ICS143A: Principles of Operating Systems

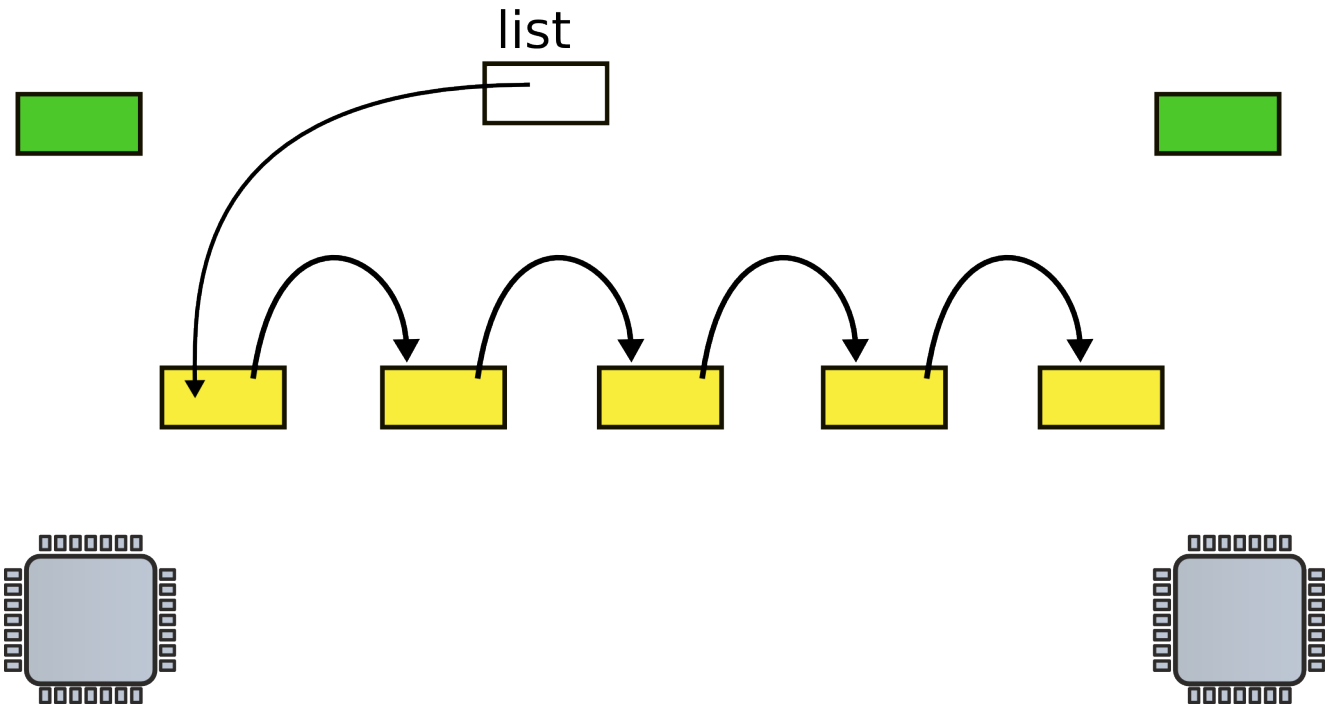
Lecture 16: Locking (continued)

