ICS143A: Principles of Operating Systems

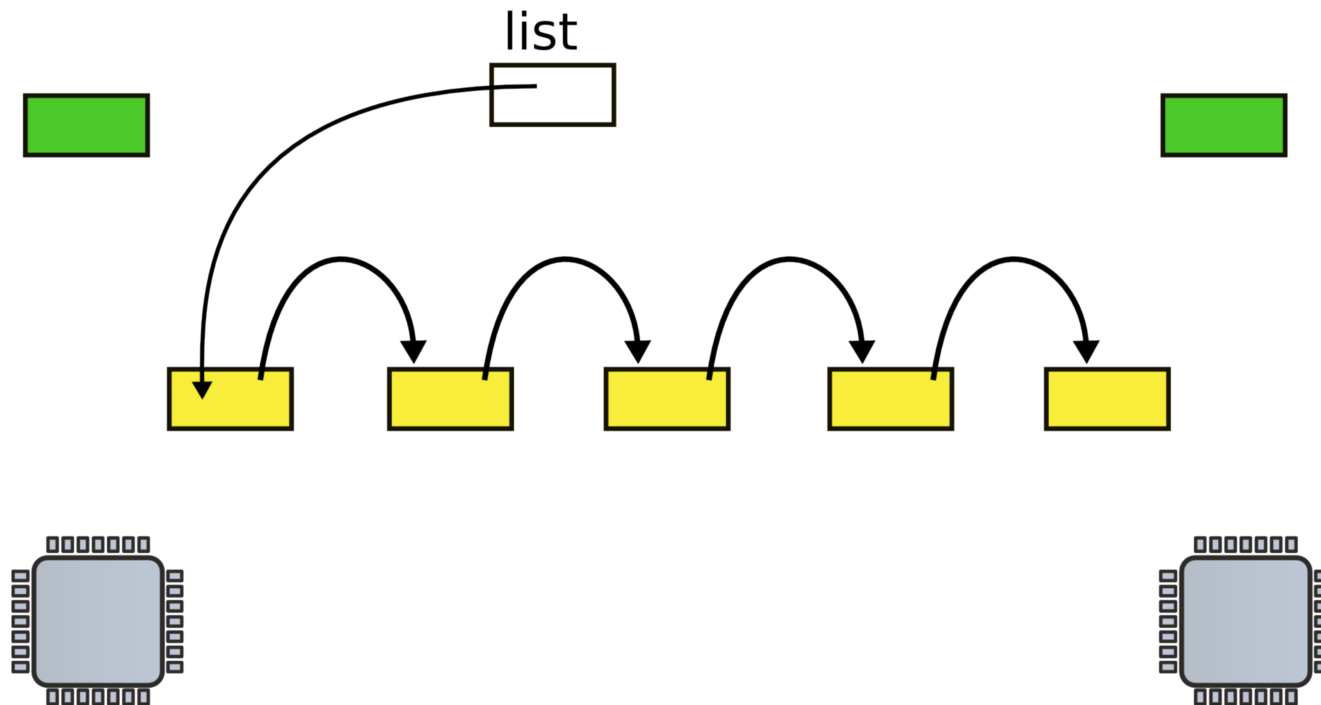
Lecture 16: Locking (continued)

Anton Burtsev
November, 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 }
```

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 }
```

- Sends one pointer between two CPUs

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 }
```

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 }
```

- Works well, but expensive if communication is rare
  - Receiver wastes CPU cycles

