lock;
  - Used to implement monitors;
- **Futures**: Waiting for one-off events;

2.5.2 Competition

Critical Sections

The classical definition is critical section is “A piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread of execution.”

However, this definition can be misleading. It suggests that the code segment itself is the issue, when the core issue lies with the shared resource being accessed.

2.6 Atomicity

Race conditions lead to errors when sections of code are interleaved

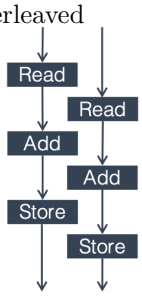


Fig. 1: Interleaved Execution

These errors can be prevented by ensuring code executes atomically

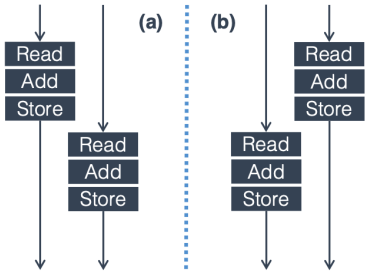


Fig. 2: Non-Interleaved (Atomic)

Instruction-level Atomicity

This is performed by a single ISA[1] instruction on a memory location address. Read the old value, calculate a new value, and write the new value to the location.

The **hardware** ensures that no other threads can access the location until the atomic operation is complete. Any other threads that access the location will typically be held in a queue until its turn. **All threads perform the atomic operation serially**.

Read-Modify-Write Operations

Most architectures:

- **test&set**: Reads the value stored in the memory location/register and writes '0' back;
- **reset**: Writes 1 into the memory location/register;
- **exchange** or **swap**: Atomically assigns a value v to a memory location/register and returns the one previous stored in such location;
- **compare&swap** / **compare&exchange**: Compares the content of a memory location/register against a specified value, and if they match, updates the content of that memory location to a new specified value.

On x86, the **lock** prefix makes an instruction atomic. When the **lock** prefix is used with an instruction, the CPU ensures that the instruction is executed atomically. This means that the entire operation completes uninterrupted, with no other processor or core able to access the involved memory location until the operation is finished. The lock prefix is only legal with certain instructions, mainly those that perform read-modify-write operations.

[1] Instruction Set Architecture

The compare&swap() primitive

Let 'X' be a shared register and *old* and *new* be two values. The primitive X.compare&swap(*old*, *new*) returns a **boolean** value and is defined by the following code, which is assumed to executed atomically:

```
X.compare&swap(old, new) is
    if (X = old) then X ← new; return(true)
                 else return (false)
    end if.
```

Note

Check out the documentation of both **C++** and **Java** in the Section B.2 of the Appendix.

Competition - Mechanisms to handle Critical Sections

Pessimistic: Assumes that data conflicts or inconsistencies are **likely** to occur when multiple threads execute the critical section. A strategy can be **Mutual exclusion** with locks, semaphores, monitors.

Optimistic: Assumes that data conflicts or inconsistencies are **not likely** to occur when multiple threads execute the critical section. Strategies may be:

- **Optimistic lock-based** solutions that reduce the size of the critical section to the minimum (studied in optimistic and lazy synchronization);
- **Lock-free** solutions;

2.7 Mutual Exclusion

2.7.1 Implementing Mutual Exclusion

How do we provide the application with an appropriate abstraction level?

Designing: An entry algorithm (also called entry protocol) and an exit algorithm (also called exit protocol) that when used to delimit a critical section ensure that the critical section code is executed by at most one thread at a time.

Operations:

- `acquire_mutex();`
- `release_mutex();`

2.7.2 Concurrent executions of acquire_mutex()

Only one of the invocations terminates and the corresponding thread p is called **the winner**. The other invocations stay **on hold** being **the losers**

A well-formed thread executes the entry and exit protocols appropriately.

```
procedure protected_code(in) is
    acquire_mutex(); r ← cs_code(in); release_mutex(); return(r)
end procedure.
```

Mutual Exclusion Problem: Definition

The mutual exclusion problem consists in implementing the operations "acquire_mutex()" and "release_mutex()" in such a way that the following properties are always satisfied:

- **Mutual exclusion**: At most one thread at a time executes the critical section code;
- **Starvation-freedom**: For any thread p, each invocation of acquire_mutex() by p eventually terminates

2.7.3 Properties

Safety Safety properties state that nothing bad happens. They can usually be expressed as invariants[1]. The invariant here is the mutual exclusion property, which states that at most one thread at a time can execute the critical section code.

> Note
>
> Note that a solution in which no thread is ever allowed to execute the critical section code would trivially satisfy the safety property.

- **Example** → Mutual exclusion: No two concurrent threads are executing cs_code(in) (used in the example code above) simultaneously.

Liveness Liveness properties state that something good eventually happens.

- **Example** → Starvation freedom: Whatever the time T, if before T one or several threads have invoked the operation acquire_mutex() and none of them has terminated its invocation at time T, then there is a time $T' > T$ at which a thread that has invoked acquire_mutex() terminates its invocation. This means that a thread that wants to enter the critical section can be bypassed an arbitrary but finite number of times by other thread.

[1] Condition or property that holds true across the execution of a program

Deadlock Two or more threads are unable to proceed because each is waiting for the other to release a resource or complete a task, resulting in a standstill where none of the them can progress.

- Starvation-freedom implies deadlock-freedom: If a thread requests access to the critical section it will eventually get permission and for it to get permission, the system cannot be deadlocked.
- Deadlock-freedom does not imply starvation-freedom: The system is operating. There is a thread willing to get access to the critical section that is always overcome by another later thread.