Anton Burtsev
February, 2017

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

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

- Critical section

Xchg instruction

- Swap a word in memory with a new value
  - Atomic!
  - 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 }
```

One last detail...

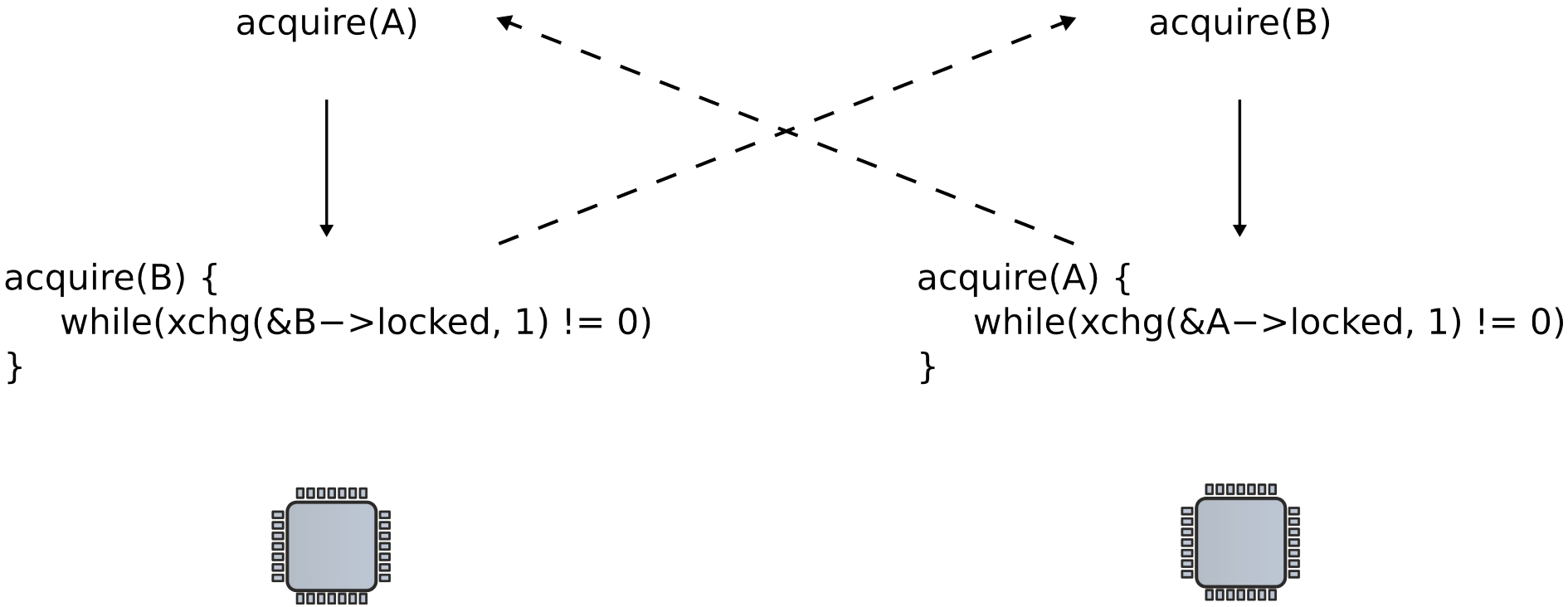
```
9 insert(int data)
10 {
11    struct list *l;
13    l = malloc(sizeof *l);
      acquire(&listlock);
14    l->data = data;
15    l->next = list;
16    list = l;
      release(&listlock);
17 }
```

Correct implementation

```
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 }
```

Deadlocks

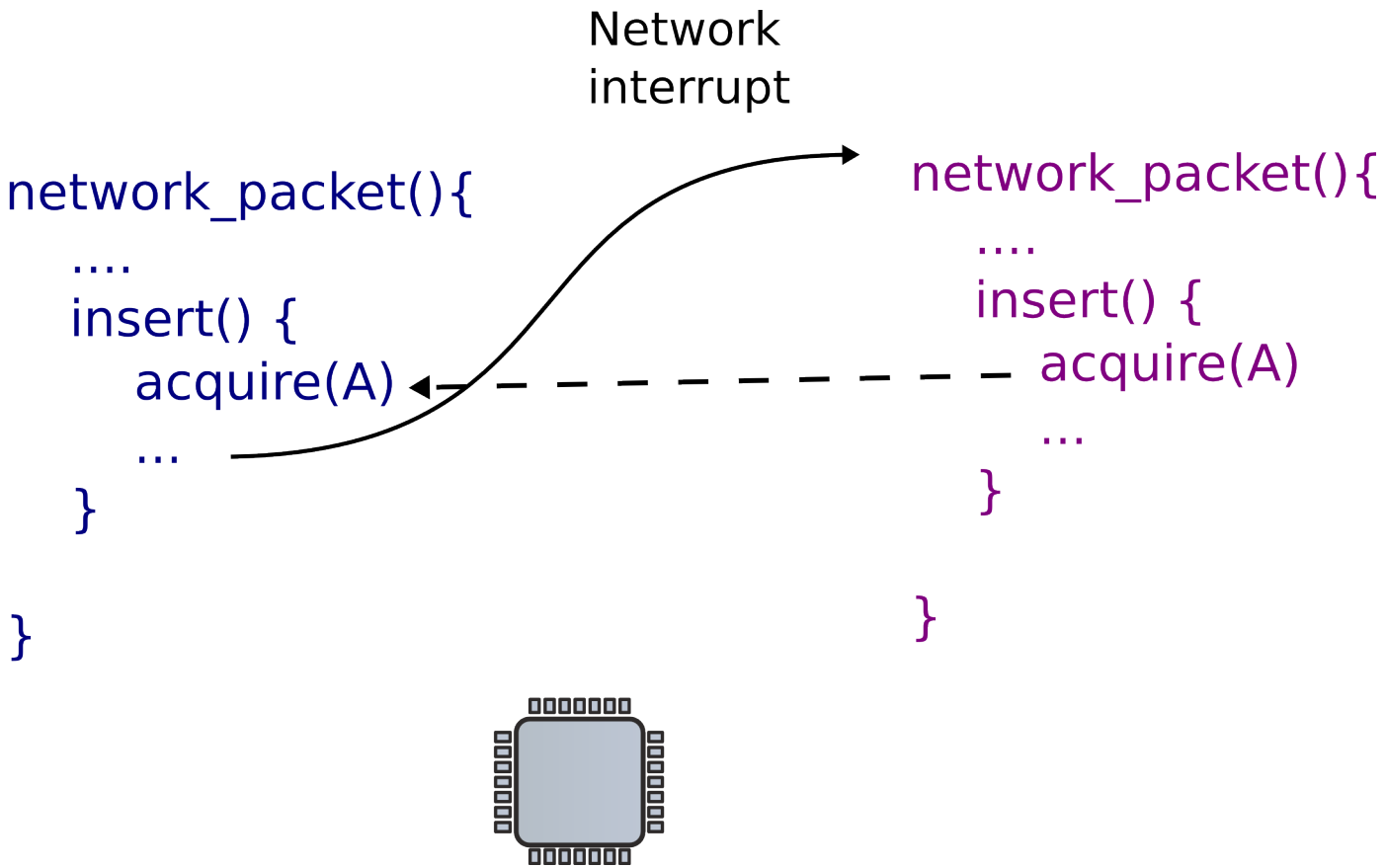
Deadlocks


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



Locks and interrupts

- Never hold a lock with interrupts enabled