Sleep and wakeup

- 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 }
```

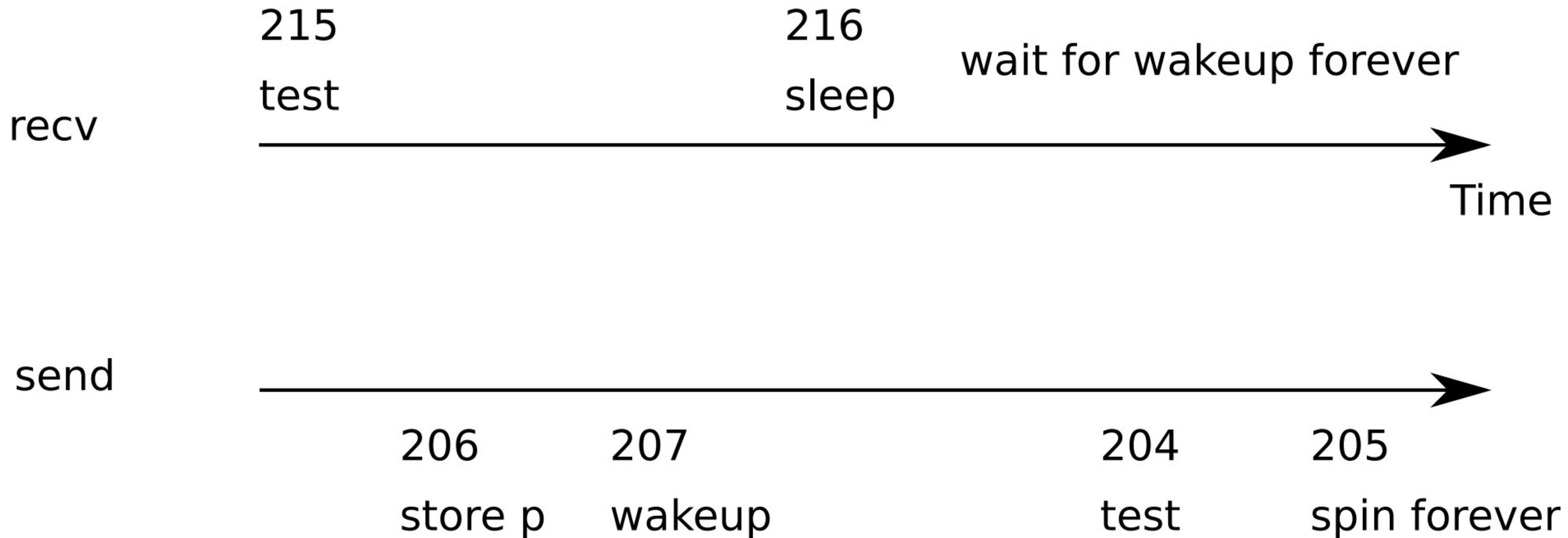
Send/receive queue

```
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



Lock the queue

```
300 struct q {
301   struct spinlock lock;
302   void *ptr;
303 };
304
305 void*
306 send(struct q *q, void *p)
307 {
308   acquire(&q->lock);
309   while(q->ptr != 0)
310     ;
311   q->ptr = p;
312   wakeup(q);
313   release(&q->lock);
314 }
```

```
316 void*
317 recv(struct q *q)
318 {
319   void *p;
320
321   acquire(&q->lock);
322   while((p = q->ptr) == 0)
323     sleep(q);
324   q->ptr = 0;
325   release(&q->lock);
326   return p;
327 }
```

- Doesn't work either: deadlocks
  - Holds a lock while sleeping

Pass lock inside sleep()

```
300 struct q {
301   struct spinlock lock;
302   void *ptr;
303 };
304
305 void*
306 send(struct q *q, void *p)
307 {
308   acquire(&q->lock);
309   while(q->ptr != 0)
310     ;
311   q->ptr = p;
312   wakeup(q);
313   release(&q->lock);
314 }
```

```
316 void*
317 recv(struct q *q)
318 {
319   void *p;
320
321   acquire(&q->lock);
322   while((p = q->ptr) == 0)
323     sleep(q, &q->lock);
324   q->ptr = 0;
325   release(&q->lock);
326   return p;
327 }
```

sleep()

