143A: Principles of Operating Systems

Lecture 13: Synchronization

Anton Burtsev December, 2019

Starting other CPUs

```
1317 main(void)
                     Started from main()
1318 {
1336
      startothers(); // start other processors
1337 kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
1338 userinit(); // first user process
      mpmain();
1339
1340 }
```

Starting other CPUs

- Copy start code in a good location
 - 0x7000 (remember same as the one used by boot loader)
- Pass start parameters on the stack
 - Allocate a new stack for each CPU
 - Send a magic inter-processor interrupt (IPI) with the entry point (mpenter())

```
1374 startothers(void)
                                    Start other CPUs
1375 {
       code = P2V(0x7000):
1384
       memmove(code, _binary_entryother_start,
1385

    Copy start code to 0x7000

    Start code is linked into the kernel.

               (uint)_binary_entryother_size);
                                                         • binary entryother start
1386
                                                         • binary entryother size
       for(c = cpus; c < cpus+ncpu; c++){</pre>
1387
         if(c == cpus+cpunum()) // We've started already.
1388
1389
            continue:
. . .
         stack = kalloc();
1394
1395
          *(void**)(code-4) = stack + KSTACKSIZE;
1396
          *(void**)(code-8) = mpenter;
          *(int**)(code-12) = (void *) V2P(entrypgdir);
1397
1398
```

lapicstartap(c->apicid, V2P(code));

1399

```
1374 startothers(void)
```

Start other CPUs

```
1375 {
       code = P2V(0x7000):
1384
1385
       memmove(code, _binary_entryother_start,
              (uint)_binary_entryother_size);
1386
       for(c = cpus; c < cpus+ncpu; c++){</pre>
1387
1388
         if(c == cpus+cpunum()) // We've started already.
1389
           continue:
         stack = kalloc();
1394
                                                      stack?
         *(void**)(code-4) = stack + KSTACKSIZE:
1395
1396
         *(void**)(code-8) = mpenter;
         *(int**)(code-12) = (void *) V2P(entrypgdir);
1397
1398
         lapicstartap(c->apicid, V2P(code));
1399
```

 Allocate a new kernel stack for each CPU

 What will be running on this

1374 startothers(void)

Start other CPUs

```
1375 {
      code = P2V(0x7000):
1384

    Allocate a new

1385
      memmove(code, _binary_entryother_start,
                                                   kernel stack for
              (uint)_binary_entryother_size);
                                                   each CPU
1386
      for(c = cpus; c < cpus+ncpu; c++){</pre>
1387
1388
         if(c == cpus+cpunum()) // We've started already.
1389
          continue:

    What will be

                                                   running on this
         stack = kalloc();
1394
                                                   stack?
         *(void**)(code-4) = stack + KSTACKSIZE:
1395

    Scheduler

1396
         *(void**)(code-8) = mpenter;
         *(int**)(code-12) = (void *) V2P(entrypgdir);
1397
1398
         lapicstartap(c->apicid, V2P(code));
1399
```

```
1374 startothers(void)
                                 Start other CPUs
1375 {
1384 code = P2V(0x7000):
      memmove(code, _binary_entryother_start,
1385
              (uint) binary entryother size);
1386
      for(c = cpus; c < cpus+ncpu; c++){</pre>
1387
         if(c == cpus+cpunum()) // We've started already.
1388
1389
          continue:

    What is done

. . .
                                                   here?
         stack = kalloc();
1394
1395
         *(void**)(code-4) = stack + KSTACKSIZE;
1396
         *(void**)(code-8) = mpenter;
         *(int**)(code-12) = (void *) V2P(entrypgdir);
1397
1398
         lapicstartap(c->apicid, V2P(code));
1399
```

```
1374 startothers(void)
                                  Start other CPUs
1375 {
1384 code = P2V(0x7000):
1385
       memmove(code, _binary_entryother_start,
              (uint) binary entryother size);
1386
       for(c = cpus; c < cpus+ncpu; c++){</pre>
1387
         if(c == cpus+cpunum()) // We've started already.
1388
1389
           continue:
                                                   What is done here?
. . .

    Kernel stack

         stack = kalloc();
1394

    Address of mpenter()

1395
         *(void**)(code-4) = stack + KSTACKSIZE;

    Physical address of

                                                       entrypgdir
1396
         *(void**)(code-8) = mpenter;
         *(int**)(code-12) = (void *) V2P(entrypgdir);
1397
1398
         lapicstartap(c->apicid, V2P(code));
1399
```

```
1374 startothers(void)
```

