238P: Operating Systems

Lecture 11: Synchronization

Anton Burtsev
November, 2018

Starting other CPUs

Started from main()

```
1317 main(void)
1318 {

...

1336    startothers(); // start other processors

1337    kinit2(P2V(4*1024*1024), P2V(PHYSTOP));

1338    userinit(); // first user process

1339    mpmain();

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())

Start other CPUs

```
1374  startothers(void)
1375  {
1384    code = P2V(0x7000);
1385    memmove(code, _binary_entryother_start,
                (uint)_binary_entryother_size);
1386
1387    for(c = cpus; c < cpus+ncpu; c++){
1388      if(c == cpus+cpunum()) // We've started already.
1389        continue;
...
1394      stack = kalloc();
1395      *(void**)(code-4) = stack + KSTACKSIZE;
1396      *(void**)(code-8) = mpenter;
1397      *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399      lapicstartap(c->apicid, V2P(code));
```

- Copy start code to 0x7000
- Start code is linked into the kernel
  - `_binary_entryother_start`
  - `_binary_entryother_size`

Start other CPUs

```
1374 startothers(void)
1375 {
1384   code = P2V(0x7000);
1385   memmove(code, _binary_entryother_start,
            (uint)_binary_entryother_size);
1386
1387   for(c = cpus; c < cpus+ncpu; c++){
1388     if(c == cpus+cpunum()) // We've started already.
1389       continue;
...
1394   stack = kalloc();
1395   *(void**)(code-4) = stack + KSTACKSIZE;
1396   *(void**)(code-8) = mpenter;
1397   *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399   lapicstartap(c->apicid, V2P(code));
```

- Allocate a new kernel stack for each CPU

- What will be running on this stack?

Start other CPUs

```
1374 startothers(void)
1375 {
1384   code = P2V(0x7000);
1385   memmove(code, _binary_entryother_start,
              (uint)_binary_entryother_size);
1386
1387   for(c = cpus; c < cpus+ncpu; c++){
1388     if(c == cpus+cpunum()) // We've started already.
1389       continue;
...
1394     stack = kalloc();
1395     *(void**)(code-4) = stack + KSTACKSIZE;
1396     *(void**)(code-8) = mpenter;
1397     *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399     lapicstartap(c->apicid, V2P(code));
```

- Allocate a new kernel stack for each CPU

- What will be running on this stack?
  - Scheduler

Start other CPUs

```
1374 startothers(void)
1375 {
1384   code = P2V(0x7000);
1385   memmove(code, _binary_entryother_start,
              (uint)_binary_entryother_size);
1386
1387   for(c = cpus; c < cpus+ncpu; c++){
1388     if(c == cpus+cpunum()) // We've started already.
1389       continue;
...
1394     stack = kalloc();
1395     *(void**)(code-4) = stack + KSTACKSIZE;
1396     *(void**)(code-8) = mpenter;
1397     *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399     lapicstartap(c->apicid, V2P(code));
```

- What is done here?

Start other CPUs

```
1374 startothers(void)
1375 {
1384    code = P2V(0x7000);
1385    memmove(code, _binary_entryother_start,
                (uint)_binary_entryother_size);
1386
1387    for(c = cpus; c < cpus+ncpu; c++){
1388      if(c == cpus+cpunum()) // We've started already.
1389        continue;
...
1394      stack = kalloc();
1395      *(void**)(code-4) = stack + KSTACKSIZE;
1396      *(void**)(code-8) = mpenter;
1397      *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399      lapicstartap(c->apicid, V2P(code));
```

- What is done here?
  - Kernel stack
  - Address of mpenter()
  - Physical address of entrypgdir

Start other CPUs

```
1374 startothers(void)
1375 {
1384   code = P2V(0x7000);
1385   memmove(code, _binary_entryother_start,
              (uint)_binary_entryother_size);
1386
1387   for(c = cpus; c < cpus+ncpu; c++){
1388     if(c == cpus+cpunum()) // We've started already.
1389       continue;
...
1394   stack = kalloc();
1395   *(void**)(code-4) = stack + KSTACKSIZE;
1396   *(void**)(code-8) = mpenter;
1397   *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399   lapicstartap(c->apicid, V2P(code));
```

- Send "magic" interrupt
- Wake up other CPUs

entryother.S

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

- Disable interrupts
- Init segments with 0

entryother.S

```asm
1133    lgdt gdtdesc
1134    movl %cr0, %eax
1135    orl $CR0_PE, %eax
1136    movl %eax, %cr0
1150    ljmpl $(SEG_KCODE<<3), $(start32)
1151
1152 .code32
1153 start32:
1154    movw $(SEG_KDATA<<3), %ax
1155    movw %ax, %ds
1156    movw %ax, %es
1157    movw %ax, %ss
1158    movw $0, %ax
1159    movw %ax, %fs
1160    movw %ax, %gs
```

- 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
1164    orl $(CR4_PSE), %eax
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
1171    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
1172    movl %eax, %cr0
1173
1174    # Switch to the stack allocated by startothers()
1175    movl (start-4), %esp
1176    # Call mpenter()
1177    call *(start-8)
```