- Acquire ptable.lock
  - All process operations are protected with ptable.lock

```
2809 sleep(void *chan, struct spinlock *lk)
2810 {
...
2823   if(lk != &ptable.lock){
2824     acquire(&ptable.lock);
2825     release(lk);
2826   }
2827
2828   // Go to sleep.
2829   proc->chan = chan;
2830   proc->state = SLEEPING;
2831   sched();
...
2836   // Reacquire original lock.
2837   if(lk != &ptable.lock){
2838     release(&ptable.lock);
2839     acquire(lk);
2840   }
2841 }
```

sleep()

```
2809 sleep(void *chan, struct spinlock *lk)
2810 {
...
2823   if(lk != &ptable.lock){
2824     acquire(&ptable.lock);
2825     release(lk);
2826   }
2827
2828   // Go to sleep.
2829   proc->chan = chan;
2830   proc->state = SLEEPING;
2831   sched();
...
2836   // Reacquire original lock.
2837   if(lk != &ptable.lock){
2838     release(&ptable.lock);
2839     acquire(lk);
2840   }
2841 }
```

- Acquire `ptable.lock`
  - All process operations are protected with `ptable.lock`
- Release `lk`
  - Why is it safe?

sleep()

```
2809 sleep(void *chan, struct spinlock *lk)
2810 {
...
2823   if(lk != &ptable.lock){
2824     acquire(&ptable.lock);
2825     release(lk);
2826   }
2827
2828   // Go to sleep.
2829   proc->chan = chan;
2830   proc->state = SLEEPING;
2831   sched();
...
2836   // Reacquire original lock.
2837   if(lk != &ptable.lock){
2838     release(&ptable.lock);
2839     acquire(lk);
2840   }
2841 }
```

- Acquire ptable.lock
  - All process operations are protected with `ptable.lock`
- Release `lk`
  - Why is it safe?
  - Even if new wakeup starts at this point, it cannot proceed
  - Sleep() holds `ptable.lock`

wakeup()

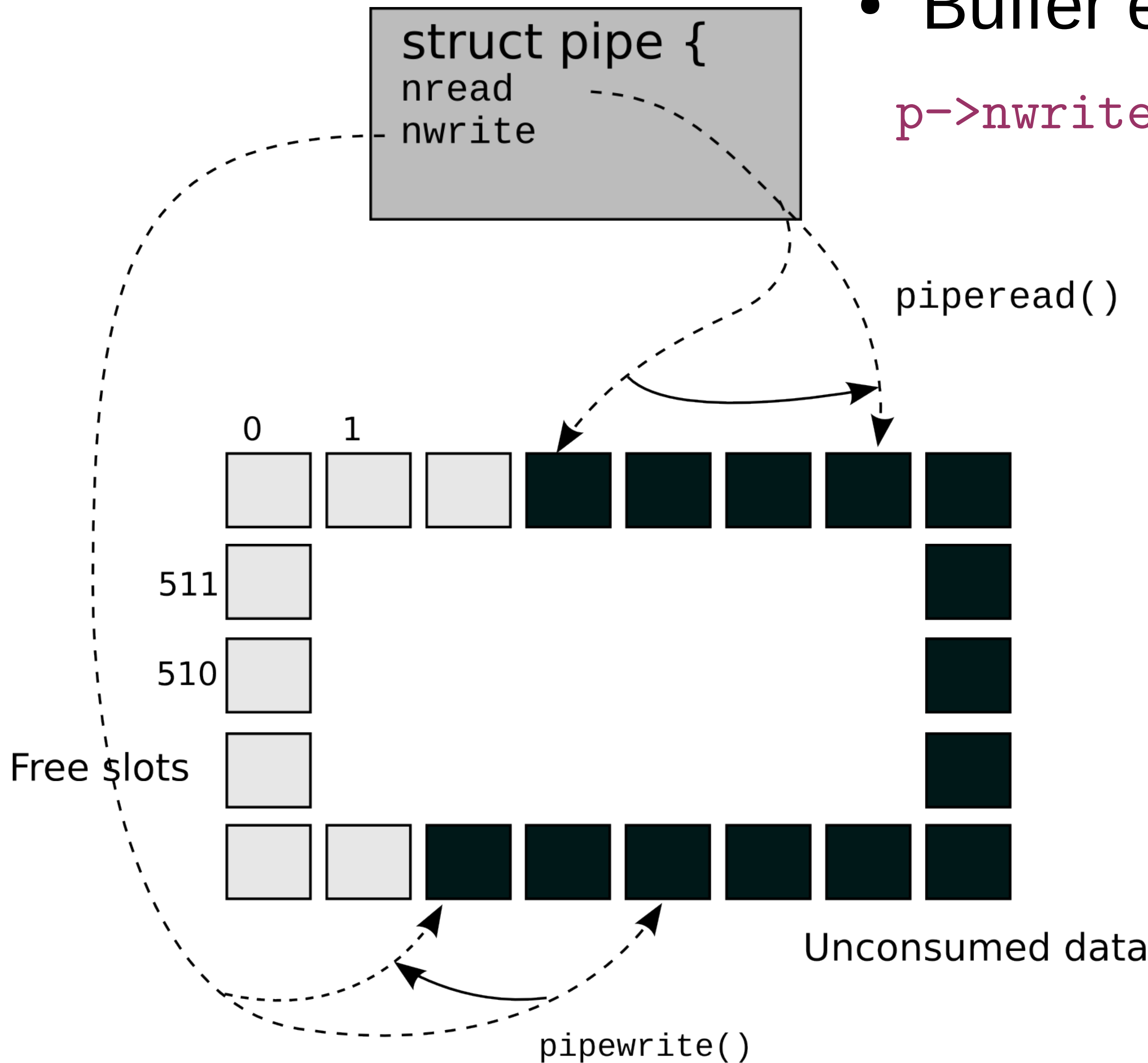
```
2853 wakeup1(void *chan)
2854 {
2855   struct proc *p;
2856
2857   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2858     if(p->state == SLEEPING && p->chan == chan)
2859       p->state = RUNNABLE;
2860 }
..
2864 wakeup(void *chan)
2865 {
2866   acquire(&ptable.lock);
2867   wakeup1(chan);
2868   release(&ptable.lock);
2869 }
```