Start other CPUs

```
1375 {
      code = P2V(0x7000):
1384
       memmove(code, _binary_entryother_start,
1385
              (uint) binary entryother size);
1386
       for(c = cpus; c < cpus+ncpu; c++){</pre>
1387
1388
         if(c == cpus+cpunum()) // We've started already.
1389
           continue:

    Send "magic"

                                                    interrupt
         stack = kalloc();
1394

    Wake up other

         *(void**)(code-4) = stack + KSTACKSIZE;
1395
                                                     CPUs
1396
         *(void**)(code-8) = mpenter;
         *(int**)(code-12) = (void *) V2P(entrypgdir);
1397
1398
         lapicstartap(c->apicid, V2P(code));
1399
```

```
1123 .code16
1124 .globl start
1125 start:
1126
       cli
1127
1128
       xorw %ax, %ax
       movw %ax, %ds
1129
1130
       movw %ax, %es
       movw %ax, %ss
1131
1132
```

entryother.S

- Disable interrupts
- Init segments with 0

```
1133
       lgdt gdtdesc
1134
       movl %cr0, %eax
      orl $CRO_PE, %eax
1135
      movl %eax, %cr0
1136
       ljmpl $(SEG_KCODE<<3), $(start32)</pre>
1150
1151
1152 .code32
1153 start32:
      movw $(SEG_KDATA<<3), %ax</pre>
1154
1155
      movw %ax, %ds
      movw %ax, %es
1156
      movw %ax, %ss
1157
1158
      movw $0, %ax
1159
      movw %ax, %fs
      movw %ax, %gs
1160
```

entryother.S

- Load GDT
- Switch to 32bit mode
 - Long jump to start32
- Load segments

```
1162
      # Turn on page size extension for 4Mbyte pages
1163
      movl %cr4, %eax
      orl $(CR4_PSE), %eax
1164
1165
      movl %eax, %cr4
      # Use enterpgdir as our initial page table
1166
1167
      movl (start-12), %eax
1168
      movl %eax, %cr3
1169
     # Turn on paging.
1170
      movl %cr0, %eax
      orl $(CRO_PE|CRO_PG|CRO_WP), %eax
1171
1172
      movl %eax. %cr0
1173
1174
      # Switch to the stack allocated by startothers()
1175
      movl (start-4), %esp
1176
      # Call mpenter()
                                              entryother.S
      call *(start-8)
1177
```

```
1162
      # Turn on page size extension for 4Mbyte pages
1163
      movl %cr4, %eax
      orl $(CR4_PSE), %eax
1164
1165
      movl %eax, %cr4
      # Use enterpgdir as our initial page table
1166
1167
      movl (start-12), %eax
1168
      movl %eax, %cr3
1169
     # Turn on paging.
1170
      movl %cr0, %eax
      orl $(CRO_PE|CRO_PG|CRO_WP), %eax
1171
1172
      movl %eax. %cr0
1173
1174
      # Switch to the stack allocated by startothers()
1175
      movl (start-4), %esp
1176
      # Call mpenter()
                                             entryother.S
      call *(start-8)
1177
```

```
1162
      # Turn on page size extension for 4Mbyte pages
1163
      movl %cr4, %eax
      orl $(CR4_PSE), %eax
1164
1165
      movl %eax, %cr4
1166
      # Use enterpgdir as our initial page table
      movl (start-12), %eax
1167
1168
      movl %eax, %cr3
1169
     # Turn on paging.
1170
      movl %cr0, %eax
      orl $(CRO_PE|CRO_PG|CRO_WP), %eax
1171
1172
      movl %eax, %cr0
1173
1174
      # Switch to the stack allocated by startothers()
1175
      movl (start-4), %esp
1176
      # Call mpenter()
                                             entryother.S
      call *(start-8)
1177
```

```
1162
      # Turn on page size extension for 4Mbyte pages
1163
      movl %cr4, %eax
      orl $(CR4_PSE), %eax
1164
1165
      movl %eax, %cr4
1166
      # Use enterpgdir as our initial page table
1167
      movl (start-12), %eax
1168
      movl %eax, %cr3
1169
     # Turn on paging.
1170
      movl %cr0, %eax
      orl $(CRO_PE|CRO_PG|CRO_WP), %eax
1171
      movl %eax, %cr0
1172
1173
1174
      # Switch to the stack allocated by startothers()
1175
      movl (start-4), %esp
1176
      # Call mpenter()
                                             entryother.S
      call *(start-8)
1177
```

```
1251 static void
1252 mpenter(void)
1253 {
1254
       switchkvm();
1255
       seginit();
1256
       lapicinit();
       mpmain();
1257
1258 }
```

```
1251 static void
1252 mpenter(void)
                       Init segments
1253 {
1254
       switchkvm();
1255
       seginit();
       lapicinit();
1256
1257
       mpmain();
1258 }
```

```
seginit(void)
                                    Init segments
 struct cpu *c;
 // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
 // because it would have to have DPL_USR, but the CPU forbids
 // an interrupt from CPL=0 to DPL=3.
 c = &cpus[cpuid()];
 c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
 c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
 c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
 c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
 lgdt(c->gdt, sizeof(c->gdt));
```

