Lecture 2 (Synchronisation)

1 Definition

1. Do operation at a certain time T
2. Two or a set of events should happen together - barrier
3. Any two events should not happen together - mutual exclusion, critical section
4. Event A should happen after event B - conditions

2 Logical Clock

1. Every entity maintains a counter
2. Increment happens at every 'event' of that entity
3. Interaction between entities happens via `data + counter`
4. On receiving message, if recipient counter < received counter, then increase local counter to received counter and increment it by one since receiving is also an event
5. This is **Lamport's Timestamp Algorithm**
6. It allows partial ordering of events
7. Causality is maintained: $A \rightarrow B \implies time(A) < time(B)$, however the inverse need not be true, all we know is $time(A) < time(B) \implies B \not\rightarrow A$
8. We can have a vector clock instead to have strong causality (so that the inverse is also true)
9. Partial ordering can be changed to total ordering using process ID, but isn't much useful

3 Lower Level Primitives for Synchronisation

1. Locks
2. Semaphores
3. Register
4. Transactional memory

4 Progress

1. Starvation - each synchroniser gets to make progress, it is starvation-free
2. Deadlock - if each synchroniser gets to make some progress, it is deadlock-free

4.1 Types of Primitives

- Busy-wait vs OS-scheduled
  1. Busy-wait - `while (!condition);`
  2. OS-scheduled - scheduler sends signal to start computation, until then process is inactive
- Blocking vs non-blocking
- Fairness vs liveness

4.2 Fairness

1. Strong - if any synchroniser is ready infinitely often, then it should be executed infinitely often
2. Weak - if any synchroniser is ready, it should be executed eventually

	Not lock-based (independent of scheduler)	Lock-based (OS scheduling)
Everyone progresses	Wait free	Starvation free
Someone progresses	Lock free	Deadlock free

5 Lock

1. `mutex` and `lock_guard` is used
2. `mutex.lock()` can also be used
3. Alternatively `unique_lock` can be used which provides more freedom

5.1 Types

1. Re-entrant
2. Recursive
3. Timed
4. Exclusive
5. Shared

6 Condition Variable

```
// defining
std::condition_variable acv;

// implement wait
acv.wait(some_lock);

// notify to release the wait
acv.notify_one();
```

```
// or
acv.notify_all();
```

7 Barrier

Wait for all

```
// works only in C++ 2020
// define
std::barrier abarrier(count, completion_function);

// on reaching the barrier
abarrier.arrive_and_wait(); // barrier waits here until 'count' number of synchronisers
                            // have reached
                            // and then calls completion_function
```

8 Critical Section

```
// omp automatically locks code with same names
#pragma omp critical (aname)
{
    // mutually excluded code
}
```