```
1573 void
1574 acquire(struct spinlock *lk)
1575 {
1576   pushcli(); // disable interrupts to avoid deadlock.
1577   if(holding(lk))
1578     panic("acquire");
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
...
1587   __sync_synchronize();
...
1592 }
```

Disabling 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

Pushcli()/popcli()

```
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 }
```

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

```
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 }
```

- 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 }
```

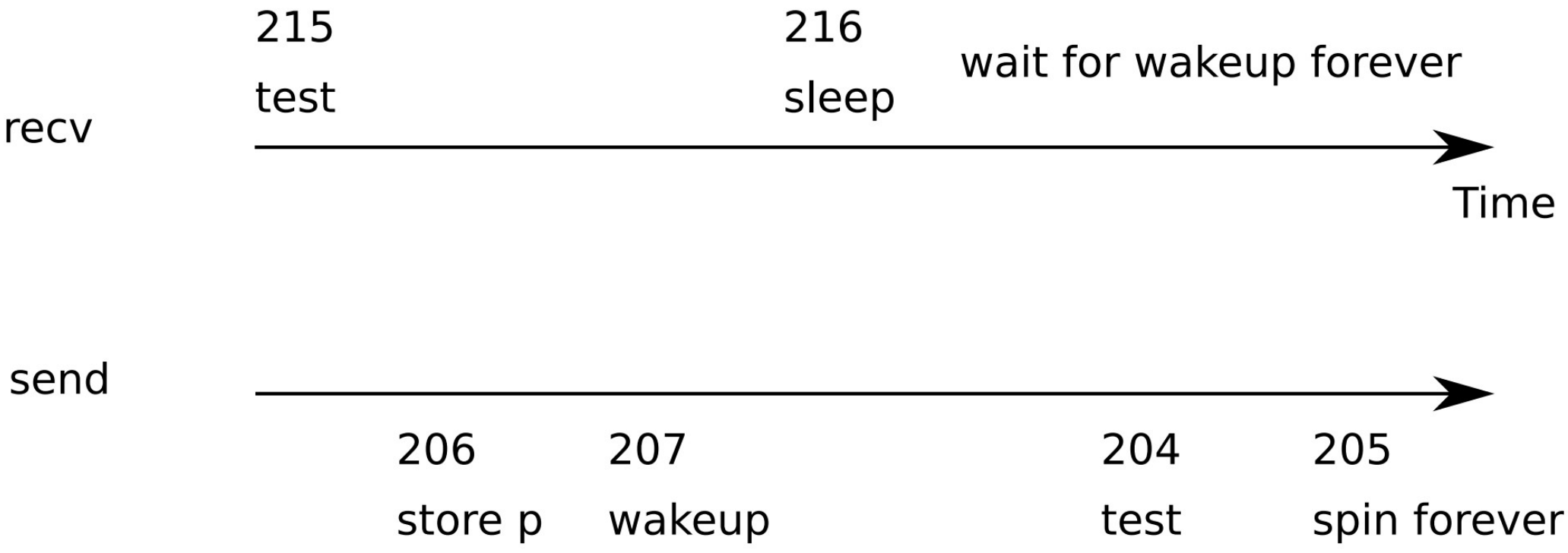
Send/receive queue

```
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



Thank you!