Per-CPU variables

Variables private to each CPU

Per-CPU variables

- Variables private to each CPU
 - Current running process
 - Kernel stack for interrupts
 - Hence, TSS that stores that stack

```
struct cpu cpus[NCPU];
```

```
// Per-CPU state
struct cpu {
                // Local APIC ID
 uchar apicid;
 struct context *scheduler; // swtch() here to enter scheduler
  struct taskstate ts; // Used by x86 to find stack for
interrupt
 struct segdesc gdt[NSEGS]; // x86 global descriptor table
 volatile uint started; // Has the CPU started?
 int ncli;
                             // Depth of pushcli nesting.
                             // Were interrupts enabled before
 int intena;
pushcli?
                            // The process running on this cpu or
 struct proc *proc;
null
};
```

extern struct cpu cpus[NCPU];

```
cpuid()
// Must be called with interrupts disabled
int cpuid() {
 return mycpu()-cpus;
struct cpu* mycpu(void)
 int apicid, i;
  if(readeflags()&FL_IF)
   panic("mycpu called with interrupts enabled\n");
 apicid = lapicid();
 // APIC IDs are not guaranteed to be contiguous. Maybe we should have
 // a reverse map, or reserve a register to store &cpus[i].
 for (i = 0; i < ncpu; ++i) {
    if (cpus[i].apicid == apicid)
     return &cpus[i];
 }
 panic("unknown apicid\n");
```

```
1250 // Common CPU setup code.
1251 static void
1252 mpmain(void)
                                       mpmain()
1253 {
1254 cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
1255 idtinit(); // load idt register
     xchg(&(mycpu()->started), 1); // tell startothers() we're up
1256
1257 scheduler(); // start running processes
1258 }
```

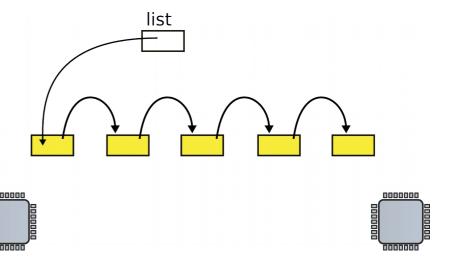
Synchronization

Race conditions

- Example:
 - Disk driver maintains a list of outstanding requests
 - Each process can add requests to the list

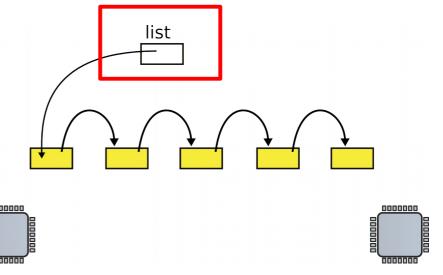
```
1 struct list {
    int data;
    struct list *next;
4 };
6 struct list *list = 0;
9 insert(int data)
10 {
11
     struct list *1;
12
13
     1 = malloc(sizeof *1);
14
     1->data = data;
15
     1->next = list;
16
     list = 1;
17 }
```

- List
 - One data element
 - Pointer to the next element



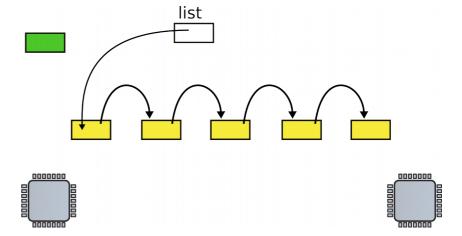
```
1 struct list {
    int data;
3
    struct list *next;
4 };
6 struct list *list = 0;
9 insert(int data)
10 {
11
     struct list *1;
12
13
     1 = malloc(sizeof *1);
     1->data = data;
14
15
     1->next = list;
16
     list = 1;
17 }
```

Global head



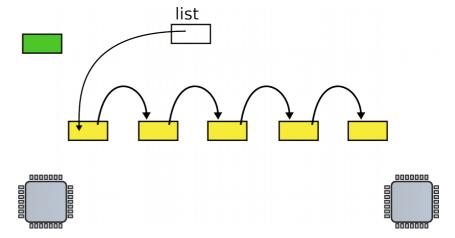
```
1 struct list {
   int data;
    struct list *next;
4 };
6 struct list *list = 0;
9 insert(int data)
10 {
11
     struct list *1;
12
     1 = malloc(sizeof *1);
13
14
     1->data = data;
15
     1->next = list;
16
     list = 1;
17 }
```

- Insertion
 - Allocate new list element



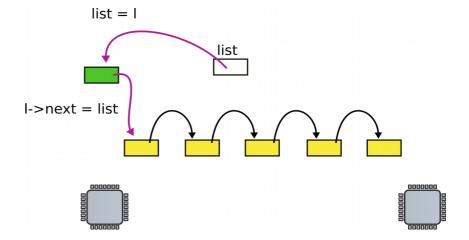
```
1 struct list {
    int data;
    struct list *next;
4 };
6 struct list *list = 0;
9 insert(int data)
10 {
11
     struct list *1:
12
13
     1 = malloc(sizeof *1);
14
     1->data = data;
     1->next = list;
15
16
     list = 1;
17 }
```

