Race conditions

- Disk driver maintains a list of outstanding requests
- Each process can add requests to the list
1 struct list {
2   int data;
3   struct list *next;
4 }
...
6 struct list *list = 0;
...
9 insert(int data)
10 {
11   struct list *l;
12
13   l = malloc(sizeof *l);
14   l->data = data;
15   l->next = list;
16   list = l;
17 }
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

l->next = list

list
CPU1 updates head pointer

list = l
CPU2 updates head pointer

list = l
State after the race

list = l
Mutual exclusion

- Only one CPU can update list at a time
List implementation with locks

- Critical section
• 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
#include <spinlock.h>

21 void
22 acquire(struct spinlock *lk)
23 {
24   for(;;) {
25     if(!lk->locked) {
26       lk->locked = 1;
27       break;
28     }
29   }
30 }

Still incorrect

- 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
void acquire(struct spinlock *lk) {
  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0) ;
  ...
  ...
  }

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;
}
void acquire(struct spinlock *lk) {
  ...
  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0) ;
  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section’s memory
  // references happen after the lock is acquired.
  __sync_synchronize();
  ...
}
Deadlocks
Deadlocks

acquire(A)  ←  acquire(B)

acquire(B) {  
    while(xchg(&B->locked, 1) != 0)
}  ↓  acquire(A) {  
    while(xchg(&A->locked, 1) != 0)
}  ↓
Lock ordering

- Locks need to be acquired in the same order
Locks and interrupts

network_packet()
    ....
    insert()
        acquire(A)
    ...
}

Network interrupt

network_packet()
    ....
    insert()
        acquire(A)
    ...
}
Locks and interrupts

- Never hold a lock with interrupts enabled
void acquire(struct spinlock *lk) {
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");
    // The xchg is atomic.
    while(xchg(&lk−>locked, 1) != 0)
        ;

    __sync_synchronize();

    ...

    DISABLE_INTERRUPTS
Simple disable/enable is not enough

- If two locks are acquired
- Interrupts should be re-enabled only after the second lock is released
- Pushcli() uses a counter
1655 pushcli(void)
1656 {
1657   int eflags;
1658
1659   eflags = readeflags();
1660   cli();
1661   if(cpu->ncli == 0)
1662     cpu->intena = eflags & FL_IF;
1663   cpu->ncli += 1;
1664 }
...
1667 popcli(void)
1668 {
1669   if(readeflags()&FL_IF)
1670     panic("popcli - interruptible");
1671   if(--cpu->ncli < 0)
1672     panic("popcli");
1673   if(cpu->ncli == 0 && cpu->intena)
1674     sti();
1675 }
Thank you!