entryother.S

```
1162    # Turn on page size extension for 4Mbyte pages
1163    movl %cr4, %eax
1164    orl $(CR4_PSE), %eax
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
1171    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
1172    movl %eax, %cr0
1173
1174    # Switch to the stack allocated by startothers()
1175    movl (start-4), %esp
1176    # Call mpenter()
1177    call *(start-8)
```

entryother.S

```
1162    # Turn on page size extension for 4Mbyte pages
1163    movl %cr4, %eax
1164    orl $(CR4_PSE), %eax
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
1171    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
1172    movl %eax, %cr0
1173
1174    # Switch to the stack allocated by startothers()
1175    movl (start-4), %esp
1176    # Call mpenter()
1177    call *(start-8)
```

entryother.S

```
1162    # Turn on page size extension for 4Mbyte pages
1163    movl %cr4, %eax
1164    orl $(CR4_PSE), %eax
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
1171    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
1172    movl %eax, %cr0
1173
1174    # Switch to the stack allocated by startothers()
1175    movl (start-4), %esp
1176    # Call mpenter()
1177    call *(start-8)
```

entryother.S

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

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

Init segments

Init segments

```
seginit(void)
{
  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];
```

```c
// Per-CPU state
struct cpu {
  uchar apicid;                   // Local APIC ID
  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.
  int intena;                     // Were interrupts enabled before pushcli?
  struct proc *proc;              // The process running on this cpu or null
};

extern struct cpu cpus[NCPU];
```

cpuid()

```c
// 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");
}
```
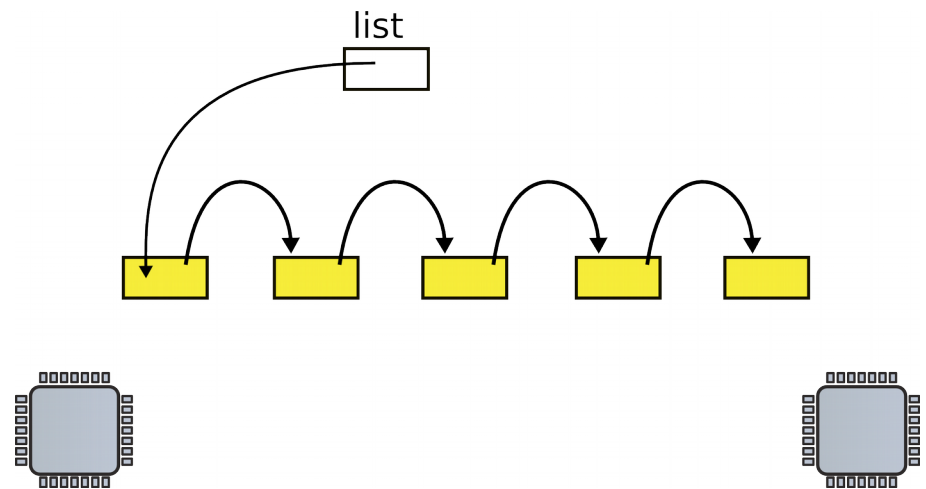
Synchronization

Race conditions

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

List implementation (no locks)

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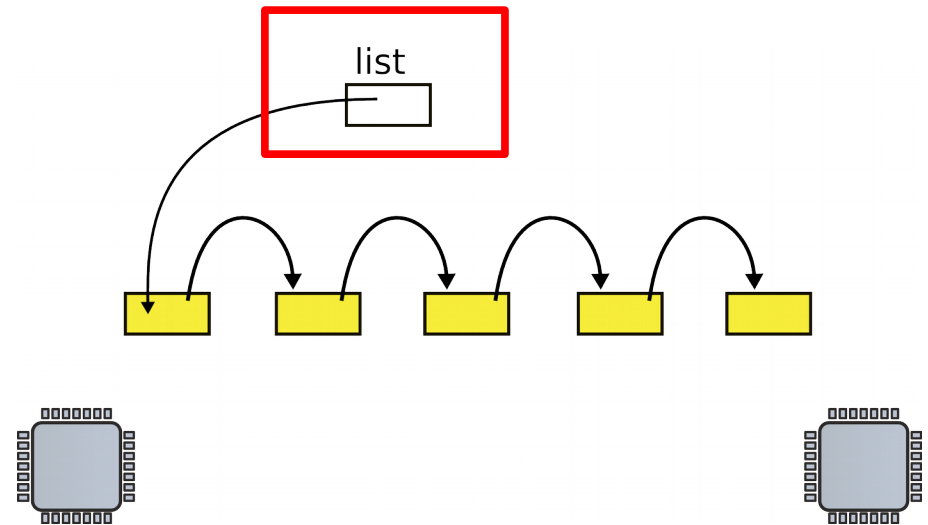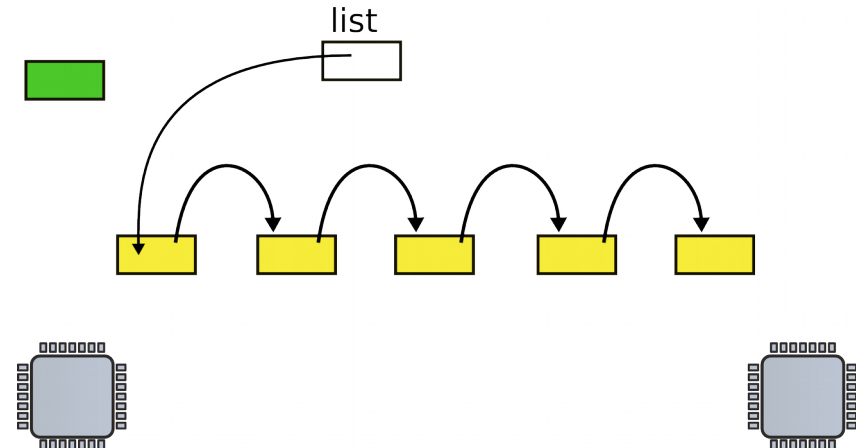
- List
  - One data element
  - Pointer to the next element