- Insertion
 - Allocate new list element
 - Save data into that element



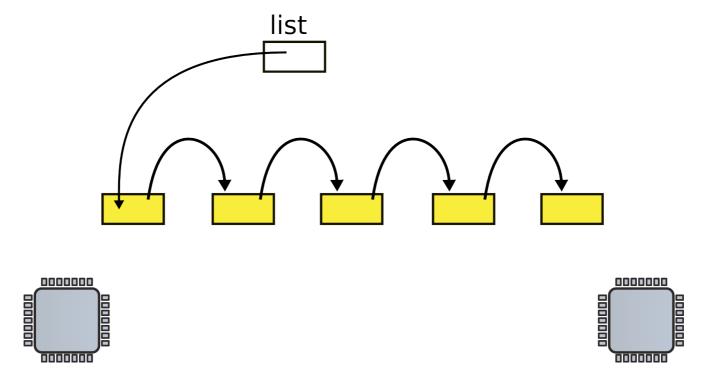
```
1 struct list {
    int data;
3
    struct list *next;
4 };
6 struct list *list = 0;
9 insert(int data)
10 {
11
     struct list *1:
12
13
     1 = malloc(sizeof *1);
14
     1->data = data;
15
     l->next = list;
16
     list = 1;
```

- Insertion
 - Allocate new list element
 - Save data into that element
 - Insert into the list



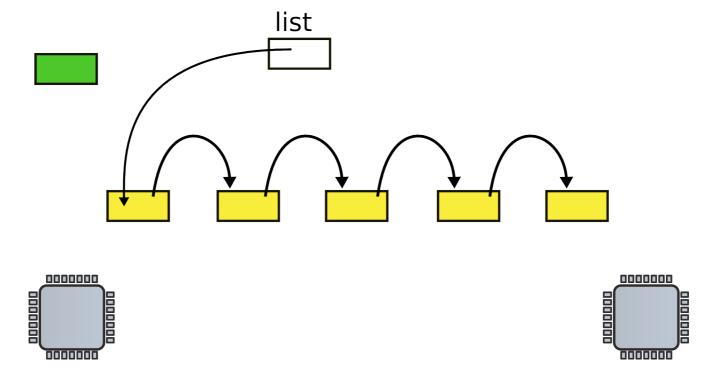
Now what happens when two CPUs access the same list

Request queue (e.g. pending disk requests)

