Recap: Race conditions

- Disk driver maintains a list of outstanding requests
- Each process can add requests to the list
Request queue (e.g. incoming network packets)

- Linked list, list is pointer to the first element
List implementation with locks

- Critical section

```c
9 insert(int data)
10 {
11   struct list *l;
13   l = malloc(sizeof *l);
14     acquire(&listlock);
14   l->data = data;
15   l->next = list;
16     release(&listlock);
17   list = l;
17 }
```
Xchg instruction

- Swap a word in memory with a new value
  - Atomic!
  - Return old value
1573 void
1574 acquire(struct spinlock *lk)
1575 {
...
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
...
1592 }
insert(int data)
{
    struct list *l;
    l = malloc(sizeof *l);
    acquire(&listlock);
    l->data = data;
    l->next = list;
    list = l;
    release(&listlock);
}
Correct implementation

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_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();
    ...
}
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
pushcli(void)
{
  int eflags;
  eflags = readeflags();
  cli();
  if(cpu->ncli == 0)
    cpu->intena = eflags & FL_IF;
  cpu->ncli += 1;
}

popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli − interruptible");
  if(--cpu->ncli < 0)
    panic("popcli");
  if(cpu->ncli == 0 && cpu->intena)
    sti();
}
Locks and interprocess communication
Send/receive queue

100 struct q {
101   void *ptr;
102 };
103
104 void*
105 send(struct q *q, void *p)
106 {
107   while(q->ptr != 0)
108     ;
109   q->ptr = p;
110 }
112 void*
113 recv(struct q *q)
114 {
115   void *p;
116
117   while((p = q->ptr) == 0)
118     ;
119   q->ptr = 0;
120   return p;
121 }
Send/receive queue

- Expensive if communication is rare
- Receiver wastes CPU cycles
Sleep and wakeup syscalls

- sleep(channel)
  - Put calling process to sleep
  - Release CPU for other work
- wakeup(channel)
  - Wakes all processes sleeping on a channel
    - If any
  - i.e., causes sleep() calls to return
Send/receive queue

201 void *
202 send(struct q *q, void *p) 
203 { 
204   while(q->ptr != 0) 
205     ; 
206   q->ptr = p; 
207   wakeup(q); /*wake recv*/ 
208 } 

210 void *
211 recv(struct q *q) 
212 { 
213   void *p; 
214   
215   while((p = q->ptr) == 0) 
216     sleep(q); 
217   q->ptr = 0; 
218   return p; 
219 }
Send/receive queue

201 void*
202 send(struct q *q, void *p)
203 {
204    while(q->ptr != 0)
205        ;
206    q->ptr = p;
207    wakeup(q); /*wake recv*/
208 }

210 void*
211 recv(struct q *q)
212 {
213    void *p;
214
215    while((p = q->ptr) == 0)
216        sleep(q);
217    q->ptr = 0;
218    return p;
219 }

• recv() gives up the CPU to other processes
  • But there is a problem...
Lost wakeup problem

recv

215

test

216

sleep

wait for wakeup forever

Time

send

206

store p

207

wakeup

204

test

205

spin forever
Thank you!