Conditions for Deadlock

- deadlock requires four conditions

> - **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
> - **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
> - **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
> - **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

- to prevent deadlock, ensure that at least one of the four conditions isn't possible

Deadlock Prevention

> - **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).

- avoid the need for locks to ensure mutual exclusion through *lock-free* (and *wait-free*) data structures
  - e.g. atomic increment without locks

```
void AtomicIncrement(int *value, int amount) {
  do {
    int old = *value;
  } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

  - e.g. list insert without locks

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  do {
    n->next = head;
  } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

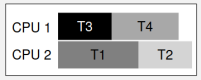
these simple versions avoid deadlock but livelock is possible

more sophisticated solutions are complex

Deadlock Avoidance

- the four conditions only mean that deadlock is possible – an unfortunate interleaving of steps *can* occur but isn't inevitable
- *deadlock avoidance* refers to scheduling threads so that deadlock doesn't occur

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no



T1, T2 need the same locks so can't be scheduled at the same time

T3 only needs one lock, so can't be involved in circular wait

- challenges
  - requires global knowledge of lock acquisition
  - can significantly limit concurrency
  - generally not a practical solution

Deadlock Detection

- if deadlock occurs infrequently, *deadlock detection* can be more effective than prevention
  - e.g. periodically build dependency graph for held and desired locks, then check for cycles
    - if cycle is found, restart

Deadlock Best Practices

- best strategy is prevention
  - be careful
  - have a lock acquisition order
- there are situations where detection is a useful addition