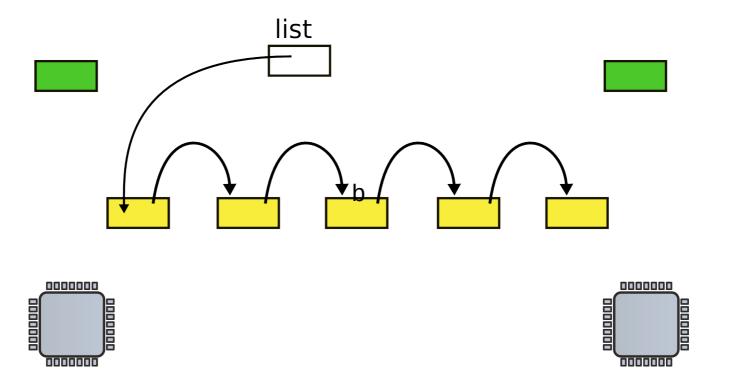


 Linked list, list is pointer to the first element

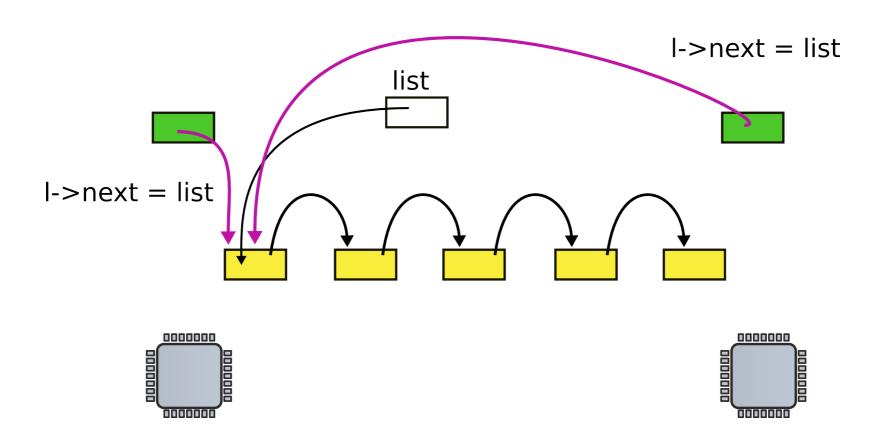
CPU1 allocates new request



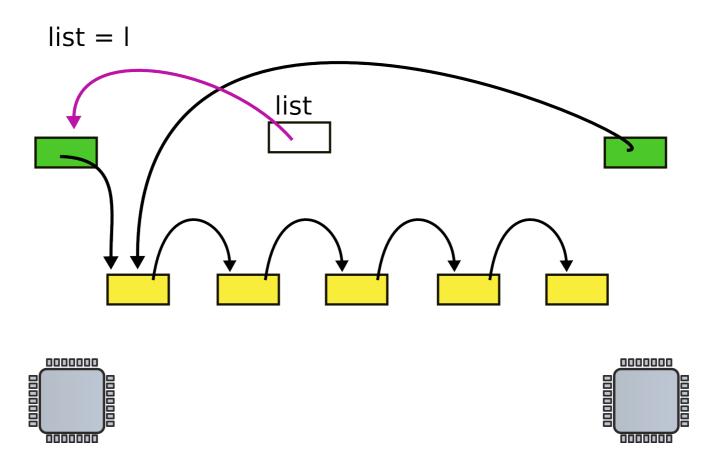
CPU2 allocates new request



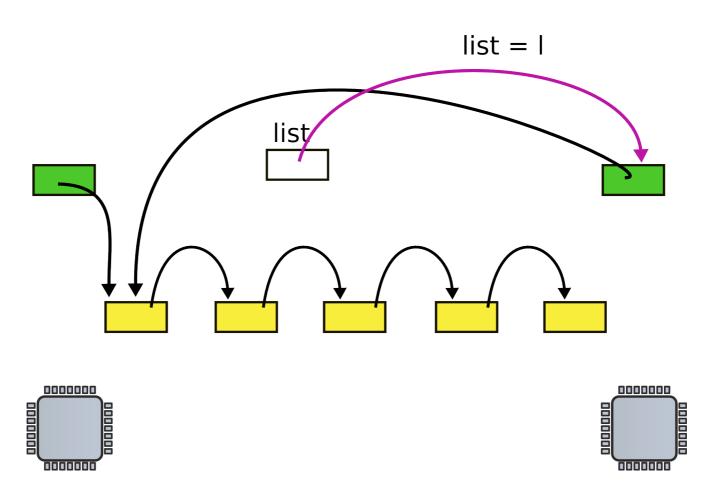
CPUs 1 and 2 update next pointer



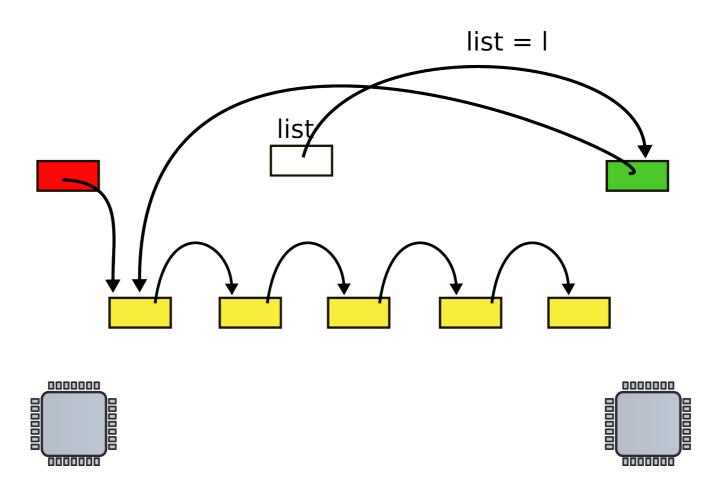
CPU1 updates head pointer



CPU2 updates head pointer



State after the race (red element is lost)



Mutual exclusion

Only one CPU can update list at a time

```
1 struct list {
   int data;
   struct list *next;
4 };
6 struct list *list = 0;
  struct lock listlock;
9 insert(int data)
10 {
11
   struct list *1;
    1 = malloc(sizeof *1);
13
     acquire(&listlock);
14
    1->data = data;
15
    1->next = list;
    list = 1;
16
     release(&listlock);
17 }
```

List implementation with locks

Critical section

How can we implement acquire()?

Spinlock

```
21 void
22 acquire(struct spinlock *lk)

    Spin until lock is 0

23 {
24 for(;;) {
                                 • Set it to 1
25
       if(!lk->locked) {
26
         lk \rightarrow locked = 1;
         break;
27
28
29 }
30 }
```

Still incorrect

```
21 void
22 acquire(struct spinlock *lk)
23 {
24 for(;;) {
25
       if(!lk->locked) {
26
         lk \rightarrow 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 }
```

xchgl instruction

```
0568 static inline uint
0569 xchg(volatile uint *addr, uint newval)
0570 {
0571 uint result;
0572
0573 // The + in "+m" denotes a read-modify-write
          operand.
     asm volatile("lock; xchgl %0, %1" :
0574
                    "+m" (*addr), "=a" (result) :
0575
                    "1" (newval) :
0576
                    "cc");
0577
0578 return result;
0579 }
```

Correct implementation

```
1573 void
1574 acquire(struct spinlock *lk)
1575 {
. . .
1580 // The xchg is atomic.
      while(xchg(&lk->locked, 1) != 0)
1581
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

