COL380

Introduction to Parallel & Distributed Programming

- Write is causally ordered after all earlier reads/writes in its thread
    - ➡ write may depends on the current complete ‘state’
- Read is causally ordered after its causative write
- Causality is transitive
- ∃ sequential order of causally related operations consistent with every thread’s view
    - ➡ Non-related writes may be seen in different order by different threads

Causal Consistency

- Write is causally ordered after all earlier reads/writes in its thread
  - ➡ write may depends on the current complete ‘state’
- Read is causally ordered after its causative write
- Causality is transitive
- ∃ sequential order of causally related operations consistent with every thread’s view
  - ➡ Non-related writes may be seen in different order by different threads

Causally Consistent

thread A	thread B	thread C	thread D
x = **a**		y1 = x (**b**)	z1=x (**a**)
concurrent	x = **b**	y2 = x (**a**)	z2=x (**b**)

Causal Consistency

- Write is causally ordered after all earlier reads/writes in its thread
    - ➡ write may depends on the current complete ‘state’
- Read is causally ordered after its causative write
- Causality is transitive
- ∃ sequential order of causally related operations consistent with every thread’s view
    - ➡ Non-related writes may be seen in different order by different threads

Causally Inonsistent

thread A	thread B	thread C	thread D
x = **a**	y1 = x (**a**)	y1 = x (**b**)	z1=x (**a**)
	x = **b**	y2 = x (**a**)	z2=x (**b**)

Processor Consistency

- All threads see all writes by each thread in the order of that thread
  - ➡ all instances of write(**x**) are seen by each thread in the same order
  - ➡ No need to consistently order writes to different variables by different threads
- Easy to implement
  - ➡ Two or more writes from a single source must remain in order, as in a pipeline
  - ➡ All writes are through to the memory

Processor Consistency

- All threads see all writes by each thread in the order of that thread
  - ➡ all instances of write(**x**) are seen by each thread ir
  - **FIFO consistency** relaxes this constraint
  - ➡ No need to consistently order writes to different variables by different threads
- Easy to implement
  - ➡ Two or more writes from a single source must remain in order, as in a pipeline
  - ➡ All writes are through to the memory

- All threads see all writes by each thread in the order of that thread
  - ➡ all instances of write(**x**) are seen by each thread in the same order
  - ➡ No need to consistently order writes to different variables by different threads
- Easy to implement
  - ➡ Two or more writes from a single source must remain in order, as in a pipeline
  - ➡ All writes are through to the memory

FIFO consistency is also known as **PRAM consistency**

Consistency Summary

Model	Description
Strict	Global time based atomic ordering of *all* shared accesses
Sequential	*All* threads see all shared accesses in the same order consistent with program order -- no centralized ordering
Causal	All threads see causally-related shared accesses in the same order
Processor	All threads see writes from each other in the order they were made. Writes to a variable must be seen in the same order by all threads
Weak	Special synchronization based reordering -- shared data consistent only after synchronization

Subodh Kumar

```
#pragma omp atomic  read/write/update/capture
    x++;
```

- Light-weight critical section
- Only for some basic expressions, e.g,
  - x binop= expr   (no mutual exclusion on expr evaluation)
  - x++
  - --x
  - v = x++

Atomic Directive

#pragma omp atomic read/write/update/capture

x++;

- Light-weight critical section
- Only for some bas
  - ➡ x binop= expr (nc
  - ➡ x++
  - ➡ --x
  - ➡ v = x++

Thread 0	Thread 1

// Produce data
data = 42;
 	
// Set flag to signal Thread 1
#pragma omp atomic write
    flag = 1;
	// Busy-wait until flag is signalled
	#pragma omp atomic read
	    myflag = flag
	while (myflag != 1) {
	    #pragma omp atomic read
	    myflag = flag
	}
	// Consume data
	printf(data=%d\n", data);

#pragma omp taskwait $\boxed{\text{depend(in:x)}}$

- Wait (suspend) for all children tasks
  - ➡ depend ⇒ only wait for some precedent tasks
- Also see:
  - ➡ #pragma omp taskgroup

Critical Section

- A block of code
- Criticality context
  - ➡ wrt other block(s)

```
#pragma omp critical (a_name)
{
    mutually_excluded_code();
}
```

Serialized

```
#pragma omp critical (a_name)
{
    mutually_excluded_code_also();
}
```

Barrier

- A group of entities
- Wait for all
    - ➡ Any post-barrier computation implies completion of pre-barrier computation in each thread of the group

```
ParallelInput();
#pragma omp barrier
ParallelProcess();
#pragma omp barrier
ParallelOutput();
```

Lock

- Object: lock
- Actions: Lock and Unlock

```
omp_lock_t *lockA;
omp_init_lock (lockA);
…
omp_destroy_lock (lockA);
```

Critical Section

```
OperateA(object *A)
{
    omp_set_lock(lockA);
    Operate_Exclusively(A)
    omp_unset_lock (lockA);
}
```

Lock

See:
```
int omp_test_lock (omp_lock_t *);
```

```
omp_lock_t *lockA;
omp_init_lock (lockA);
…
omp_destroy_lock (lockA);
```

