COL380

Introduction to
Parallel & Distributed Programming

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

Logical-Clock

- Each entity (process) maintains a counter
  - ➡ increments every ‘event,’ at its own pace
- Any 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 on

[Lamport’s Timestamp algorithm]

Causal Ordering

- Logical-clock allows partial ordering of events
- Define causality
  - ➡ $A \rightarrow B \Rightarrow \mathrm{Time}(A) < \mathrm{Time}(B)$
  - ➡ Inverse is not true (Not strong)

$\boxed{\mathrm{Time}(A) < \mathrm{Time}(B) \Rightarrow B \not\rightarrow A}$

May be concurrent

- Vector clocks allow strong causality
- Allows total ordering by using Process-ID to break tie

Synchronization

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

Overhead?

Progress
- ➡ Starvation
- ➡ Deadlock

Blocking vs Non-blocking

Safety

Busy-wait vs OS-scheduled

Fairness
Liveness

Subodh Kumar

- 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

Liveness

	Not lock-based Independent of Scheduler	Lock-based Depends on OS Scheduling
Everyone Progresses	All synchronizers succeed in a finite time Wait Free	All waiting synchronizers are scheduled Starvation Free
Someone Progresses	Some synchronizer succeeds in a finite time Lock Free	Not all synchronizers are blocked Deadlock Free

- Object: lock
- Actions: Lock and Unlock

- Reentrant
- Recursive
- Timed
- Exclusive
- Shared

```
AClass {
    std::mutex mutex;
    Other data;
}
…
Aclass obj;
{
    std::lock_guard<std::mutex> lockhere(obj.mutex);

} //
```

Critical Section

```
obj.mutex.lock();
std::unique_lock<std::mutex> alock(obj.mutex);
```

Condition Variable

- Raise the condition
- Wait for a condition to ‘hold’

```
Produce();
acv.notify_one();
```

```
std::condition_variable acv;
…

std::unique_lock<std::mutex> alock(amutex);
acv.wait(alock);
.. Condition Holds Now ..
Consume();
```

Barrier

- A group of entities
- Wait for all
- Optionally, signal completion

```
std::barrier abarrier(count, completionFn);
…
abarrier.arrive_and_wait();

moreWork();
abarrier.arrive_and_wait();
```

[c++20]

Critical Section

- Block of code
- Criticality context

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