Race conditions

- Disk driver maintains a list of outstanding requests
- Each process can add requests to the list
struct list {
    int data;
    struct list *next;
};

struct list *list = 0;

insert(int data) {
    struct list *l;

    l = malloc(sizeof *l);
    l->data = data;
    l->next = list;
    list = l;
}

List implementation
no locks
Request queue (e.g. incoming network packets)

- Linked list, list is pointer to the first element
CPU1 allocates new request
CPU2 allocates new request
CPUs 1 and 2 update next pointer

l->next = list
CPU1 updates head pointer

list = l
CPU2 updates head pointer
State after the race
Mutual exclusion

- Only one CPU can update list at a time
struct list {
    int data;
    struct list *next;
};

struct list *list = 0;

struct lock listlock;

insert(int data)
{
    struct list *l;
    l = malloc(sizeof *l);
    acquire(&listlock);
    l->data = data;
    l->next = list;
    list = l;
    release(&listlock);
}
• How can we implement acquire()?
void acquire(struct spinlock *lk) {
    for(;;) {
        if(!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}

• Spin until lock is 0
• Set it to 1
21 void
22 acquire(struct spinlock *lk)
23 {
24   for(;;) {
25     if(!lk->locked) {
26       lk->locked = 1;
27       break;
28     }
29   }
30 }

- Two CPUs can reach line #25 at the same time
  - See not locked, and
  - Acquire the lock
- Lines #25 and #26 need to be atomic
  - I.e. indivisible
Compare and swap: xchg

- Swap a word in memory with a new value
  - Return old value
Correct implementation

1573 void
1574 acquire(struct spinlock *lk)
1575 {
...
1580 // The xchg is atomic.
1581 while(xchg(&lk->locked, 1) != 0)
1582 ;
...
1592 }
static inline uint xchg(volatile uint *addr, uint newval) {
    uint result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr), "=a" (result) : "1" (newval) : "cc");

    return result;
}
1573 void
1574 acquire(struct spinlock *lk)
1575 {
...
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
1584   // Tell the C compiler and the processor to not move loads or
stores
1585   // past this point, to ensure that the critical section’s memory
1586   // references happen after the lock is acquired.
1587   __sync_synchronize();
...
1592 }
Thank you!