List implementation (no locks)

```
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 }
```
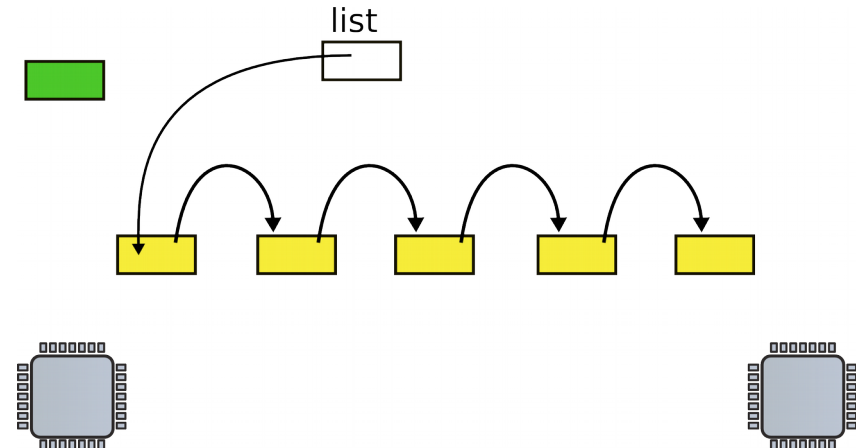
- Global head

List implementation (no locks)

```
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 }
```

- Insertion
  - Allocate new list element

List implementation (no locks)

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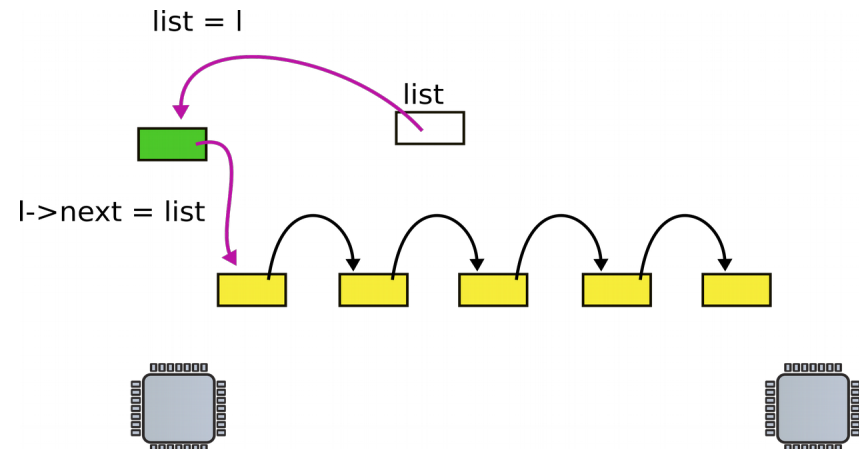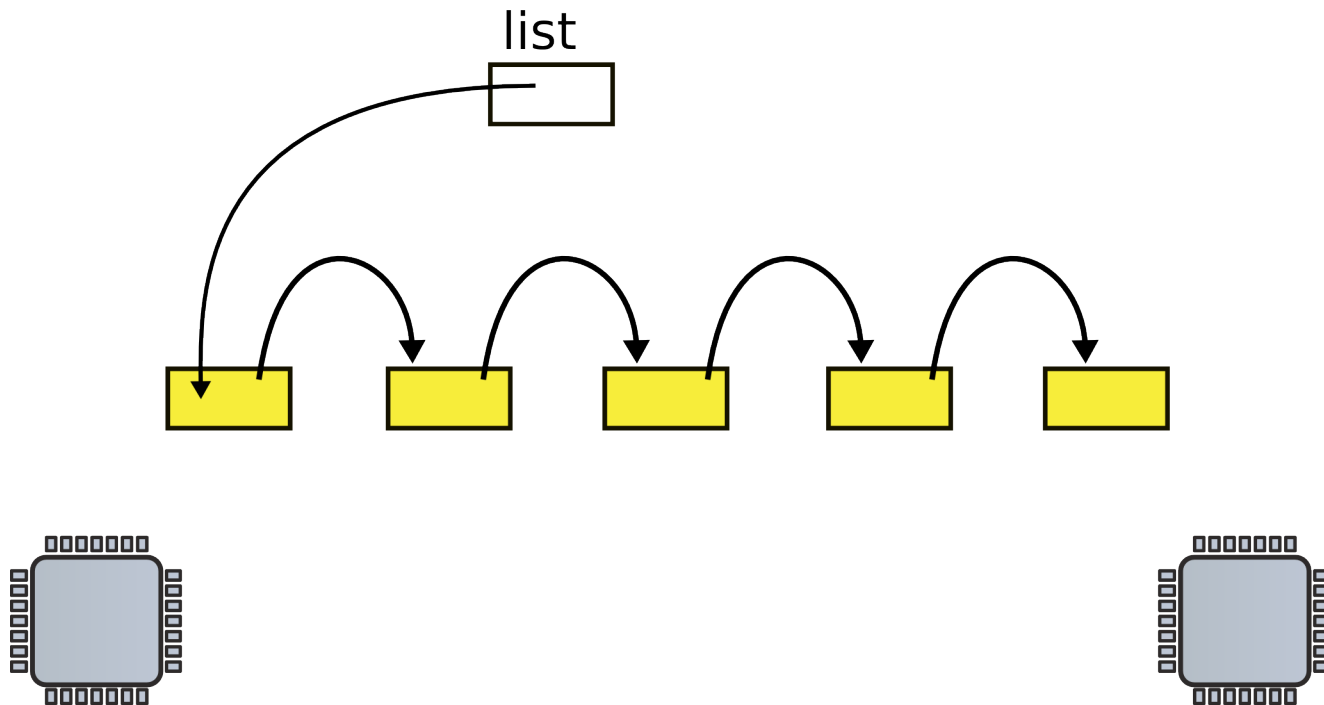
- Insertion
  - Allocate new list element
  - Save data into that element

