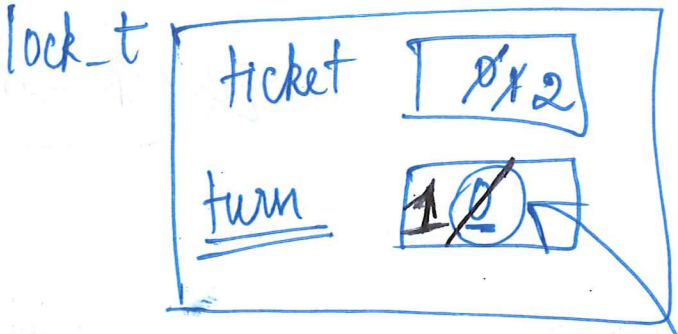


FOURTH PRIMITIVE: FETCH-AND-ADD

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

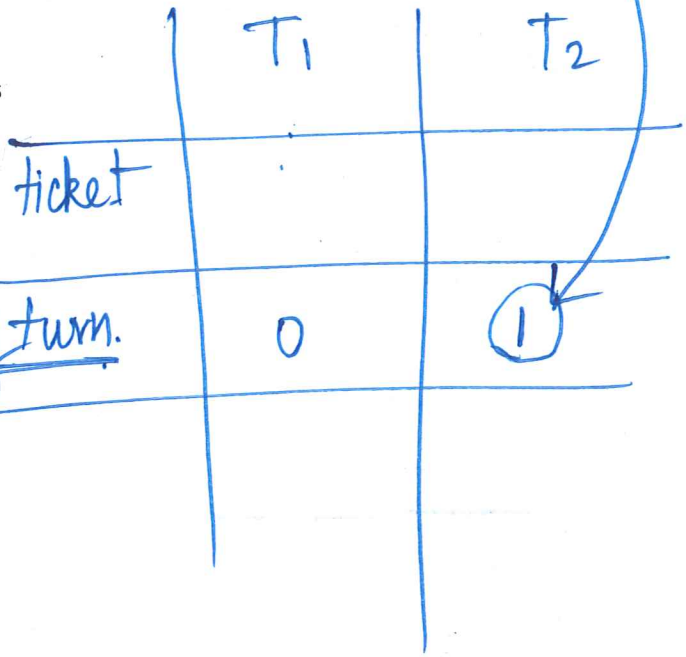
EXAMPLE USING FETCH-AND-ADD: TICKET LOCKS

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void acquire(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void release(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```



T_1 is in C.S.

T_1 is done with C.S.

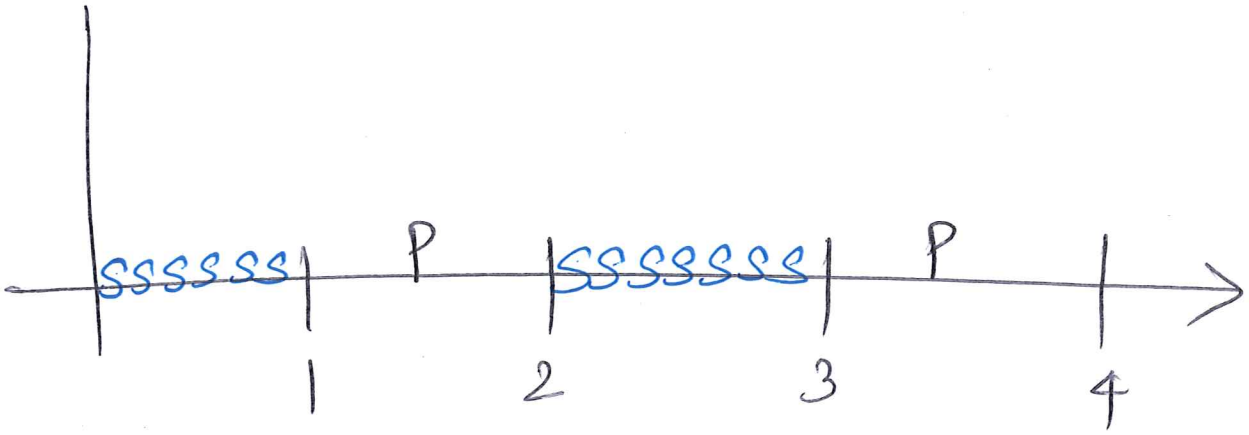
mutual ex. ✓

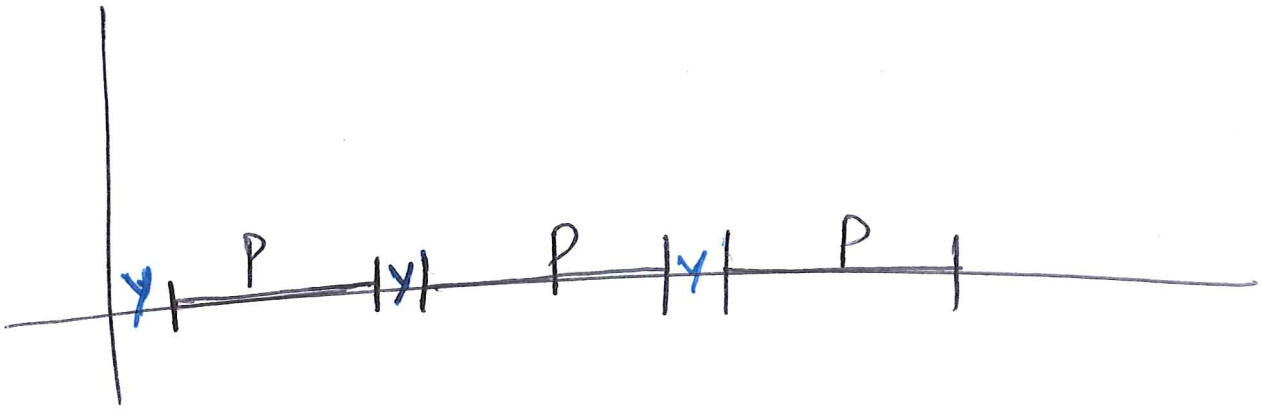
fairness ✓

performance ✗

yield()