The Mutual Exclusion Lock (MutEx) Object

- A lock is a shared object with two methods: `LOCK.acquire_lock()` and `LOCK.release_lock()`;
- A lock can be in one of two sates: **free** or **locked**;
- It is initialized to the value free;
- Its behavior is defined by a sequential specification: From an external observer point of view, all the `acquire_lock()` and `release_lock()` invocations appear as if they have been invoked one after the other;
- It solves the mutual exclusion problem: Solving the mutual exclusion problem $\iff$ implementing a lock object

MuTex Implementation

→ test&set()/reset()

```
operation acquire_mutex() is
    repeat r ← X.test&set() until (r = 1) end repeat;
end operation.

operation release_mutex() is
    X.reset(); return
end operation.
```

And so we have:
✓ Mutual Exclusion;
✓ Deadlock Freedom;
✗ No Starvation;
✗ Fairness;

Note

Implementations with *swap*() and *compare&swap*() are available in the slides (Lec.7 slide 27 and 28) but they don't provide anything this solution doesn't.

→ Deadlock and starvation-free algorithm

```
operation acquire_mutex(i) is
    (1) FLAG[i] ← up;
    (2) wait (TURN = i) ∨ (FLAG[TURN] = down);
    (3) LOCK.aquire_lock(i);
    (4) return()
end operation.

operation release_mutex(i) is
    (5) FLAG[i] ← down
    (6) if (FLAG[TURN] = down) then TURN ← (TURN mod
   n) + 1 end if;
    (7) LOCK.release_lock(i);
    (8) return()
end operation.
```

And so we have:
✓ Mutual Exclusion;
✓ Deadlock Freedom;
✓ No Starvation;
✗ Fairness;

→ Fairness with fetch&add()

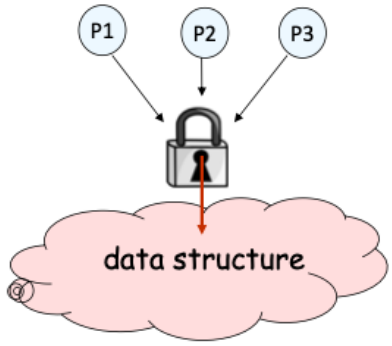
```
operation acquire_mutex() is
    myturn ← TICKET.fetch&add();
    repeat no-op until(myturn = NEXT) end repeat;
    return()
end operation.

operation release_mutex() is
    NEXT ← NEXT + 1; return()
end operation.
```

And so we have:
✓ Mutual Exclusion;
✓ Deadlock Freedom;
✓ No Starvation;
✓ Fairness;

2.8 Lock-Free Synchronization

2.8.1 Concurrent Data Structures

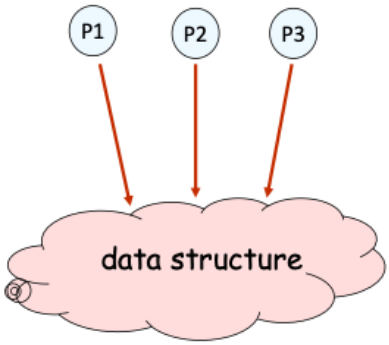
Using Locks Typically simplifies the design but MAY introduce delays and coordination overhead, affecting performance.



This solution has us using a simple programming model but can lead to:

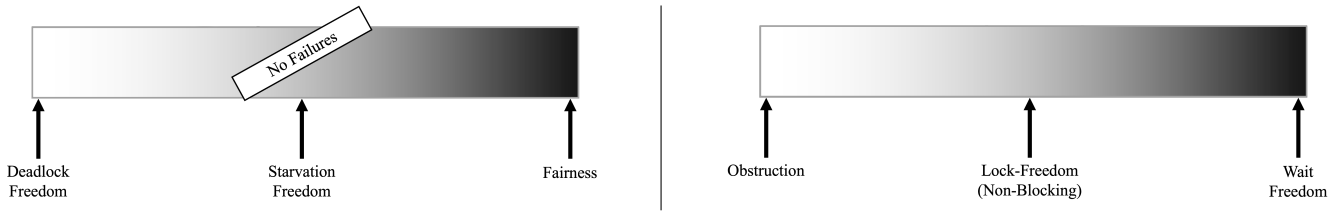
- False conflicts;
- Fault-free solutions only;
- Sequential bottleneck

Without Locks Aims to improve system responsiveness and throughput under high concurrency but often at the cost of increased complexity in algorithm design and potential challenges in ensuring correct behavior across all scenarios.



This solution is resilient to failures but can lead to:

- Often being (very very) complex;
- Memory consuming;
- Sometimes, weak progress conditions;



2.8.2 Progress Conditions

- **Obstruction-freedom** At any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) will complete its operation in a bounded number of steps. All lock-free algorithms are obstruction-free;
- **Non-Blocking** When the program threads are run sufficiently long, at least one of the threads makes progress (for some sensible definition of progress);
- **Wait-freedom** Every operation has a bound on the number of steps the algorithm will take before the operation completes: starvation-freedom for all threads in the system.

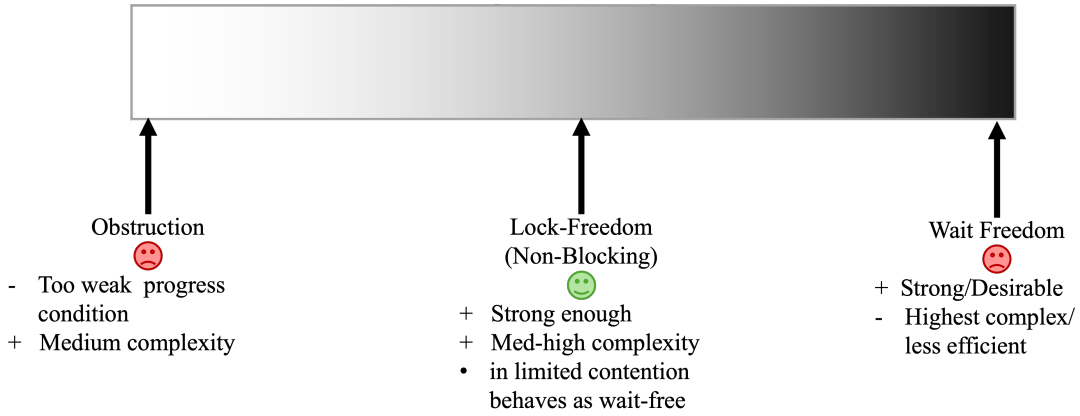


Fig. 3: Lock-free Data Structures

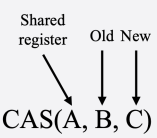
2.8.3 Lock-free Lists

After the lazy list, the next logical step is to **eliminate locking entirely**

- `contains()` wait-free
- `add()` lock-free
- `remove()` lock-free

This guarantees minimal progress in any execution, i.e., some thread will always complete a method call, even if others halt at malicious times.