List implementation (no locks)

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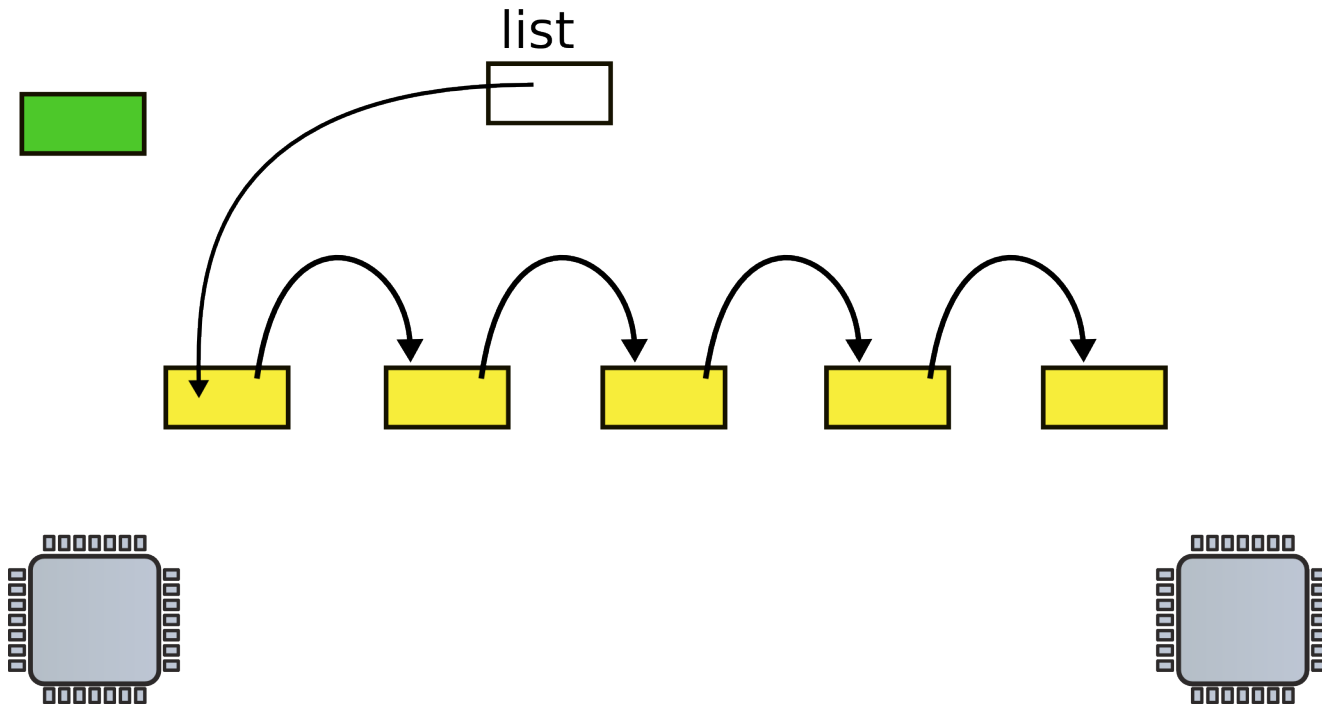
- Insertion
  - Allocate new list element
  - Save data into that element
  - Insert into the list

