ICS143A: Principles of Operating Systems

Lecture 15: Locking

Anton Burtsev
November, 2017

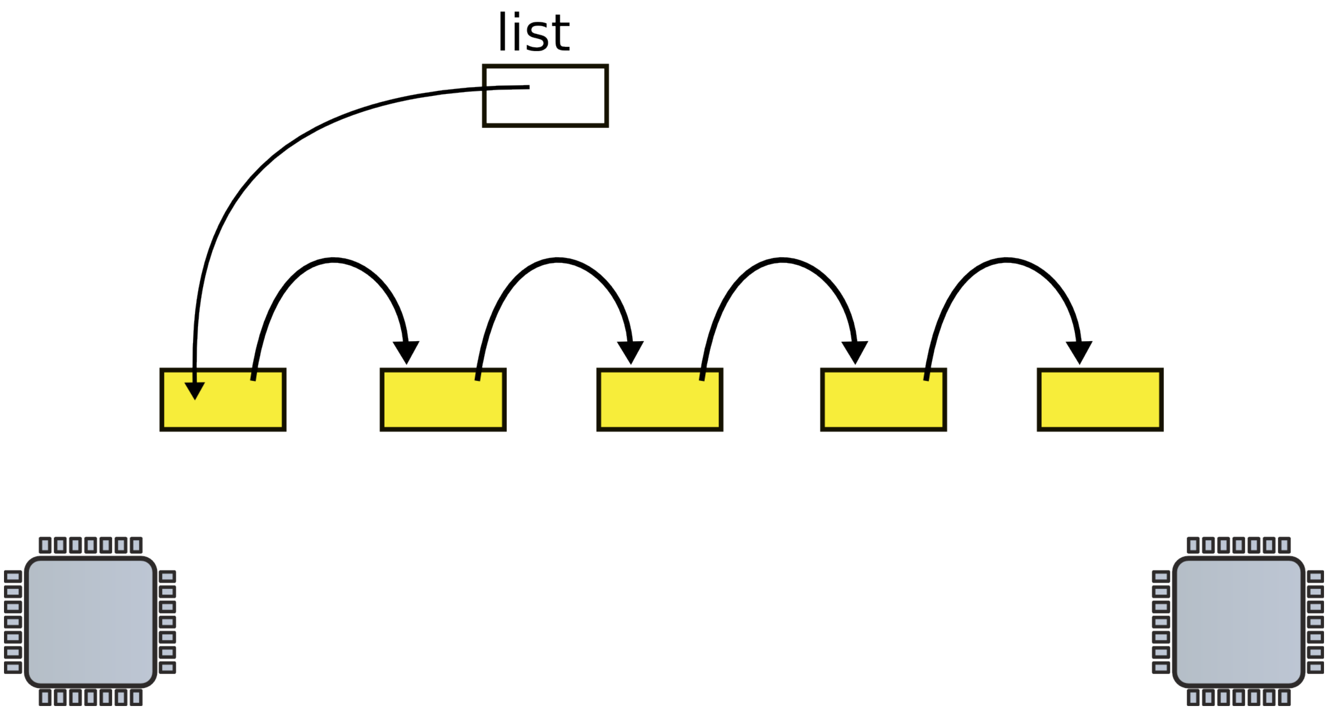
Race conditions

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

List implementation no locks

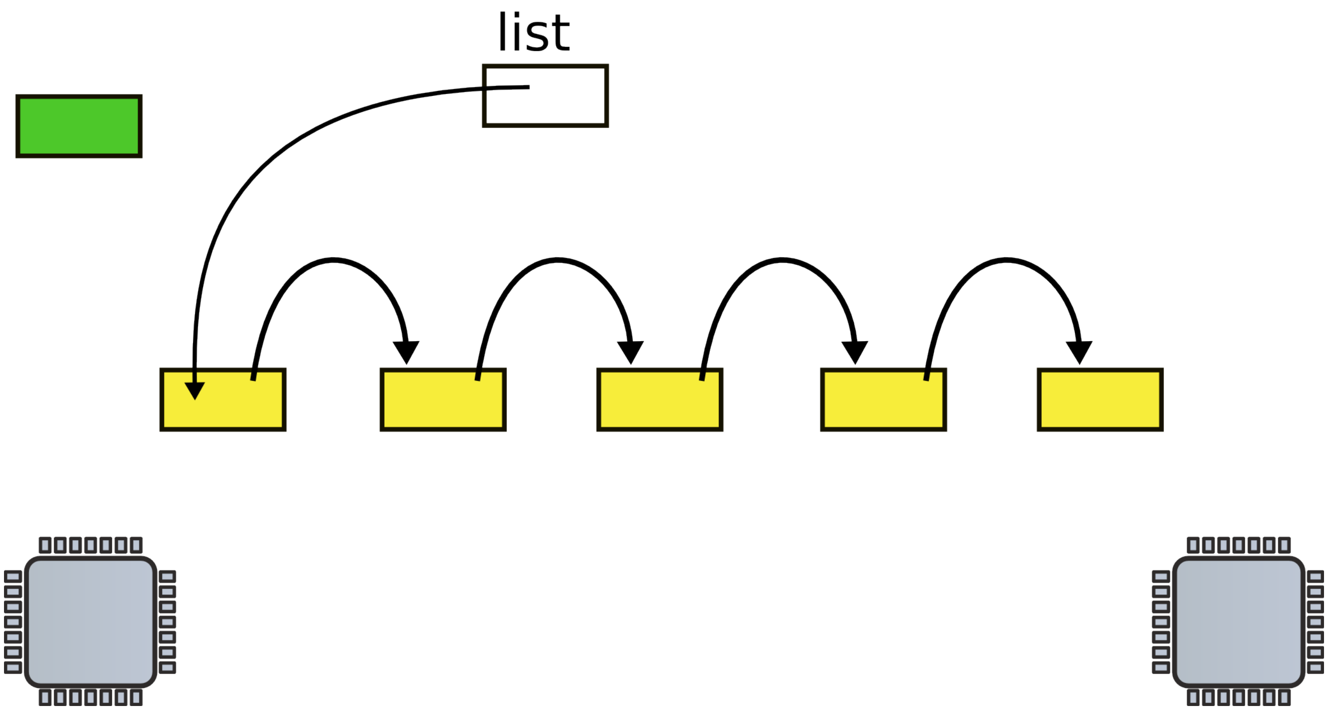