For this we first try to use `compareAndSet()`:




```
if A == B then A → () C; return(true)
          else return(false)
```

Supported by Intel, AMD, Arm, etc.

But using CAS has a problem. Let's see an example of trying to execute a `remove(`c`)`

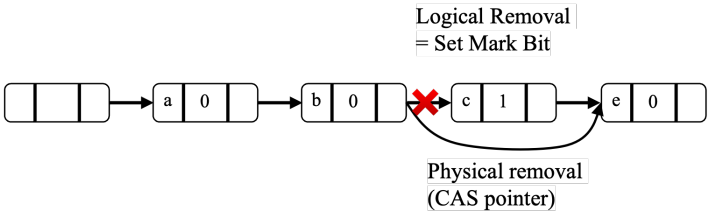


Fig. 4: Using CAS to verify pointer

But this is not enough, because if we add lets say a d the following might happen:

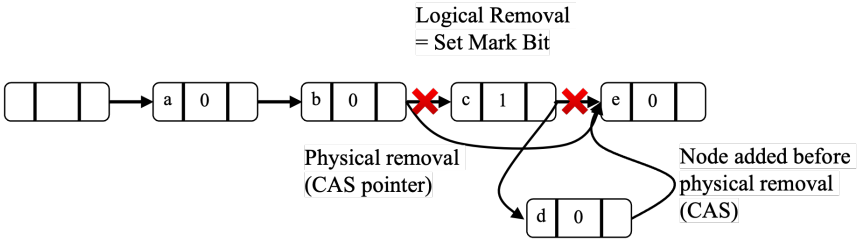


Fig. 5: Using CAS to verify pointer is not enough!

As we can see in the image above, d was not added to the list. So we must prevent manipulation of the node's pointer. The solution is to combine the bit and the pointer in one:

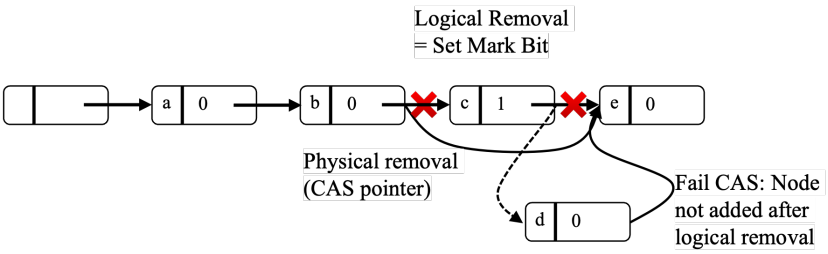


Fig. 6: Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

So the solution is to use **AtomicMarkableReference**, meaning:

Atomically

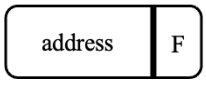
- Swing reference and
- Uptade Flag

Remove in two steps

- Set mark bit in next field
- Redirect predecessor's pointer with a CAS

Marking a Node

AtomicMarkableReference class → `java.util.concurrent.atomic package`



Extracting Reference & Mark

```
public Object get(boolean[] marked);
```

- `Object` → Returns reference
- `boolean` → Returns mark at array index 0

Extracting Reference Only

```
public boolean isMarked();
```

- `boolean` → Value of mark

Changing State

```
public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);
```

- `Object expectedRef` → "If this is the current reference..."
- `Object updateRef` → "...then change to this new reference "
- `boolean expectedMark` → "And if this is the current mark ..."
- `boolean updateMark` → "...then change to this new mark"

Traversing the List What do we you do when we find a "logically deleted" node in our path? We finish the job:

- CAS the predecessor's next field;
- Proceed (repeat as needed).

> Note
>
> I recommend you check the code in the section A.1.2 in the appendix containing the implementation using the window class.

2.8.4 To Lock or Not to Lock

Locking vs. Non-blocking: Extremist views on both sides.

The answer Nobler to compromise, combine locking and non- blocking, for example, the lazy list combines blocking `add()` and `remove()` and a wait-free `contains()`.

> Note
>
> It is important to remember that blocking/non-blocking is a property of a method

2.9 Memory Consistency

2.9.1 What is Correct Behavior for a Parallel Memory Hierarchy

Side-effects of writes are only observable when reads occur. So we will focus on the values returned by reads.

Intuitive answer Reading a location should return the latest value written (by any thread). What does "latest" mean exactly?

Within a thread, it can be defined by program order. But across threads, this becomes a reasonable question?

- The most recent write in physical time? Hopefully not, because there is no way that the hardware can pull that off. if it takes > 10 cycles to communicate between processors, there is no way that a processor a can know what a processor b did 2 clock ticks ago.
- Most recent based upon something else?

2.9.2 Refining Intuition

Thread 0

```
// write evens to X
for (i=0; i<N; i
    +=2){
     X = i;
     ...
}
```

Thread 1

```
// write evens to X
for (j=1; j<N; j
    +=2){
     X = j;
     ...
}
```

Thread 2

```
A = X;
...
B = X;
...
C = X;
...
```

What would be some clearly illegal combinations of (A,B,C)?

- (4,8,1)
- (9,12,3)
- (7,19,31)

What can we generalize from this?

- Writes from any particular thread must be consistent with program order. In this example, observed even numbers must be increasing.
- Across threads: writes must be consistent with a valid interleaving of threads (not physical time since the programmer can't rely upon that).

Visualizing Intuition

Each thread proceeds in program order. Memory accesses interleaved (one at a time) to a single-ported memory (the rate of progress of each thread is unpredictable).