list = l

list

l->next = list

Now what happens when two CPUs access the same list

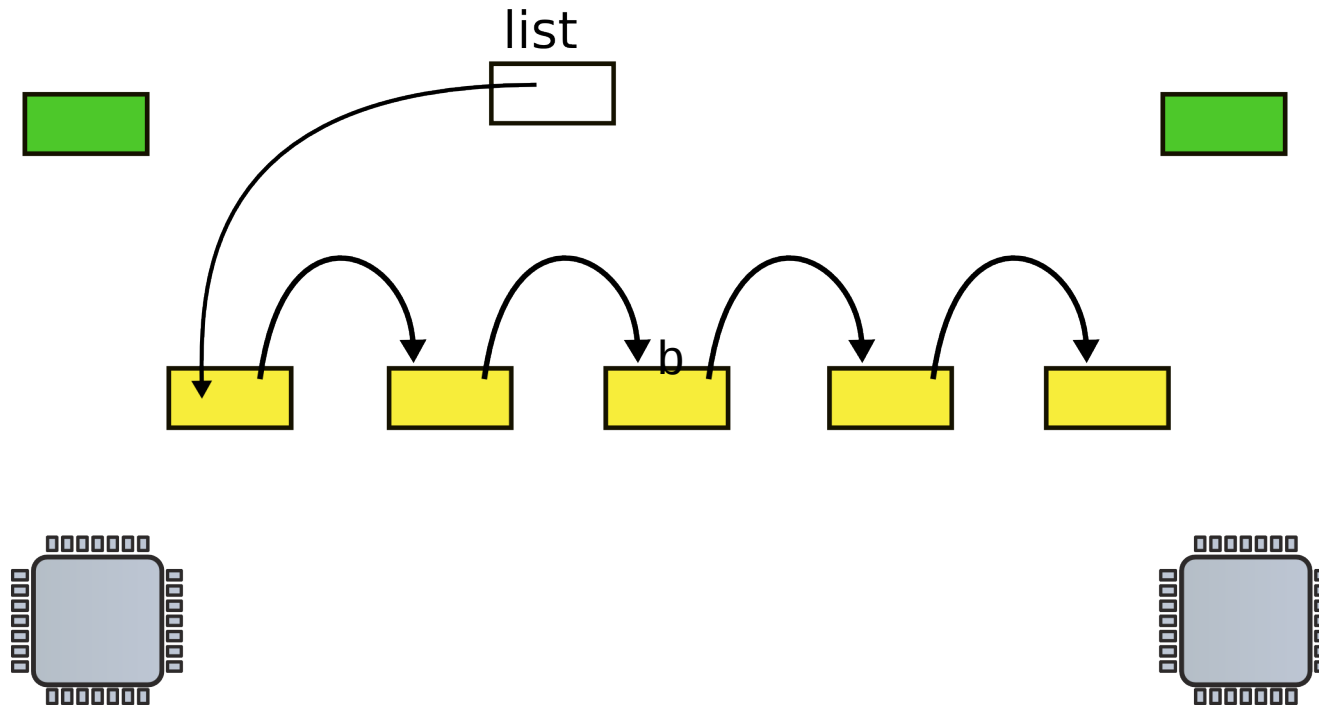# Request queue (e.g. pending disk requests)

list

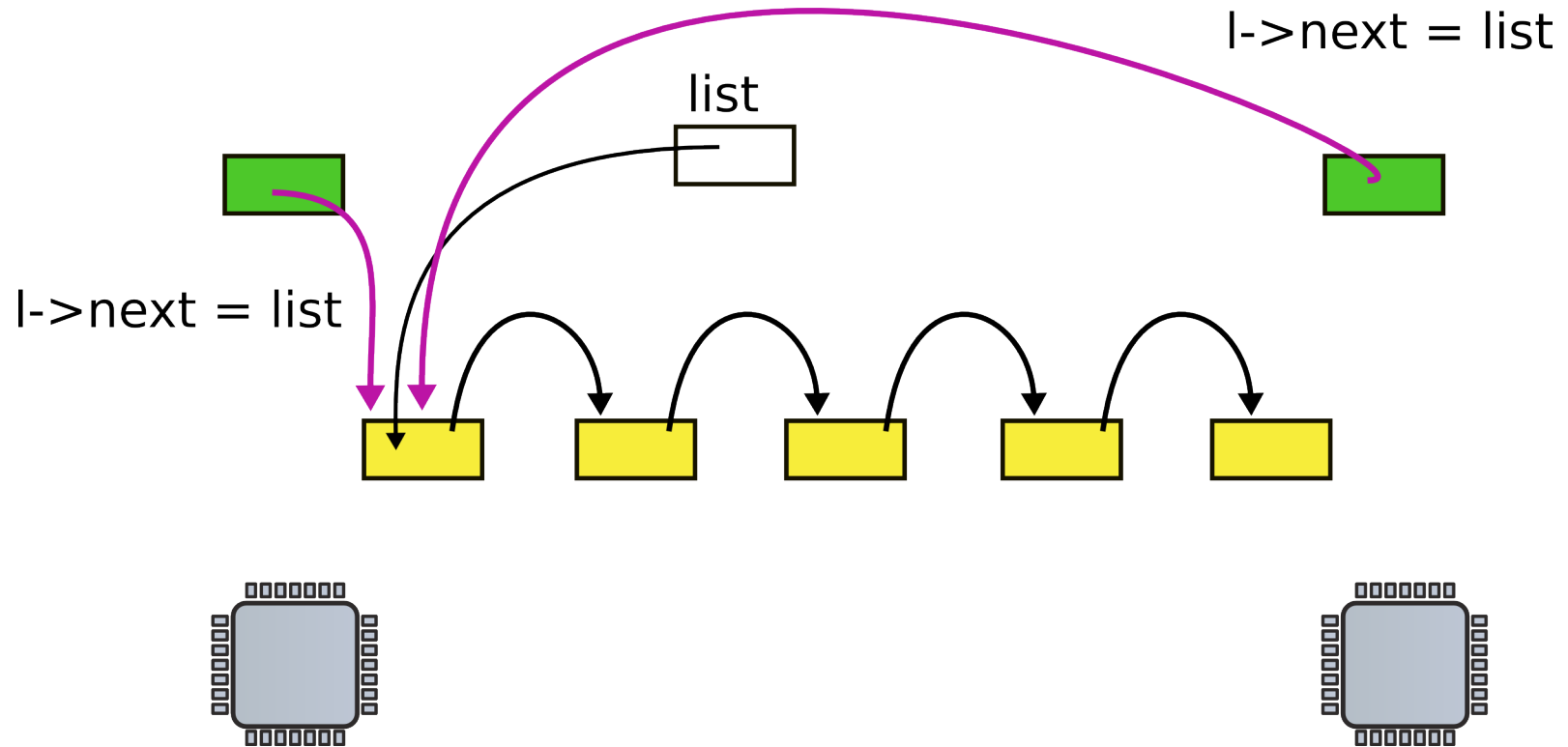- Linked list, list is pointer to the first element

