COL380

Introduction to
Parallel & Distributed Programming

Atomic Directive

```
#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();
}
```

Subodh Kumar

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

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

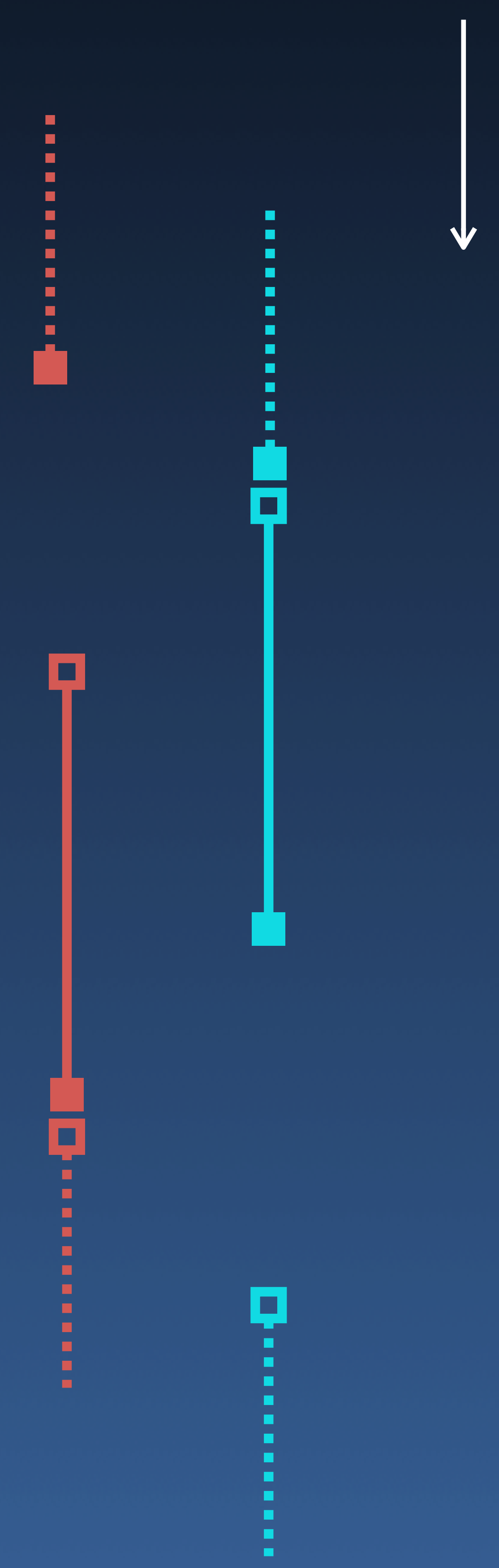
Critical Section

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

Subodh Kumar

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



Causal Ordering

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

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

- 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)

Logical-Clock
[Lamport’s Timestamp algorithm]