```
Network
                        interrupt
                                     network_packet(){
network_packet(){
                                        insert() {
  insert() {
                                          acquire(A)
     acquire(A)
                     0000000
```

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))
        panic("acquire");
1578
1580 // The xchg is atomic.
      while(xchg(&lk->locked, 1) != 0)
1581
1582
1587 __sync_synchronize();
                          Disabling interrupts
1592 }
```

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)
                         Pushcli()/popcli()
1656 {
1657
      int eflags;
1658
       eflags = readeflags();
1659
1660 cli();
       if(cpu->ncli == 0)
1661
1662
         cpu->intena = eflags & FL_IF;
1663
       cpu->ncli += 1;
1664 }
```

```
1667 popcli(void)
                       Pushcli()/popcli()
1668 {
       if(readeflags()&FL IF)
1669
         panic("popcli - interruptible");
1670
1671
       if(--cpu->ncli < 0)
         panic("popcli");
1672
       if(cpu->ncli == 0 && cpu->intena)
1673
         sti();
1674
1675 }
```



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

Sends one pointer between two CPUs

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

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

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

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

Sleep and wakeup

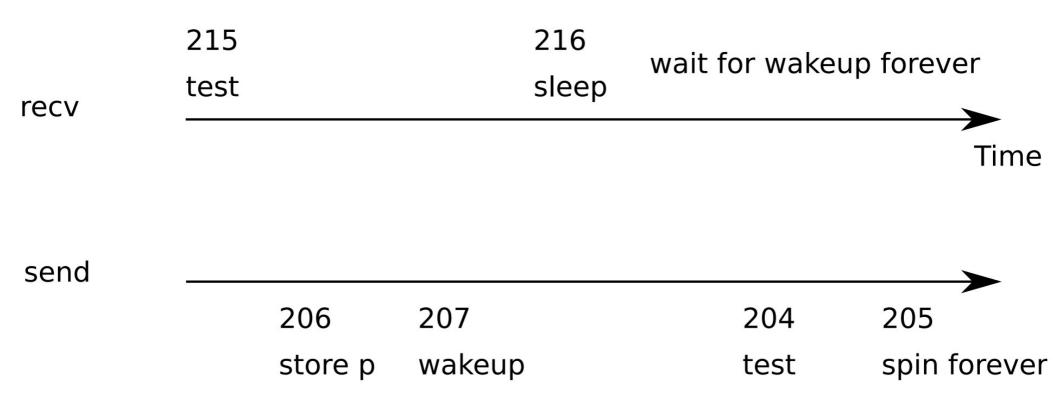
- sleep(channel)
 - Put calling process to sleep
 - Release CPU for other work
- wakeup(channel)
 - Wakes all processes sleeping on a channel
 - If any
 - i.e., causes sleep() calls to return

```
210 void*
201 void*
                                 211 recv(struct q *q)
202 send(struct q *q, void *p)
                                 212 {
203 {
                                 213 void *p;
    while(q->ptr != 0)
204
                                 214
205
                                      while((p = q->ptr) == 0)
                                 215
206 	 q->ptr = p;
                                 216
                                        sleep(q);
      wakeup(q); /*wake recv*/
207
                                 217 	 q->ptr = 0;
208 }
                                       return p;
                                 218
                                 219 }
```

```
210 void*
201 void*
                                  211 recv(struct q *q)
202 send(struct q *q, void *p)
                                   212 {
203 {
                                   213 void *p;
    while(q->ptr != 0)
204
                                   214
205 ;
                                   215 \quad \text{while}((p = q - > ptr) == 0)
206 	 q->ptr = p;
                                   216 sleep(q);
207 wakeup(q); /*wake recv*/
                                   217 	 q->ptr = 0;
208 }
                                   218
                                         return p;
                                   219 }
```

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

Lost wakeup problem



```
300 struct q {
                                  Lock the queue
    struct spinlock lock;
301
302 void *ptr;
                                  316 void*
303 };
                                  317 recv(struct q *q)
304
                                  318 {
305 void*
306 send(struct q *q, void *p)
                                  319 void *p;
307 {
                                  320
308
     acquire(&q->lock);
                                         acquire(&q->lock);
                                  321
   while(q->ptr != 0)
309
                                  322
                                         while((p = q->ptr) == 0)
310
                                  323
                                           sleep(q);
311
    q-ptr = p;
                                  324
                                        q \rightarrow ptr = 0;
312
    wakeup(q);
                                         release(&q->lock);
                                  325
      release(&q->lock);
313
                                  326
                                         return p;
314 }
                                  327 }
```

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

Pass lock inside 300 struct q { 301 struct spinlock lock; sleep() 302 void *ptr; 316 void* 303 }; 317 recv(struct q *q) 304 318 { 305 void* 306 send(struct q *q, void *p) 319 void *p; 307 { 320 acquire(&q->lock); 308 acquire(&q->lock); 321 while(q->ptr != 0) 309 322 while((p = q->ptr) == 0)310 323 sleep(q, &q->lock); 311 q-ptr = p; 324 $q \rightarrow ptr = 0;$ 312 wakeup(q); release(&q->lock); 325 release(&q->lock); 313 326 return p; 314 } 327 }