Recall "reading a location should return the latest value written (by any thread)". "Latest" means consistent with some interleaving that matches this model.

2.9.3 Memory Consistency Model

Cache Coherence Guarantees the consistency and synchronization of data stored in different caches within a multiprocessor or multi-core system (Deals with: do all loads and stores to a given memory location behave correctly?)

Memory Consistency Model Defines the allowed orderings of multiple threads on a multiprocessor (Deals with: do all loads and stores, even to separate memory locations, behave correctly?)

This is a complicated thing to understand. The main issue is that *loads* and *stores* are very expensive, even on a uni-processor, being easily able to take 10 to 100 cycles. Programmers **intuitively** expect that the processor atomically performs one instruction at a time in program order, but in reality, if the processor actually operated this way, it would be painfully slow. Instead the processor **aggressively reorders instructions** to hide memory latency.

And so the outcome is that within a given thread, the processor preserves the program order illusion from the perspective of other threads, all bets are off!

Hiding Memory Latency This is a very important step for performance. The idea is to overlap memory accesses with other accesses and computation. Hiding memory latency is simple in uni-processors since you need only to add a **write buffer** (more on this discussed furter). It is important to note that this affects correctness in multiprocessors!

Fig. 7: Hiding Memory Latency

Hiding Memory Latency of Reads This is possible by using **"Out of Order" Pipelining**. This concept was spoken about in the Computer Architecture course. When an instruction is stuck, perhaps there are subsequent instructions that can be executed, and this may include branch prediction, when in the presence of conditional branches.

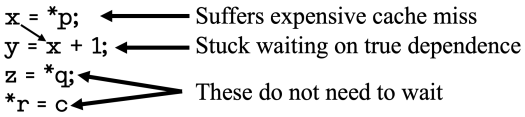


Fig. 8: Out of Order Pipelining

This implies that memory accesses may be performed out-of-order!

How Out-Of-Order Pipelining Works Fetch and graduate instructions are executed in-order, but issue instructions are executed out-of-order.

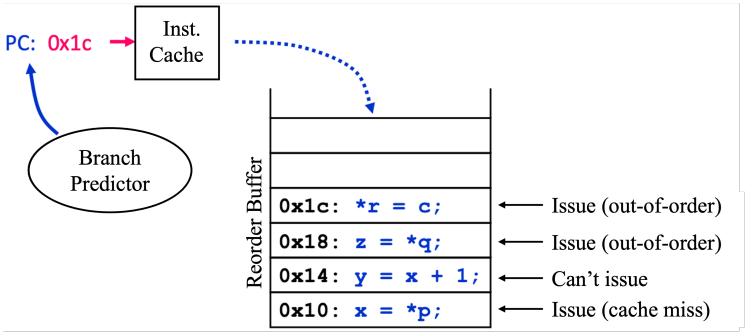
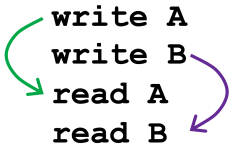
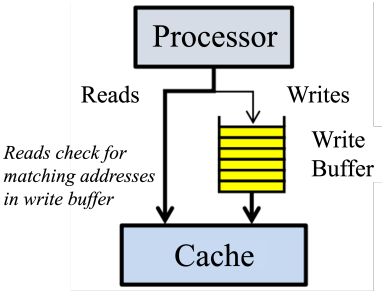


Fig. 9

Intra-thread dependences are preserved, but memory accesses get reordered.





2.9.4 Uniprocessor Memory Model

Memory model specifies ordering constraints among accesses

Uniprocessor model Memory accesses are atomic and in program order. Like this it is not necessary to maintain sequential order for correctness.

- **hardware**: buffering, pipelining
- **compiler**: register allocation, code motion

This is simple for programmers and allows for high performance.

2.9.5 In Parallel Machines (with a Shared Address Space)

Order between accesses to different locations becomes important, for example

(Initially A and Ready = 0)

P1

```
A = 1;
Ready = 1;
```

P2

```
while (Ready != 1);
... = A;  // Should be 1
```

Fig. 10

How Unsafe Reordering Can Happen

Distribution of memory resources accesses issued in order may be observed out of order