running → ready





O(threads * time slice)

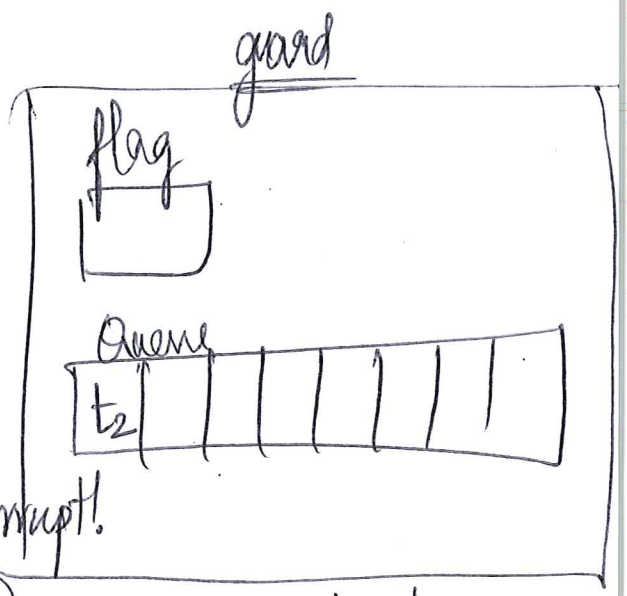
w/ yield O(threads * context switching)

```
1  // Lock with Queues, Test-And-Set, Yield, and Wakeup
2
3  typedef struct __lock_t {
4      int         flag;   // state of lock: 1=held, 0=free
5      int         guard;  // use to protect flag, queue
6      queue_t     *q;     // explicit queue of waiters
7  };
8
9
10 void lock_init(lock_t *lock) {
11     lock->flag = lock->guard = 0;
12     lock->q    = queue_init();
13 }
14
15
16 void lock(lock_t *lock) {
17     while (xchg(&lock->guard, 1) == 1)
18         ; // spin
19     if (lock->flag == 0) { // lock is free: grab it!
20         lock->flag = 1;
21         lock->guard = 0;
22     } else {                    // lock not free: sleep
23         queue_push(lock->q, gettid());
24         lock->guard = 0;
25         park();                 // put self to sleep
26     }
27 }
28
29
30 void unlock(lock_t *lock) {
31     while (xchg(&lock->guard, 1) == 1)
32         ; // spin
33     if (queue_empty(lock->q))
34         lock->flag = 0;
35     else
36         unpark(queue_pop(lock->q));
37     lock->guard = 0;
38 }
```

park()

unpark(threadID)

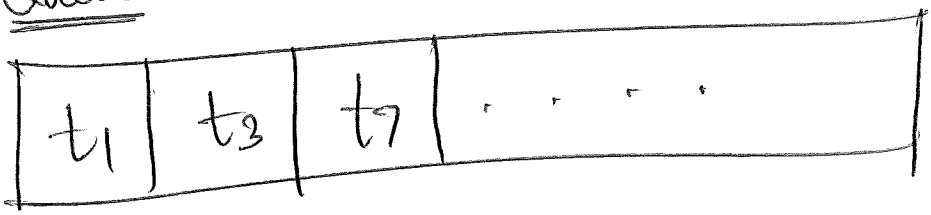
setpark();



T_1 — lock() ✓
CS ✓
unlock()
unpark(t_2)
DONE

T_2
lock()
24 ✓
interrupt!
25 park();
T_2 goes to sleep

Queue



t_1

lock()

CS

unlock()

t_2

spin wait()

```c
// Make this Hash Table implementation to be thread-safe!

#define SKIP_MAIN
#include "threads-list-simple.c"

#define HASH_BUCKETS (7)
typedef struct __hash_t {
  list_t hlists[HASH_BUCKETS];
  pthread_mutex_t m[HASH_BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
  int i;
  for (i = 0; i < HASH_BUCKETS; i++) {
    List_Init(&H->hlists[i]);
  }
}

void Hash_Insert(hash_t *H, int key) {
  int b = key % HASH_BUCKETS;
  List_Insert(&H->hlists[b], key);
}

void Hash_Print(hash_t *H) {
  int i;
  for (i = 0; i < HASH_BUCKETS; i++) {
    printf("LIST %d: ", i);
    List_Print(&H->hlists[i]);
  }
}

int main(int argc, char *argv[]) {
  hash_t myhash;
  Hash_Init(&myhash);
  Hash_Insert(&myhash, 10);
  Hash_Insert(&myhash, 5);
  Hash_Insert(&myhash, 30);
  Hash_Print(&myhash);
  return 0;
}
```

CS