CPU1 allocates new request

list

CPU2 allocates new request

list

b

CPUs 1 and 2 update next pointer

l->next = list

list

l->next = list

CPU1 updates head pointer

list = l

list

CPU2 updates head pointer

list = l

list

State after the race
(red element is lost)

list = l

list

Mutual exclusion

- Only one CPU can update list at a time

List implementation with locks

```
1 struct list {
2   int data;
3   struct list *next;
4 };

6 struct list *list = 0;

  struct lock listlock;

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

- How can we implement acquire()?

Spinlock

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

- Spin until lock is 0
- Set it to 1

Still incorrect

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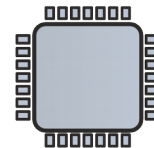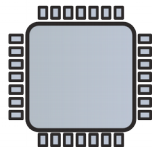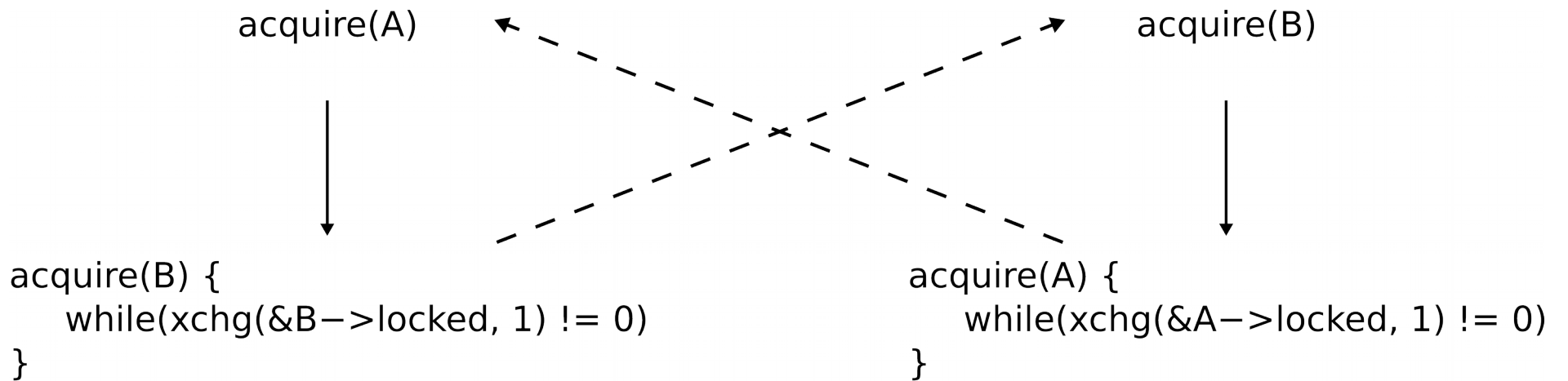
xchgl instruction

```
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.
0574   asm volatile("lock; xchgl %0, %1" :
0575                "+m" (*addr), "=a" (result) :
0576                "1" (newval) :
0577                "cc");
0578   return result;
0579 }
```

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

Network
interrupt

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

}

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

}

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 }
```

Pushcli()/popcli()

```
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 }
```