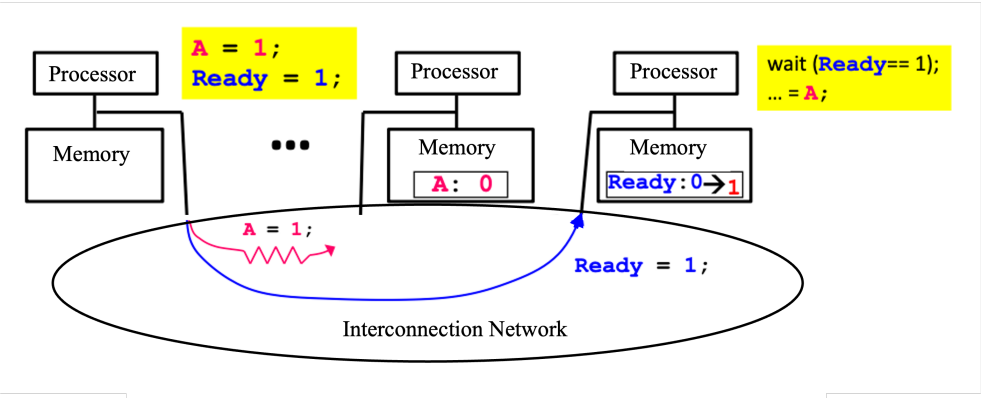


Fig. 11

Caches complicate things more due to multiple copies of the same location

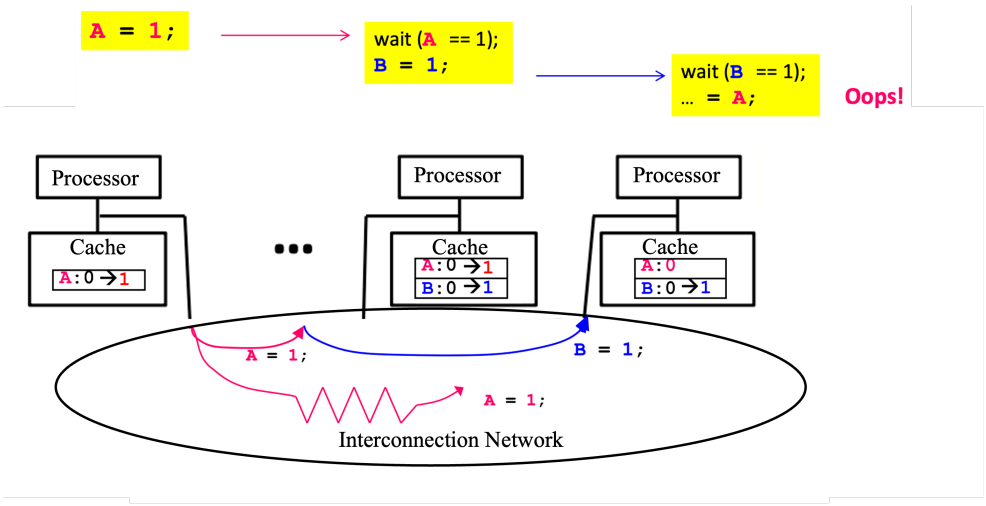


Fig. 12

2.9.6 Our Intuitive Model: "Sequential Consistency" (SC)

Accesses of each processor are in program order. All accesses appear in sequential order. Any order implicitly assumed by the programmer is maintained

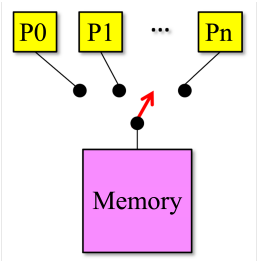


Fig. 13

Example with Sequential Consistency Simple Synchronization:

P0		P1	
A = 1	(a)		
Ready = 1	(b)	x = Ready	(c)
		y = A	(d)

Fig. 14

All locations are initialized to 0.

Possible outcomes for (x,y): (0,0) - (0,1) - (1,1)

(x,y) = (1,0) is not a possible outcome (i.e Ready = 1, A = 0).

- We know $a \to b$ and $c \to d$ by program order
- $b \to c$ implies that $a \to d$
- $y = 0$ implies $d \to a$ which leads to a contradiction

The important thing to note is that real hardware will do this!

Implementing Sequential Consistency One Approach to implementing sequential consistency is to implement cache coherence, a.k.a. writes to the same location are observed in same order by all processors. For each processor, delay the start of memory access until the previous one completes, and so each processor has only one outstanding memory access at a time.

But what does it mean for a memory access to complete?

Memory Reads: A read completes when its return value is bound.

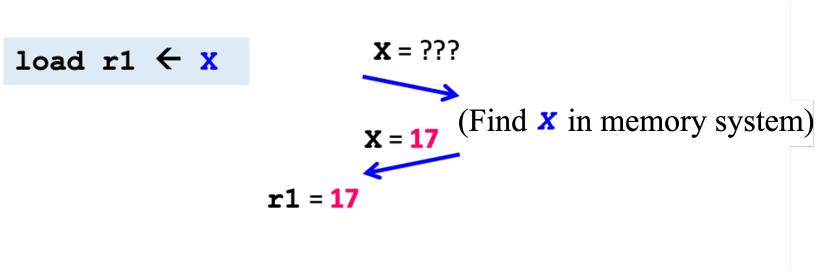


Fig. 15

Memory Writes: A write completes when the new value is "visible" to other processors

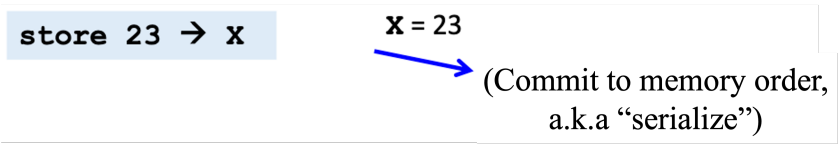


Fig. 16

But what does "visible" mean? It does NOT mean that other processors have necessarily seen the value yet. It means that the new value is committed to the hypothetical serializable order **(HSO)**. A later read of X in the HSO will see either this value or a later one (for simplicity, assume that writes occur atomically).

Alternatives to Sequential Consistency

Sequential consistency maintains order between shared accesses in each processor.

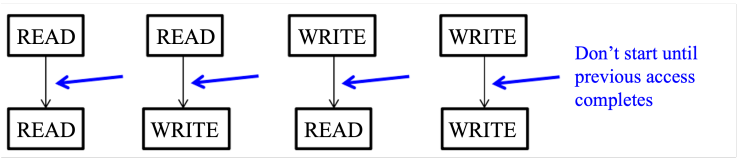
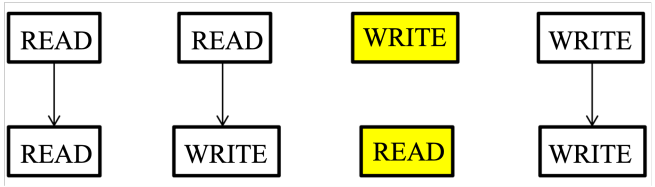
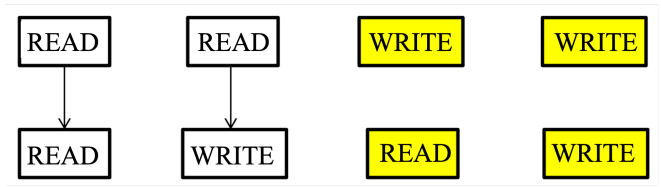


Fig. 17

But this severely restricts common hardware and compiler optimizations. So alternatives can be relaxing constraints on the memory order:



Total Store Ordering (TSO) (Similar to Intel)



Partial Store Ordering (PSO)

Performance Impact of TSO vs SC TSO Can use a write buffer and the write latency is effectively hidden.