```
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()

- 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()

- Acquire ptable.lock
 - All process operations are protected with ptable.lock
- Release 1k
 - Why is it safe?

```
2809 sleep(void *chan, struct spinlock *lk)
2810 {
. . .
2823
       if(lk != &ptable.lock){
2824
         acquire(&ptable.lock);
         release(lk):
2825
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()

- Acquire ptable.lock
 - All process operations are protected with ptable.lock
- Release 1k
 - 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
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
2858
2859
         p->state = RUNNABLE;
2860 }
2864 wakeup(void *chan)
2865 {
     acquire(&ptable.lock);
2866
     wakeup1(chan);
2867
      release(&ptable.lock);
2868
2869 }
```

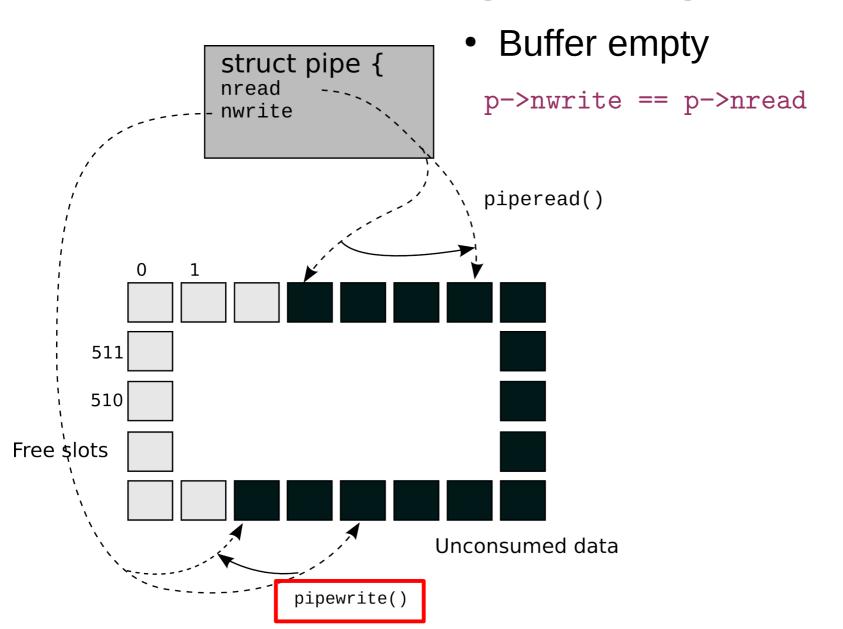
Pipes

```
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
      int writeopen; // write fd is still open
6467
6468 };
```

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

Buffer full

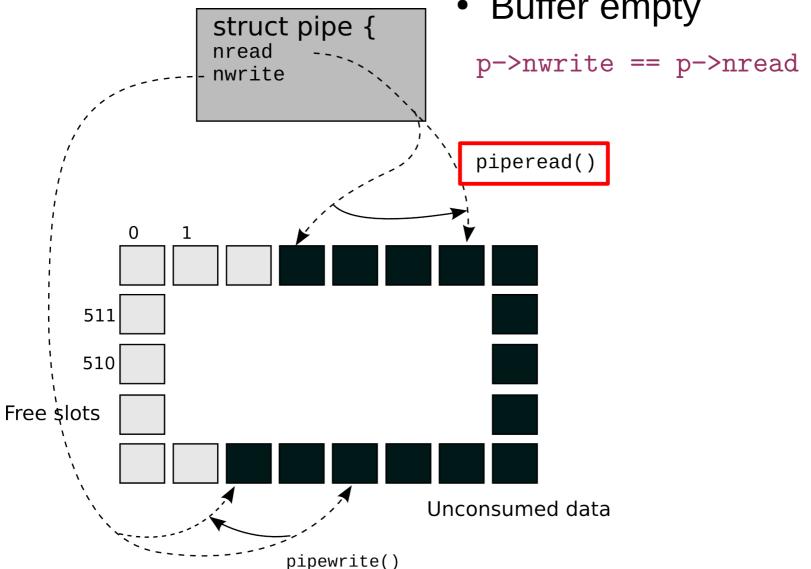
```
p->nwrite == p->nread + PIPESIZE
```



