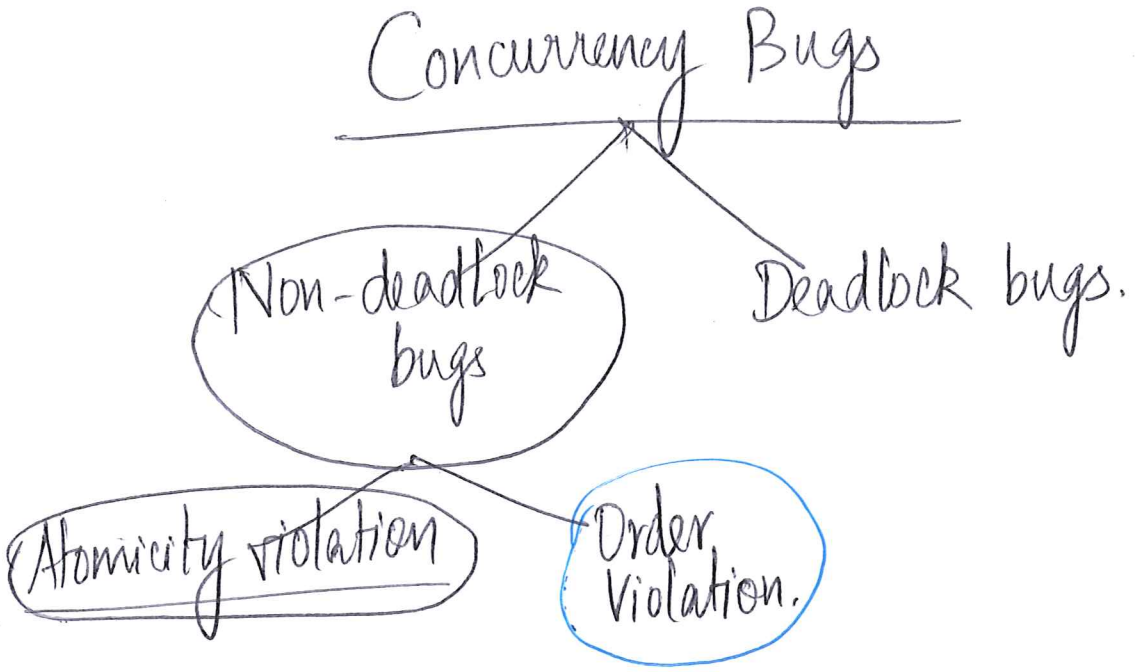


T_1 :

```
. . .
if (thd -> proc_info) {
    . . .
    fputs (thd -> proc_info, ...);
    . . .
}
```

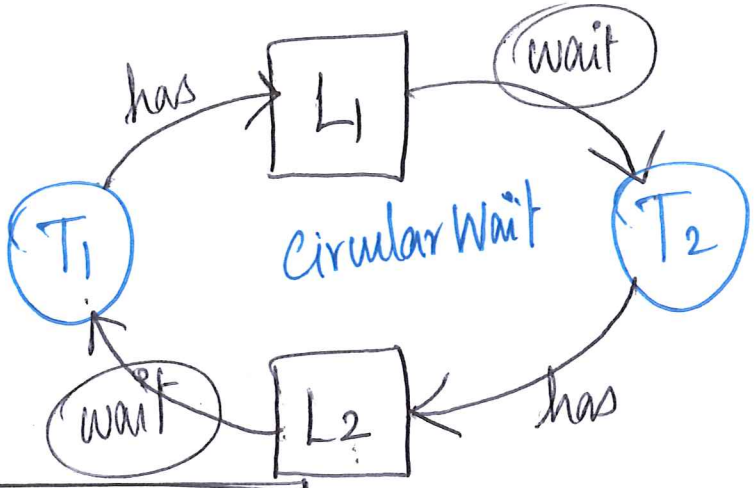
T_2 :

```
thd -> proc_info = NULL;
```

Deadlocks

T_1 :
① lock(L_1);
③ lock(L_2);

T_2 :
② lock(L_2);
④ lock(L_1);



Encapsulation

T_1

```
Vector v1, v2;
v1.addAll(v2);
```

T_2

```
v2.addAll(v1);
```

4 Conditions for Deadlock

1. Mutual exclusion.
2. Hold-and-Wait.
3. No preemption.
4. Circular Wait.

Solutions

1. Prevention
2. Avoidance.

① Prevention →

1. ~~Circular Wait~~

Ordering of Locks

- Total: $L_1 \to L_2 \to L_5 \to L_3 \to L_4$
- Partial: $L_2 \to L_3 \to L_4$, $L_1 \to L_5 \to L_2$.

---

② Hold-And-Wait

T_1: lock(gL)
L_1
L_2
unlock(gL)

T_2: lock(gL)
L_2
L_1
unlock(gL).

3. No preemption.

T_1: L_1
L_2

T_2: L_2
L_1

trylock(L_1) (0)
- return success if lock is free
- " error " " held.

top: →
t_1 lock(L_1);
t_2 if (trylock(L_2) != 0) {
t_3 unlock(L_1);
t_4 goto top;
}

top:
q_1 lock(L_2);
q_2 if (trylock(L_1) != 0)
{
q_3 unlock(L_2);
q_4 goto top;
}

LiveLock

t_1 ~~t_2~~ q_1 t_2 q_2 t_3 q_3 t_4 q_4 . .

t_1 q_1 t_2 q_2 t_3 q_3 t_4 q_4

(contd...)