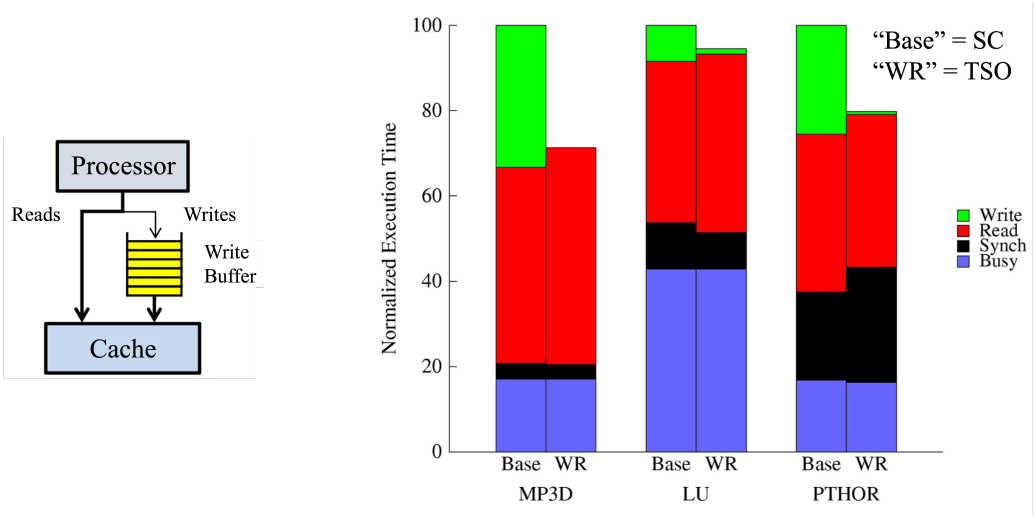


Fig. 18: TSO vs SC

And this makes us ask **can programs live with weaker memory orders?**

For this, the program needs to be **correct**, a.k.a. achieve the same results as sequential consistency, and most programs don't require strict ordering (for all of the time) to achiveve "correctness"

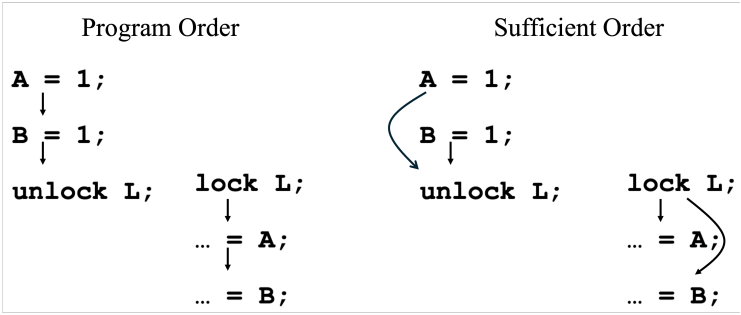
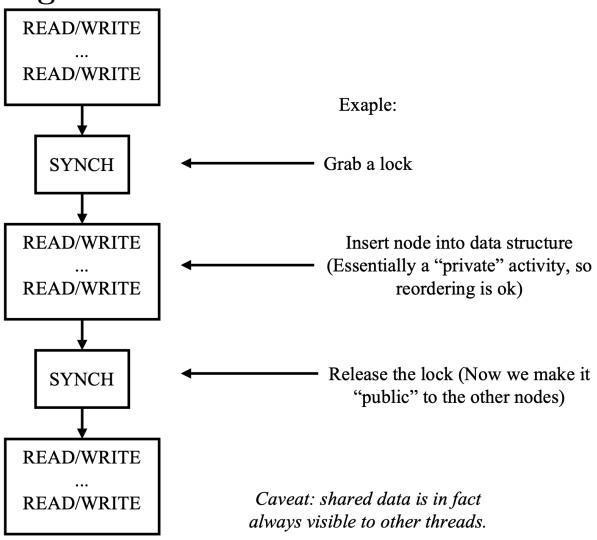


Fig. 19: Program Order vs Sufficient Order

Optimizations for Synchronized Programs

The intuition is that may parallel programs have mixtures of "private" and "public" parts. The "private" parts must be protected by synchronization (e.g. locks)



And so we exploit information about synchronization. Properly synchronized programs should yield the same result as on an SC machine.

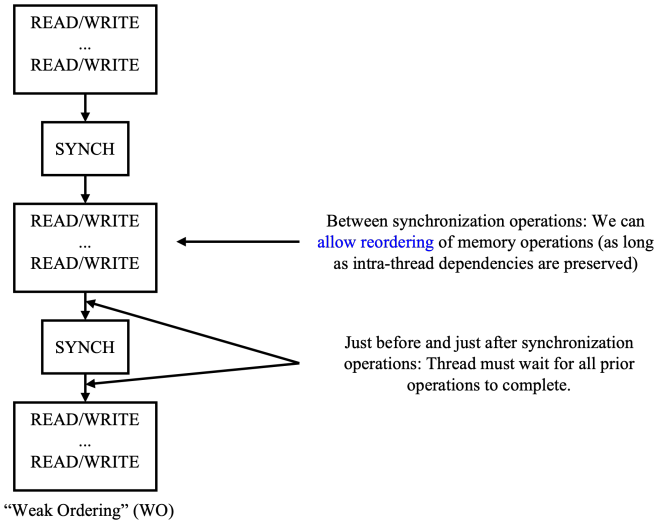


Fig. 20

Intel's Memory Fence Operation (MFENCE) MFENCE operantion enforces the ordering seen above:

- Does not begin until all prior reads & writes from that thread have completed.
- No subsequent read or write from that thread can start until after it finishes

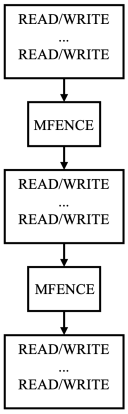


Fig. 21

A MFENCE operation does **not** push values out to other threads (it's not a magic "make every thread up-to-date" operation). It simply stalls the thread that performs the MFENCE until write buffer empty.

