Introduction to Parallel & Distributed Programming

Lec 10—Memory Consistency (Causal, Processor), Synchronisation 1 — Flush & Atomic

Subodh Sharma | Feb 2, 2026

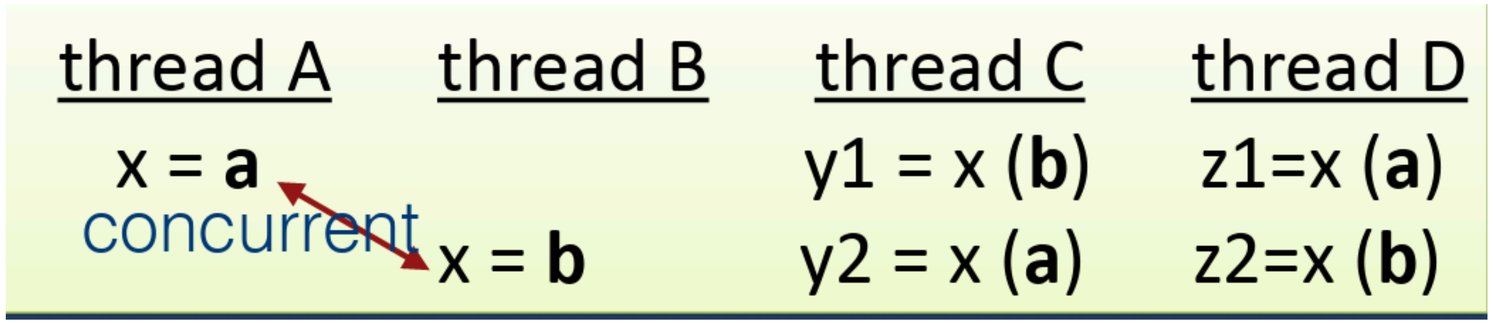
Causal Consistency

- **Rules of Happens-Before relation ($\rightarrow$):**
  - **Thread order:** If in a thread, $a < b$, then $a \rightarrow b$
  - **Reads-from:** If a read operation b reads the value from a write operation a then $a \rightarrow b$
  - **Transitivity**

Causal Consistency

- **Rules of causality guarantee:**
  - **Causal Writes:** If process P writes to x, say w_1, and then some process observes x and then writes to y, say w_2, then $w_1 \rightarrow w_2$
    - That is: every process must observe the same order of writes
  - Rules of HB hold.
- **What is allowed?**

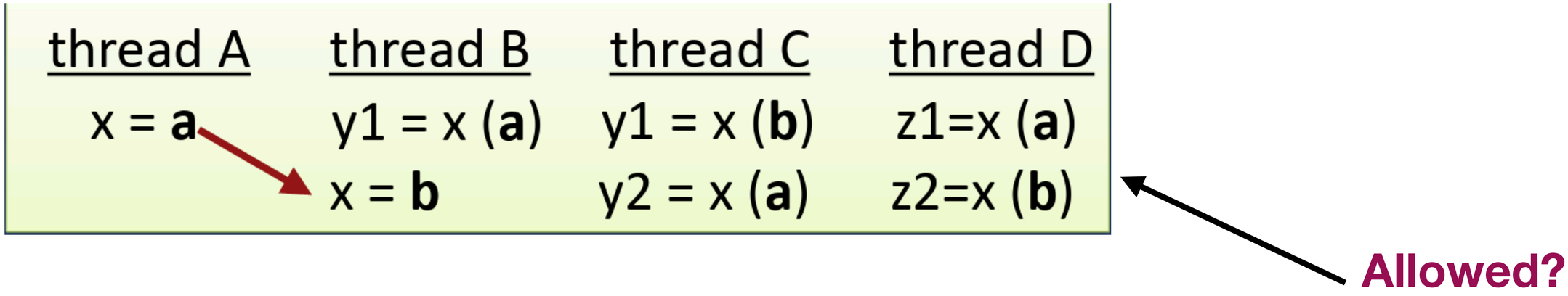
Causal Consistency



Allowed?