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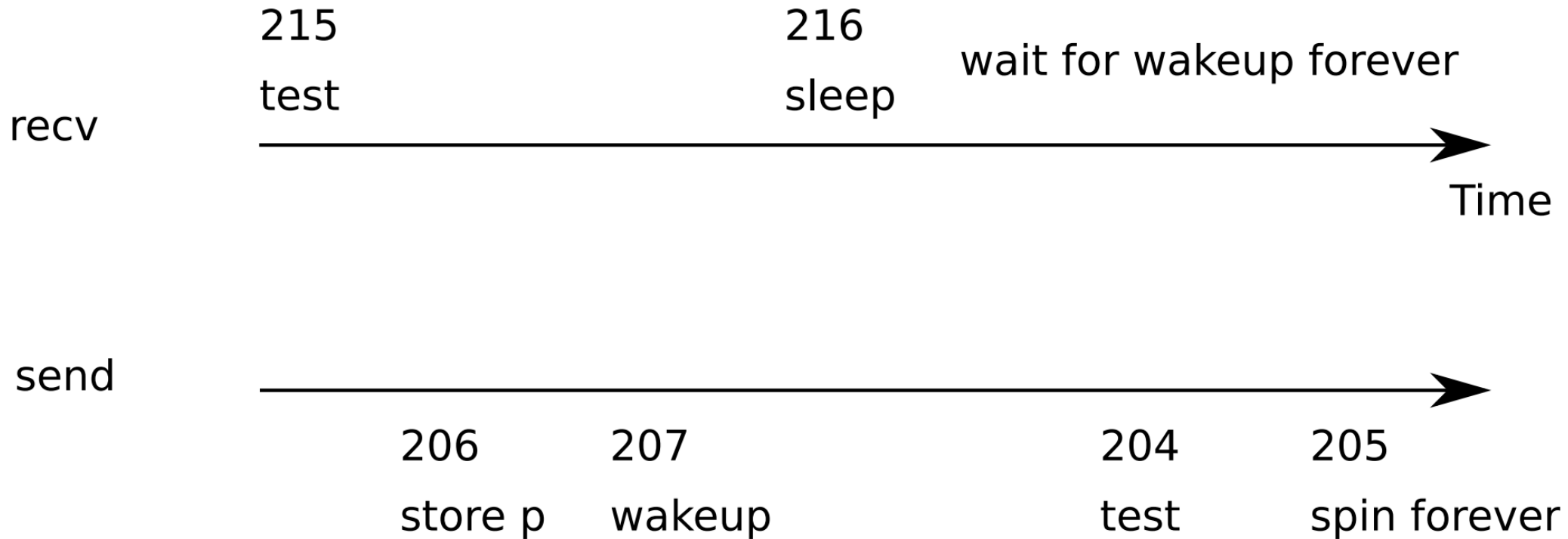
- recv() gives up the CPU to other processes
  - But there is a problem...

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()

```
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

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