4th prevention technique -- Avoid Mutual Exclusion.

```
int CompareAndSwap(int *addr, int expected, int new)
{
    if (*addr == expected) {
        *addr = new;
        return 1; // success.
    }
    return 0;
}
```

* Atomically increment a value by a certain amount.

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

* Insert in a Linked List.

```
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);
}
```

4. Mutual Exclusion

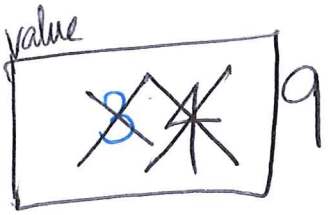
lock-free data structures

wait-free "

```
void AtomicIncrement (int *value, int amoun
{   do {
        int old = *value;   // l1
    } while (CompareAndSwap (value, old, old + amount)
                                          // l2
                == 0
}
```

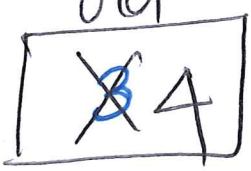
amount = 5

value



l1:

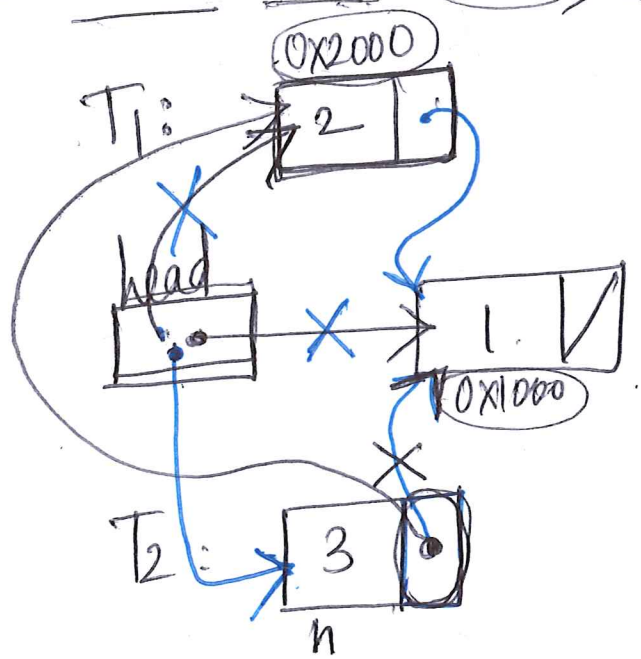
old

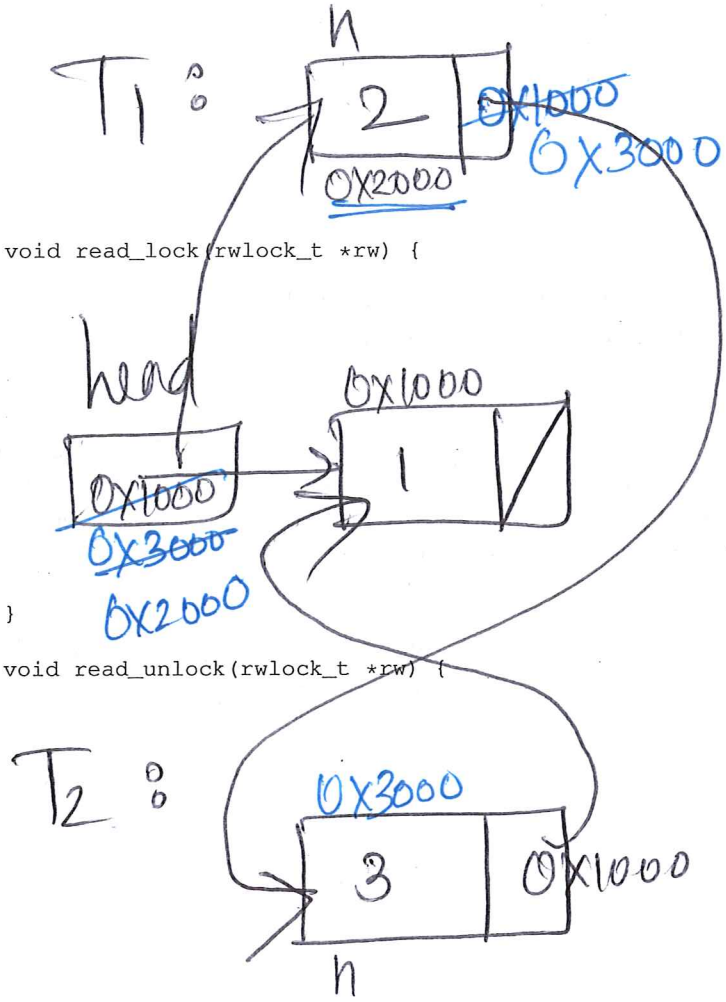


```
void ListInsert(int value) {
    node_t *n = malloc(sizeof(node_t));
    n->value = value;
    n->next = head;
    head = n;
}
```

```
do {
    n->next = head;
} while(CAS(&head, n->next, n) == 0)
```

interrupt






```
void read_lock(rwlock_t *rw) {

}
void read_unlock(rwlock_t *rw) {

}
void write_lock(rwlock_t *rw) {

}
void write_unlock(rwlock_t *rw) {

}
```

0X1000 == 0x1000

0X 3000 == 0x100

0x3000 == 0x3000