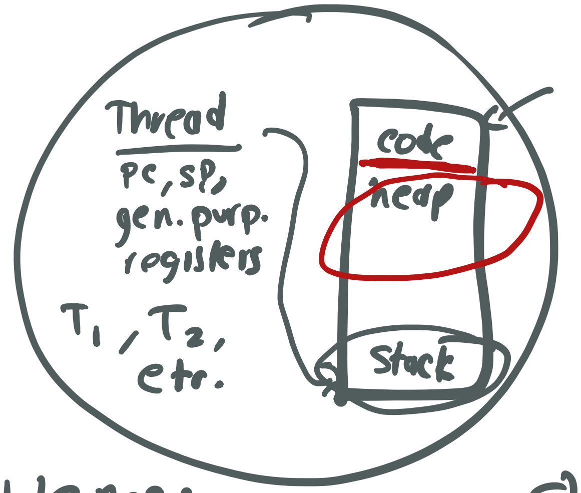
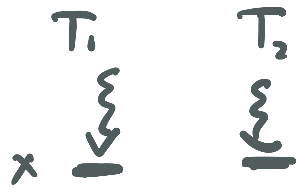
Concurrency : Continued



Problems:

→ Atomicity — data — arises when: concurrent updates to shared data

→ Ordering → [later] control (execution) of instructions



t_2 to wait until T_1 has executed x

classic example:

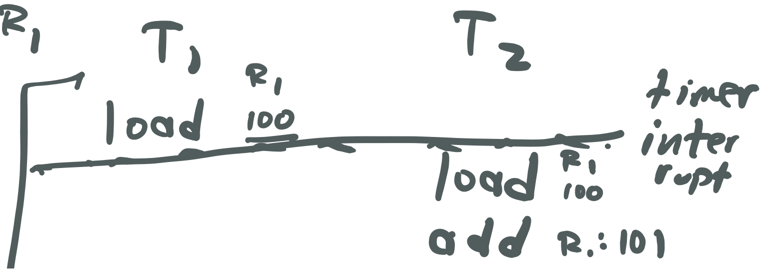
if shared, a worry! (concurrency)

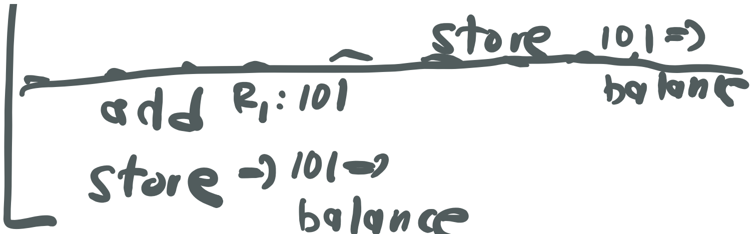
balance++;

want: all at once

- load → R_1
- add
- store

balance: 100





run twice, but balance only inc'd once

another example:

list insert (shared data structure)

T₁	T₂
list_insert( )	list_insert( )

```
struct node {
    int value;
    node_t *next;
};
```

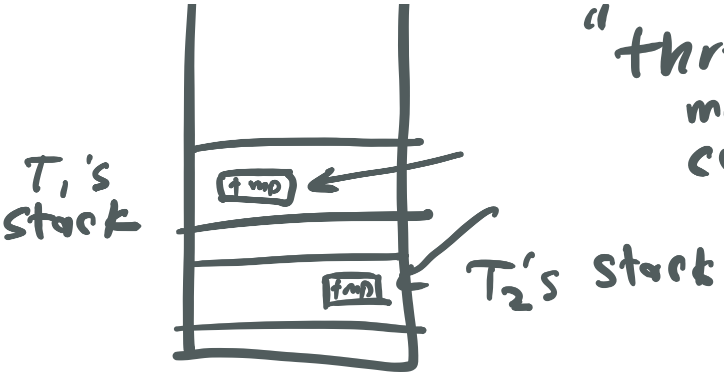
global:
node_t *head;
// init'd to NULL

```
void list_insert(int v) {    assume: "thread safe"
L1 ✓ node_t *tmp = malloc(sizeof(node_t));
L2 ✓ assert(tmp != NULL);  ←
L3 ✓ tmp->value = v;
L4   tmp->next = head;
L5   head = tmp;
}
```