Buffer full

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

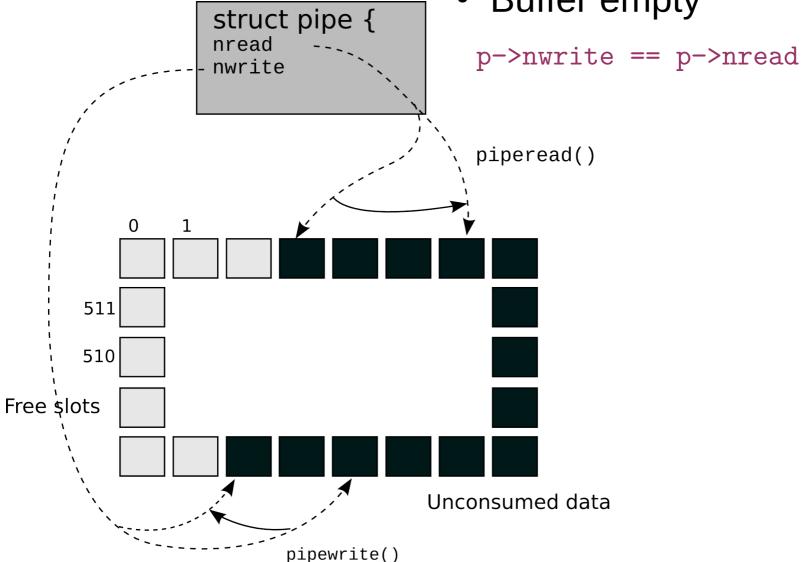




Buffer full

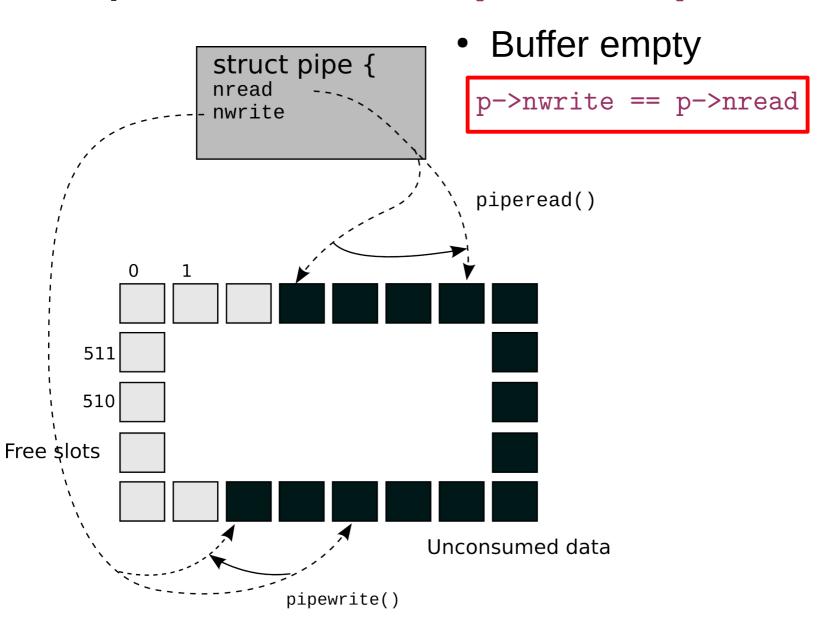
p->nwrite == p->nread + PIPESIZE

Buffer empty



Buffer full

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



```
6551 piperead(struct pipe *p, char *addr, int n)
6552 {
6553
       int i;
6554
       acquire(&p->lock);
6555
6556
       while(p->nread == p->nwrite && p->writeopen){
6557
         if(proc->killed){
           release(&p->lock);
6558
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 }
```

piperead()

- Acquire pipe lock
 - All pipe operations are are protected with the lock

```
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){
           release(&p->lock);
6558
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);
       release(&p->lock);
6569
6570
       return i;
6571 }
```

piperead()

- If the buffer is empty && the write end is still open
 - Go to sleep

```
6551 piperead(struct pipe *p, char *addr, int n)
6552 {
6553
       int i;
6554
       acquire(&p->lock);
6555
6556
       while(p->nread == p->nwrite && p->writeopen){
6557
         if(proc->killed){
           release(&p->lock);
6558
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
       wakeup(&p->nwrite);
6568
       release(&p->lock);
6569
6570
       return i;
6571 }
```

piperead()

- After reading some data from the buffer
 - Wakeup the writer

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

pipewrite()

- If the buffer is full
 - Wakeup reader
 - Go to sleep

```
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
           }
           wakeup(&p->nread);
6541
6542
           sleep(&p->nwrite, &p->lock);
6543
         p->data[p->nwrite++ % PIPESIZE] = addr[i];
6544
6545
       }
6546
       wakeup(&p->nread);
6547
       release(&p->lock);
6548
       return n;
6549 }
```

pipewrite()

- If the buffer is full
 - Wakeup reader
 - Go to sleep
- However if the read end is closed
 - Return an error
 - (-1)

```
6530 pipewrite(struct pipe *p, char *addr, int n)
6531 {
6532
       int i;
6533
6534
       acquire(&p->lock);
       for(i = 0; i < n; i++){
6535
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
         p->data[p->nwrite++ % PIPESIZE] = addr[i];
6544
6545
6546
       wakeup(&p->nread);
       release(&p->lock);
6547
6548
       return n;
6549 }
```

pipewrite()

- Otherwise keep writing bytes into the pipe
- When done
 - Wakeup reader

Thank you!