- Object: lock
- Actions: Lock and Unlock

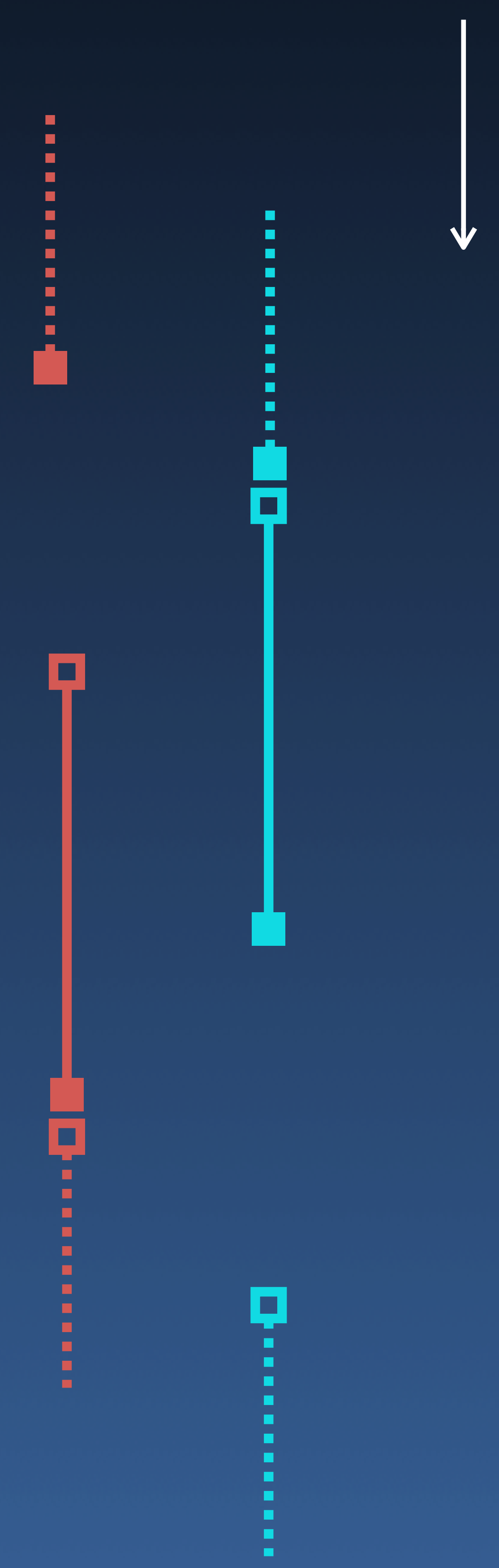
Critical Section

```
OperateA(object *A)
{
    omp_set_lock(lockA);
    Operate_Exclusively(A)
    omp_unset_lock (lockA);
}
```

Synchronization

- Do Operation X at time T
- Do we need a precise notion of time to make progress?
  - ➡ Always increasing
  - ➡ Shared view or individual view?
- Basic synchronization
  - ➡ Two (or a set of) events should happen together
  - ➡ Any two (from a set of) events should NOT happen together
  - ★ Event A should happen after event B

Stop and Go



- Define a partial order
  - ➡ Causality: A ➔ B ⇒ Time(A) < Time(B)
  - ➡ Inverse is not required to be true

- Define a partial order
  - ➡ Causality: $A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)$
  - $\boxed{\text{Same as: } \text{Time}(A) < \text{Time}(B) \Rightarrow B \not\rightarrow A}$
  - ➡ Inverse is not required to be true

Causal Ordering

- Define a partial order
    - ➡ Causality: $A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)$ $\boxed{\text{Same as: Time}(A) < \text{Time}(B) \Rightarrow B \not\rightarrow A}$
    - ➡ Inverse is not required to be true $\boxed{\text{Means they can are concurrent}}$

Causal Ordering

- Define a partial order
  - ➡ Causality: $A \rightarrow B \Rightarrow Time(A) < Time(B)$ $\boxed{\text{Same as: } Time(A) < Time(B) \Rightarrow B \not\rightarrow A}$
  - ➡ Inverse is not required to be true $\boxed{\text{Means they can are concurrent}}$

$\boxed{\text{Strong causality: } A \rightarrow B \Rightarrow Time(A) < Time(B) \text{ \&\& } Time(A) < Time(B) \Rightarrow A \rightarrow B}$

Causal Ordering

- Define a partial order
  - ➡ Causality: $A \rightarrow B \Rightarrow Time(A) < Time(B)$ $\boxed{\text{Same as: } Time(A) < Time(B) \Rightarrow B \not\rightarrow A}$
  - ➡ Inverse is not required to be true $\boxed{\text{Means they can are concurrent}}$

$\boxed{\text{Strong causality: } A \rightarrow B \Rightarrow Time(A) < Time(B) \ \&\& \ Time(A) < Time(B) \Rightarrow A \rightarrow B}$

- Clocks at least must support partial ordering of events
  - ➡ Can construct total ordering (e.g., by using Process-ID to break tie)
  - ➡ Possible to build "counters" that can support total order (strong causality)