Revisiting the previous example in Fig 14, we want to use **MFENCE** operations to fix this:

P0	**P1**
`A = 1;`	
`MFENCE;`	`X = Ready;`
`Ready = 1;`	`MFENCE;`
	`y = A;`

Fig. 22

"Release Consistency"

- Lock operation: only gains ("**acquires**") permission to access data;
- Unlock operation: only gives away ("**releases**") permission to access data;

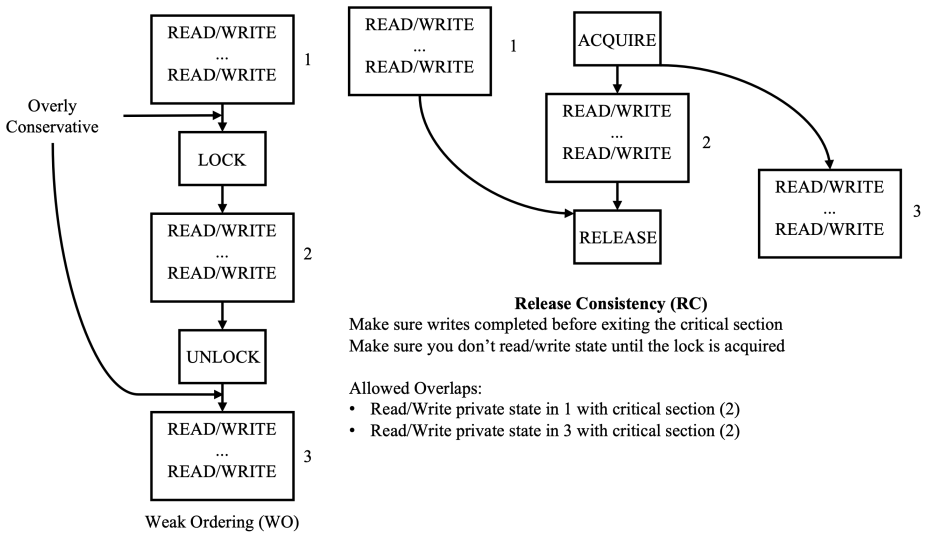


Fig. 23

In addition to **MFENCE**, Intel also supports two other fence operations:

- **LFENCE:** Serializes only with respect to load operations (not stores!)
- **SFENCE:** Serializes only with respect to store operations (not loads!)

So going back to the Fig. 22 we can use these to make it even better:

P0

```
A = 1;
SFENCE;
Ready = 1;
```

P1

```
X = Ready;
LFENCE;
y = A;
```

Fig. 24

2.10 Concurrency Errors

Appendix

A. Lectures' Bulk Code

A.1 Synchronization Code

A.1.1 Fine-Grained Synchronization

Remove Method

```java
public boolean remove(Item item) { int key = item.hashCode();
    Node pred, curr;
    try {
        pred = this.head;
        pred.lock();
        curr = pred.next;
        curr.lock();
        // Travesing the list:
        while (curr.key <= key) { // Search key range
            if (item == curr.item) {
                // If the item is found remove node
                pred.next = curr.next;
                return true;
            }
            // Unlock the predecessor. Only 1 node is locked
            pred.unlock();
            pred = curr;    // Demote current
            curr = curr.next;
            curr.lock();    // Find and lock the new current
        }
        return false; // Element not present
    } finally { // Makes sure the locks are released
        curr.unlock();
        pred.unlock();
    }
}
```

Add Method

```
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        // Initialize pointers to traverse the list
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            // Lock the nodes
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        // Try the operation and either succeed or fail
        try {
            // If the locked nodes are still accessible,
            // that means they are still in the list
            if (validate(pred, curr)){
                if (curr.key == key) {
                    // If item already in list, fail
                    return false;
                } else {
                    // If item not present, create new node insert
                    // into the list, and succeed
                    Node node = new Node(item);
                    node.next = curr;
                    pred.next = node;
                    return true;
                }
            }
        // Always unlock (with both success and failure)
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

A.1.2 Lock-Free Synchronization

The Window Inner Class

```
class Window {
    // A container for pred and current values
    public Node pred;
    public Node curr;
    Window(Node pred, Node curr) {
        this.pred = pred; this.curr = curr;
    }
}
```

The find method

```
// Find returns window:
// pred: node with the largest key < "key"
// curr: node with the least key >= "key"
Window window = find(head, key);
// Extract pred and curr
Node pred = window.pred;
Node curr = window.curr;
```

Remove(Lock-Free)

```
public boolean remove(T item) {
    boolean snip;
    int key = item.hashCode();
    while (true) { // Keep Trying
        Window window = find(head, key);     // Find Neighbours
        Node pred = window.pred, curr = window.curr;
        if (curr.key != key) { // It's not there
            return false;
        } else {
            Node succ = curr.next.getReference();
            snip = curr.next.attemptMark(succ, true); // Try to mark node
     as deleted
            // If it doesn't work, just retry. If it does the job, is
    essentially done.
            if (!snip) continue;
            // Try to advance reference
            // (if we don't succeed, someone else did or will do).
            pred.next.compareAndSet(curr, succ, false, false);
            return(true);
        }
    }
}
```

Add(Lock-Free)

```
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key == key) { // Item already there.
            return false;
        } else {
            // Create new node
            Node node = new Node(item);
            node.next = new AtomicMarkableRef(curr, false);
            // Install new node, else retry loop
            if (pred.next.compareAndSet(curr, node, false, false)){
                return true;
            }
        }
    }
}
```

Contains(Wait-Free)

```
public boolean contains(T item) {
    boolean[] marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
        curr = curr.next.getReference();
    // Only diff is that we get and check marked
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
}
```

Contains(Wait-Free)