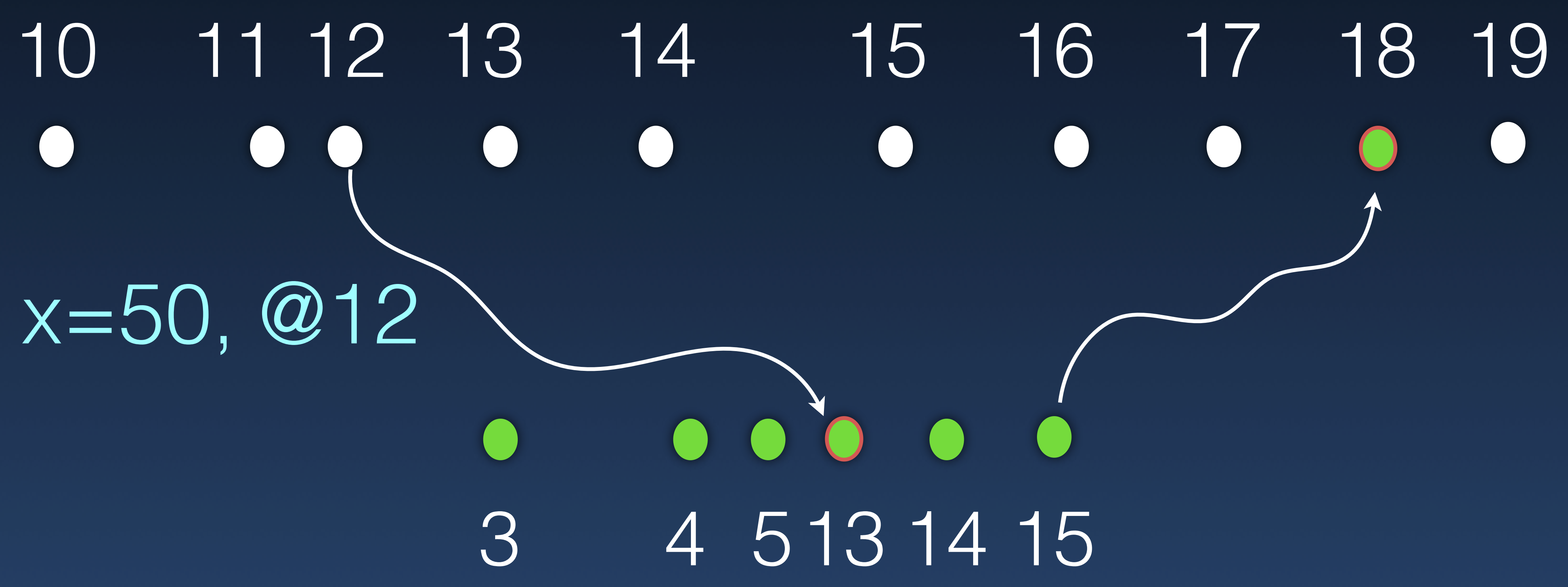
- Each entity (process) maintains a counter
  - ➡ increments every ‘event,’ at its own pace
- Interaction between entities is through messages
  - ➡ Data + counter
- On message receipt:
  - ➡ If recipient counter < received counter
    - Increase local counter to received counter
    - Receive is an event, so increment by one