- **To justify Thread C's behaviour:** $W(x,b) \rightarrow R(x,b) \rightarrow W(x,a) \rightarrow R(x,a)$
- **To justify Thread D's behaviour:** $W(x,a) \rightarrow R(x,a) \rightarrow W(x,b) \rightarrow R(x,b)$
- **Since the two writes are concurrent, different threads can see them take place in different orders**
- **CAUSAL CONSISTENT**

Causal Consistency



- **To justify Thread C's behaviour:** $W(x,b) \rightarrow R(x,b) \rightarrow W(x,a) \rightarrow R(x,a)$
- **To justify Thread D's behaviour:** $W(x,a) \rightarrow R(x,a) \rightarrow W(x,b) \rightarrow R(x,b)$
- **But note that** $W(x,a) \rightarrow W(x,b)$**; but thread C violates this order**
- **NOT CAUSAL CONSISTENT**

Processor Consistency

- **Rules:**
  - **Writes by a single processor are observed in the order they were issued**
  - **Writes from different processors to the same location must be seen in the same order**
- **Weaker than Causal Consistency - WHY?**

T_1	T_2	T_3
$W(x,1)$	$R(x,1)$	$R(y,1)$
	$W(y,1)$	$R(x,?)$

Processor Consistency

T_1	T_2	T_3
$W(x,1)$	$R(x,1)$	$R(y,1)$
	$W(y,1)$	$R(x, v)$

- **Under PC:** Even though $R(y,1)$, it is not obligated that $R(x, v)$ where $v = 1$
  - Which means v = 0/1
- Under Causal Consistency: $W(x,1) \rightarrow R(x,1) \rightarrow W(y,1)$ …
  - Which means in $R(x, v)$, $v = 1$

Processor Consistency

T_1	T_2	T_3
$W(x,1)$	$R(x,2)$	$R(y,3)$
$W(x,2)$	$W(y,3)$	$R(x,1)$

- **The execution is Processor Consistent**

Processor Consistency

T_1	T_2
$W(x,1)$	$R(y,3)$
$W(x,2)$	$R(x,1)$
$W(y,3)$	

- **The execution is NOT Processor Consistent**

Synchronisation

Types of Synchronisation Tools

- **Memory fences** (eg: `# pragma omp flush`, h/w memory fences like `mfence`)
- **Atomic Operations:** event should happen uninterrupted
  - **Test & set, Fetch & add, Compare & swap**
- **Critical sections, Lock, Mutexes:** Events should **NOT** happen together
- **Barriers:** Events should happen together
- **Wait, Condition variables:** event A should happen before event B

Properties of Synchronisation

- Safety, Liveness
- Blocking
- Starvation-free, Deadlock-free, Lock-free, Wait-free

	Not lock-based Independent of Scheduler	Lock-based Depends on Scheduler
Everyone Progresses	Wait Free	Starvation Free
Someone Progresses	Lock Free	Deadlock Free

The Flush Operation

- Flush directive performs two primary actions:
  - It forces the thread's temporary view of the variables to be written back to memory
  - It forces the thread to invalidate its local copy and reload vars from the memory
- Flush doesn't provide atomicity or mutual exclusion
  - It only ensures consistent visibility

The Flush Operation

Thread A	Thread B
flagA = 1;	flagB = 1;
#pragma omp flush	#pragma omp flush
if (flagB == 0) {	if (flagA == 0) {
shared ++; ← mutual exclusion →	shared++;
}	}
flagA = 0;	flagB = 0;
#pragma omp flush	#pragma omp flush

Atomic Operations

Test & Set

- **Test-and-Set (TAS):** Atomically reads a location and sets it to 1 and returns the old value
  - **Semantics:**

```
bool old_value = *location;
*location = true;
return old_value;
```

  - Can be used to implement Locks
  - **Limitations?**

Atomic Operations

Fetch-and-Add

- **FAA -** Atomically adds a value to a memory location and returns the old value
- **Signature: T fetch_add(T* location, T increment)**
- **Semantics:**

```
T old_value = *location;
*location = old_value + increment;
return old_value;
```

- **Limitations:** No conditional update, Limited to addition, cache line contention, overflows-underflows