Pipes

Pipe

```
6459 #define PIPESIZE 512
6460
6461 struct pipe {
6462   struct spinlock lock;
6463   char data[PIPESIZE];
6464   uint nread; // number of bytes read
6465   uint nwrite; // number of bytes written
6466   int readopen; // read fd is still open
6467   int writeopen; // write fd is still open
6468 };
```

Pipe buffer



- Buffer full

`p->nwrite == p->nread + PIPESIZE`

- Buffer empty

`p->nwrite == p->nread`

pipewrite()

```
6530 pipewrite(struct pipe *p, char *addr, int n)
6531 {
6532   int i;
6533
6534   acquire(&p->lock);
6535   for(i = 0; i < n; i++){
6536     while(p->nwrite == p->nread + PIPESIZE){
6537       if(p->readopen == 0 || proc->killed){
6538         release(&p->lock);
6539         return -1;
6540       }
6541       wakeup(&p->nread);
6542       sleep(&p->nwrite, &p->lock);
6543     }
6544     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6545   }
6546   wakeup(&p->nread);
6547   release(&p->lock);
6548   return n;
6549 }
```

piperead()

```
6551 piperead(struct pipe *p, char *addr, int n)
6552 {
6553   int i;
6554
6555   acquire(&p->lock);
6556   while(p->nread == p->nwrite && p->writeopen){
6557     if(proc->killed){
6558       release(&p->lock);
6559       return -1;
6560     }
6561     sleep(&p->nread, &p->lock);
6562   }
6563   for(i = 0; i < n; i++){
6564     if(p->nread == p->nwrite)
6565       break;
6566     addr[i] = p->data[p->nread++ % PIPESIZE];
6567   }
6568   wakeup(&p->nwrite);
6569   release(&p->lock);
6570   return i;
6571 }
```

Thank you!