tmp: stack

T₁	T₂
L₁	L₁

each have their own stack, own tmp



"thread safe:" multiple threads can call it w/o locking

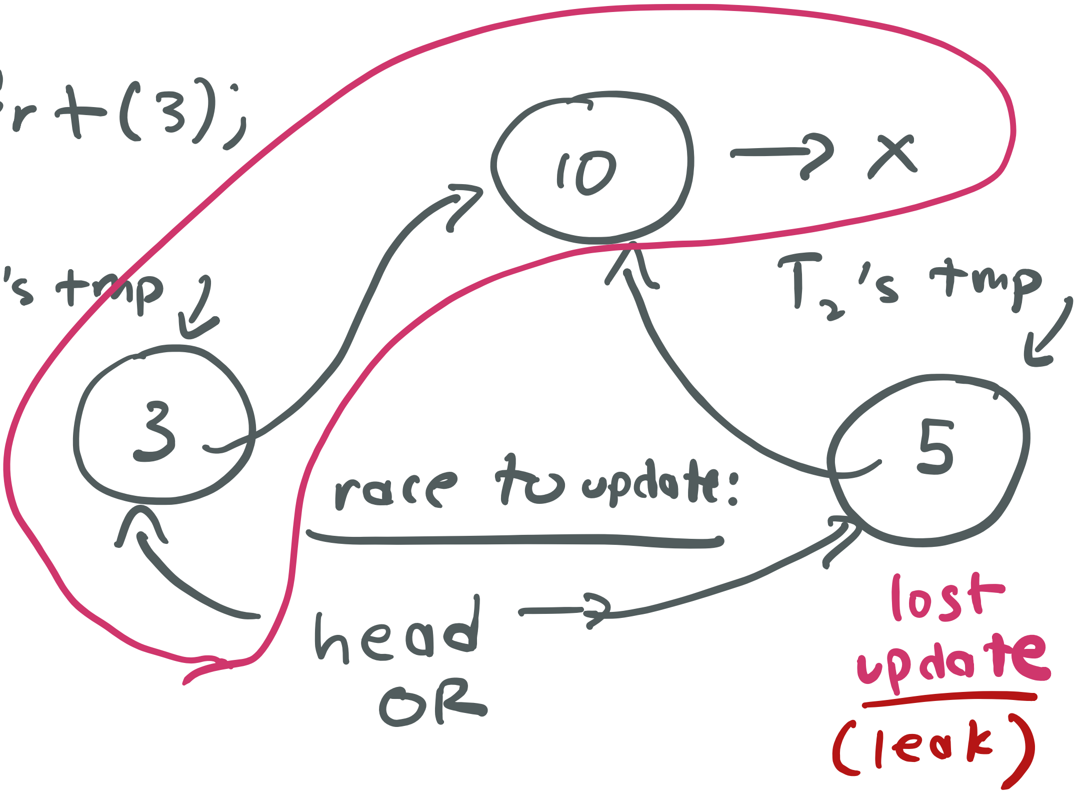
critical section

```
   Lock();
L4 tmp -> next = head;
L5 head = tmp;
   unlock();
```

{data race}

T_2
insert(5);

T_1
insert(3);



```
Lock();
node_t *tmp = malloc(sizeof(node_t));
if (tmp == NULL)  ← forgot to unlock?
    unlock();
    return;
tmp -> value = v;
[tmp -> next = head;]
[head = tmp;        ]
Unlock();
```

smaller crit. sects are better

Q1) How to implement a lock?

key: need h/w support
(hardware)

[more powerful instructions]

=> later, also need OS support

Q2) what makes a good lock?

properties: fairness, low overhead, correctness

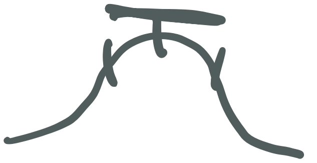
Admin:

- do

=) **p2b** ↙

=) **midterm'**: [**graded**]

average: ~22/32



low: >0 high: **31**

"Spin Lock"

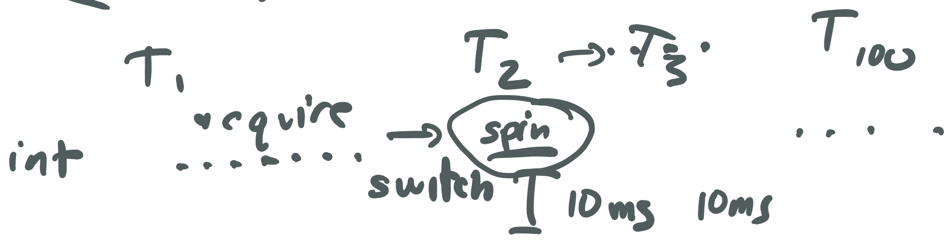
=) correct ✓

=) **overhead**

[=) under high contention, lots of CPU cycles wasted]

=) **fairness**

[could spin for "long time"]


T₁
acquire
int
T₂ → T₃
spin
switch
10ms
10ms
T₁₀₀