```
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 };
```
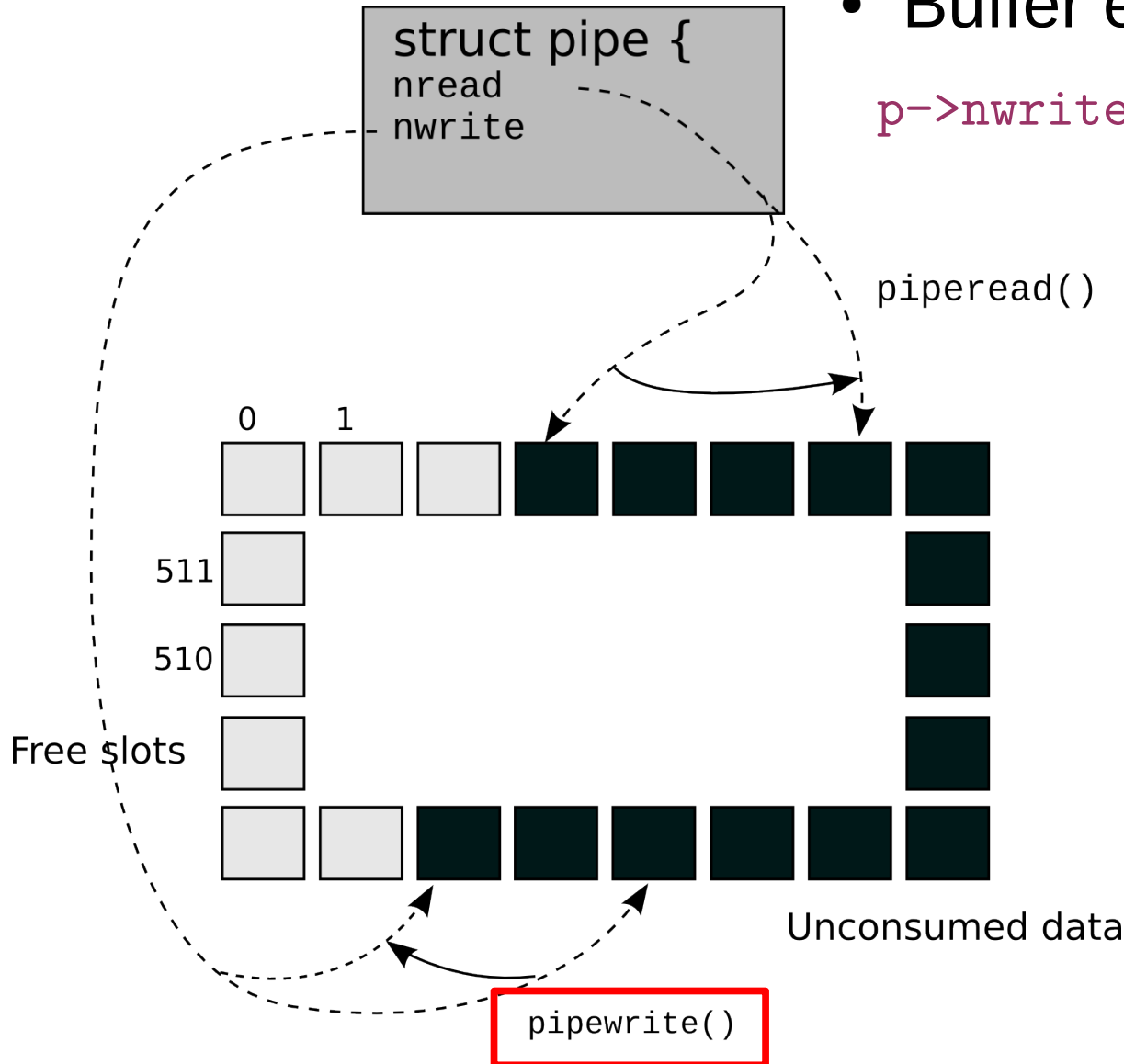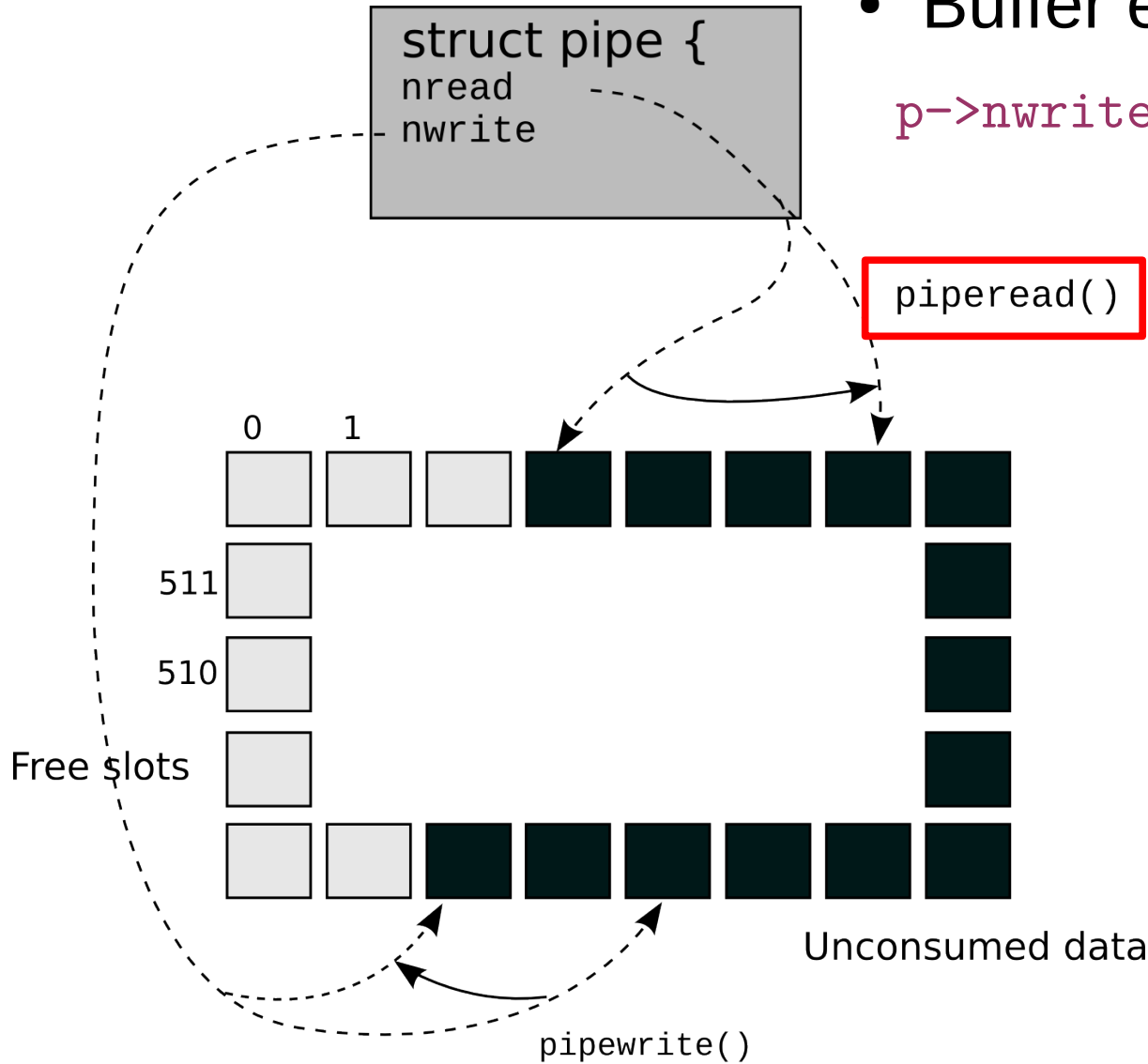
Pipe buffer

- **Buffer full**

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

- **Buffer empty**

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

Pipe buffer

- ## Buffer full

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

- ## Buffer empty

  p->nwrite == p->nread

struct pipe {
nread
nwrite

piperead()

0   1

511

510

Free slots

Unconsumed data

pipewrite()

Pipe buffer

- Buffer full

$$p\text{->}nwrite == p\text{->}nread + PIPESIZE$$

- Buffer empty

$$p\text{->}nwrite == p\text{->}nread$$

```
struct pipe {
nread
nwrite
```

piperead()

0   1

511

510

Free slots

Unconsumed data

pipewrite()

Pipe buffer

- Buffer full

$$p\text{->}nwrite == p\text{->}nread + PIPESIZE$$

- Buffer empty

$$\boxed{p\text{->}nwrite == p\text{->}nread}$$

struct pipe {
nread
nwrite

piperead()

0   1

511

510

Free slots

Unconsumed data

pipewrite()

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 }
```

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

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 }
```

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

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 }
```

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

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 }
```

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

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 }
```

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

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 }
```

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

Thank you!