```
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; boolean snip;
    // If list changes while traversed, start over Lock-Free
    // because we start over only if someone else makes
    retry: while (true) {
        // Start looking from head
        pred = head;
        curr = pred.next.getReference();
        while (true) { // Move down the list
            succ = curr.next.get(marked); //Get ref to successor and
    current deleted bit
            // Try to remove deleted nodes in path.
            while (marked[0]) {
                // Try to snip out node
                snip = pred.next.compareAndSet(curr,
                                succ, false, false);
                if (!snip) continue retry; // if predecessor's next field
     changed must retry whole
                // Otherwise move on to check if next node deleted
                curr = succ;
                succ = curr.next.get(marked);
            }
            // If curr key is greater or equal, return pred and curr
            if (curr.key >= key)
                return new Window(pred, curr);
            // Otherwise advance window and loop again
            pred = curr;
            curr = succ;
        }
    }
}
```

B. Cheat Sheets

B.1 OpenMP Cheat Sheet

`#pragma omp parallel for collapse(n)`
: Collapses nested loops into a single loop to be parallelized, where `n` specifies the number of nested loops.

`#pragma omp parallel for schedule(static, chunk_size)`
: Divides the loop iterations into chunks of size `chunk_size` and assigns each chunk to a thread in a round-robin fashion. If `chunk_size` is not specified, the iterations are divided into chunks that are as evenly sized as possible. Best for loops where each iteration takes roughly the same amount of time to execute.

`#pragma omp parallel for schedule(dynamic, chunk_size)`
: Initially assigns chunks of size `chunk_size` to threads. When a thread finishes its chunk, it is dynamically assigned a new chunk until no chunks remain. If `chunk_size` is not specified, it defaults to 1. Suitable for loops with iterations that have varying execution times. It helps in load balancing by assigning work to threads that finish their tasks early.

`#pragma omp parallel for schedule(guided, chunk_size)`
: Similar to dynamic scheduling, but the size of the chunks decreases over time. The size of the next chunk is proportional to the number of unassigned iterations divided by the number of threads, but it will not be smaller than `chunk_size`. If `chunk_size` is not specified, it defaults to a small chunk size determined by the implementation. Useful for loops where later iterations are quicker to execute or when you want to balance the load dynamically with larger initial chunks.

`#pragma omp parallel for schedule(auto)`
: The decision about scheduling is delegated to the compiler and/or runtime system. It chooses the scheduling based on the loop and system characteristics. When you prefer the compiler/runtime to make scheduling decisions, potentially using insights you may not have.

`#pragma omp parallel for schedule(runtime)`
: The scheduling policy and chunk size are determined by the OMP_SCHEDULE environment variable, which can be set to any of the above scheduling strategies and a chunk size. Offers flexibility to experiment with different scheduling strategies without recompiling. Suitable for fine-tuning performance based on the execution environment.

`#pragma omp parallel`
: This directive is used to fork additional threads to carry out the work enclosed in the block in parallel. It creates a parallel region wherein each thread executes the code inside the block independently. This directive is fundamental in OpenMP for initiating parallelism. It's typically the first directive you use when parallelizing a serial program.

`#pragma omp single`
: This directive specifies that the enclosed code block should be executed by only one thread (the first one that reaches the directive). Other threads will skip the block and wait at an implicit barrier at the end of the single region unless a `nowait` clause is specified. Useful for tasks that should only be done once and do not need to be repeated by each thread, like initialization tasks or I/O operations that are not thread-safe.

`#pragma omp task`
: This directive creates a task that can be executed asynchronously with respect to the thread that creates it. The task is queued for execution as per the availability of the threads in the team. This is key in defining fine-grained parallelism, particularly for irregular or recursive algorithms where workloads are dynamic and not evenly distributed.

`#pragma omp taskwait`
: This directive creates a wait point where a thread will not proceed until all tasks created by this thread in the task region are complete. Ensures that all tasks that were generated before this directive are completed before proceeding, which is crucial for synchronizing tasks, for example, when subsequent computation depends on the results of these tasks.

`#pragma omp parallel for reduction((operation: variable_list))`
: The reduction clause ensures that each thread has a private copy of the variables specified in `variable_list`, performs the specified operation on each private copy during the loop, and then combines all the private copies into a single global value using the same operation after the loop finishes. 'operation': This is a predefined reduction operator such as '+', '*', 'max', 'min', '&', '|', '&&', '||', etc. Very useful in loop parallelism where the loop iterations are independent but result in an update to the same output variables, typical in scenarios like accumulating sums or computing mathematical vectors.

Example:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < SIZE; i++) {
    sum += array[i];
}
```

`#pragma omp simd`
: This directive indicates that the loop should be vectorized using SIMD (Single Instruction, Multiple Data) instructions if possible. This can significantly speed up the loop execution by utilizing the vector processing capabilities of modern CPUs. Enhances performance by allowing multiple iterations of the loop to be executed simultaneously using SIMD instructions. It is beneficial for loops with operations that can be vectorized, such as basic arithmetic or logical operations on arrays.

B.2 Atomic Programming

B.2.1 At Programming Language Level – C++

atomic	Atomic class template and specializations for bool, integral, floating-point, (since C++20) and pointer types (class template)
atomic_ref	Provides atomic operations on non-atomic objects (class template)
atomic_flag	The lock-free boolean atomic type (class)
atomic_signed_lock_free	A signed integral atomic type that is lock-free and for which waiting/notifying is most efficient (typedef)
atomic_unsigned_lock_free	An unsigned integral atomic type that is lock-free and for which waiting/notifying is most efficient (typedef)

B.2.2 At Programming Language Level – Java

Class	Description
AtomicBoolean	A boolean value that may be updated atomically.
AtomicInteger	An int value that may be updated atomically.
AtomicIntegerArray	An int array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
AtomicLong	A long value that may be updated atomically.
AtomicLongArray	A long array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.
AtomicMarkableReference<V>	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.
AtomicStampedReference<V>	An AtomicStampedReference maintains an object reference along with an integer ”stamp”, that can be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running double value updated using a supplied function.
DoubleAdder	One or more variables that together maintain an initially zero double sum.
LongAccumulator	One or more variables that together maintain a running long value updated using a supplied function.
LongAdder	One or more variables that together maintain an initially zero long sum.