$\boxed{A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)}$

e.g., Lamport Clock

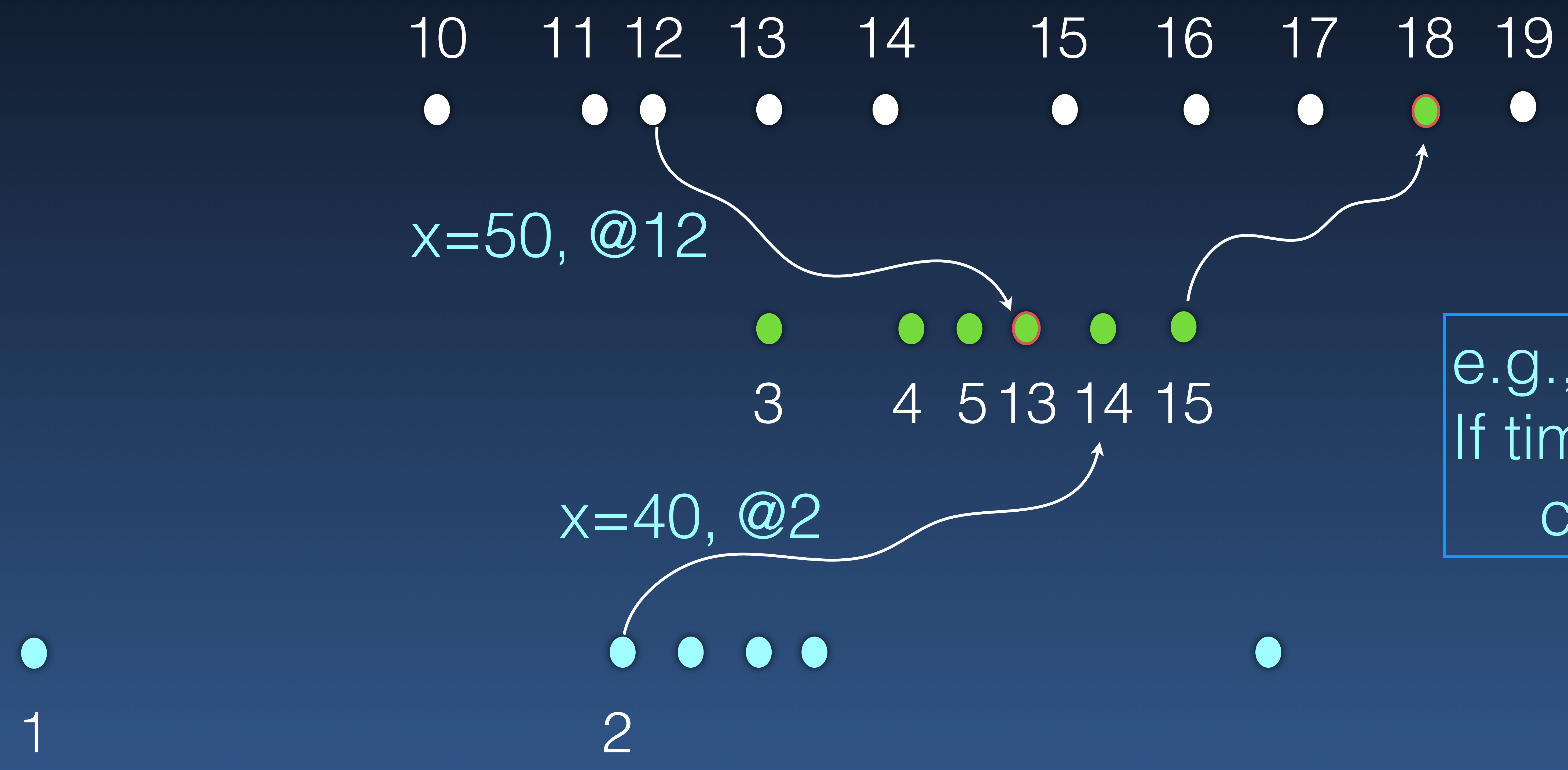
Subodh Kumar

e.g., Lamport Clock



Subodh Kumar

e.g., Lamport Clock



e.g.,
If time >= 2 at all threads:
commit x = 40

Subodh Kumar

- Memory Fences, consistent memory
  - ➡ Registers
- Atomic operations
  - ➡ Test & Set, Fetch & Add, Compare & Swap
- Critical section, Mutex, Ordered
- Barrier
- Lock
- Wait, Condition variables

Properties of Synchronization

- Safety, Liveness
- Blocking
- Starvation-free, Deadlock-free, Obstruction-free, Lockfree, Waitfree
- Distributed or Centralized
  - ➡ OS scheduler, Runtime tools

Synchronization

- Events should happen together
  - ➡ Barrier
- Events should NOT happen together
  - ➡ Mutual Exclusion, Critical Section
- A should happen before B
  - ➡ Conditions, Wait, Ordered
- Lower level Primitives
  - ➡ Locks, Semaphores, Registers, Transactional memory

Ease of use? Overhead?

Progress
- ➡ Starvation
- ➡ Deadlock

Blocking vs Non-blocking

Safety

Liveness

Busy-wait vs OS-scheduled

Fairness

Fairness (Scheduler)

- Strong Fairness
  - ➡ If any synchronizer is ready infinitely often, it should be executed infinitely often
- Weak Fairness
  - ➡ If any synchronizer is ready, it should be executed eventually

Progress

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

Progress

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

All synchronizers succeed in a finite time

Progress

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

Some synchronizer succeeds in a finite time

Subodh Kumar

Progress

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

All waiting synchronizers
are scheduled

Progress

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

Not all synchronizers are blocked

Progress

Liveness

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

Scheduler Agnostic Progress

- Obstruction-free
    - ➡ A given tasks will complete if no other tasks are running (interfering)
- Lockfree
    - ➡ Some task will complete (no matter what others do)
- Waitfree
    - ➡ Each task will complete (no matter what others do)

Complete in a finite number of its own steps

Assume others do not behave maliciously