```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
   struct list *l;

   l = malloc(sizeof *l);
   l->data = data;
   l->next = list;
   list = l;
}
```

Request queue (e.g. incoming network packets)

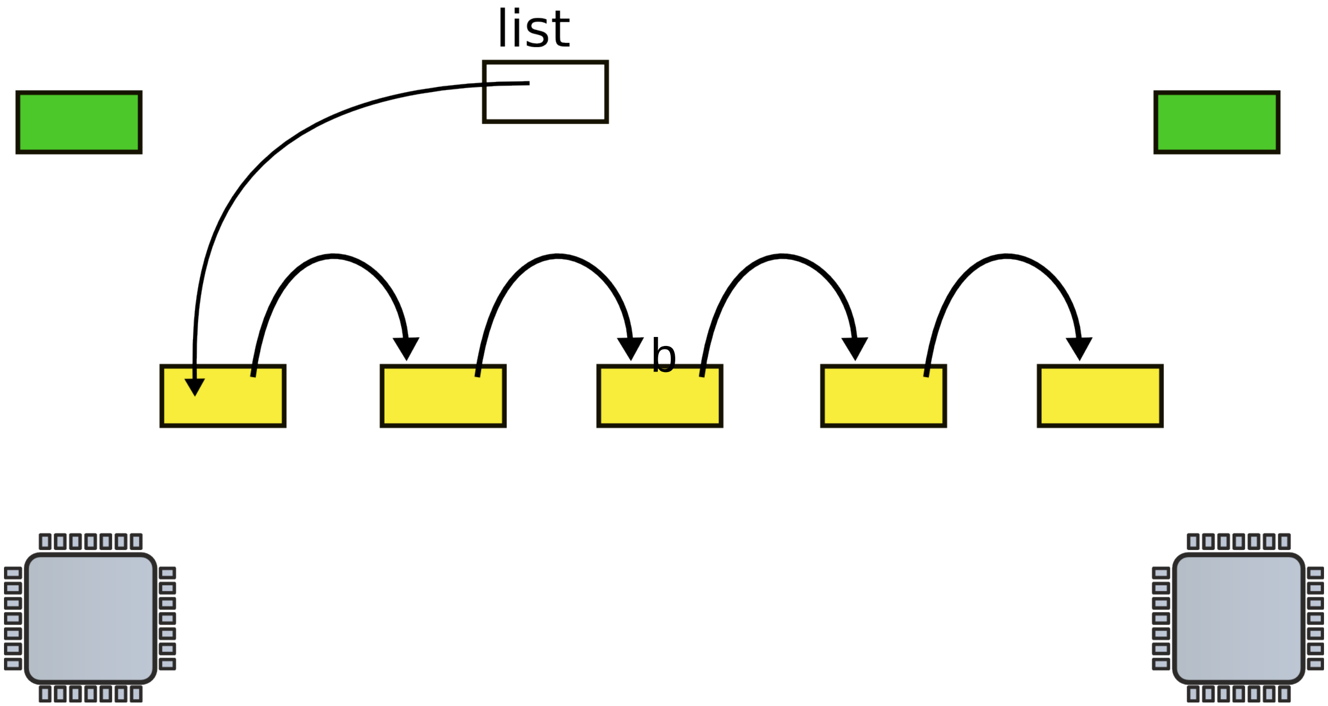


- Linked list, list is pointer to the first element

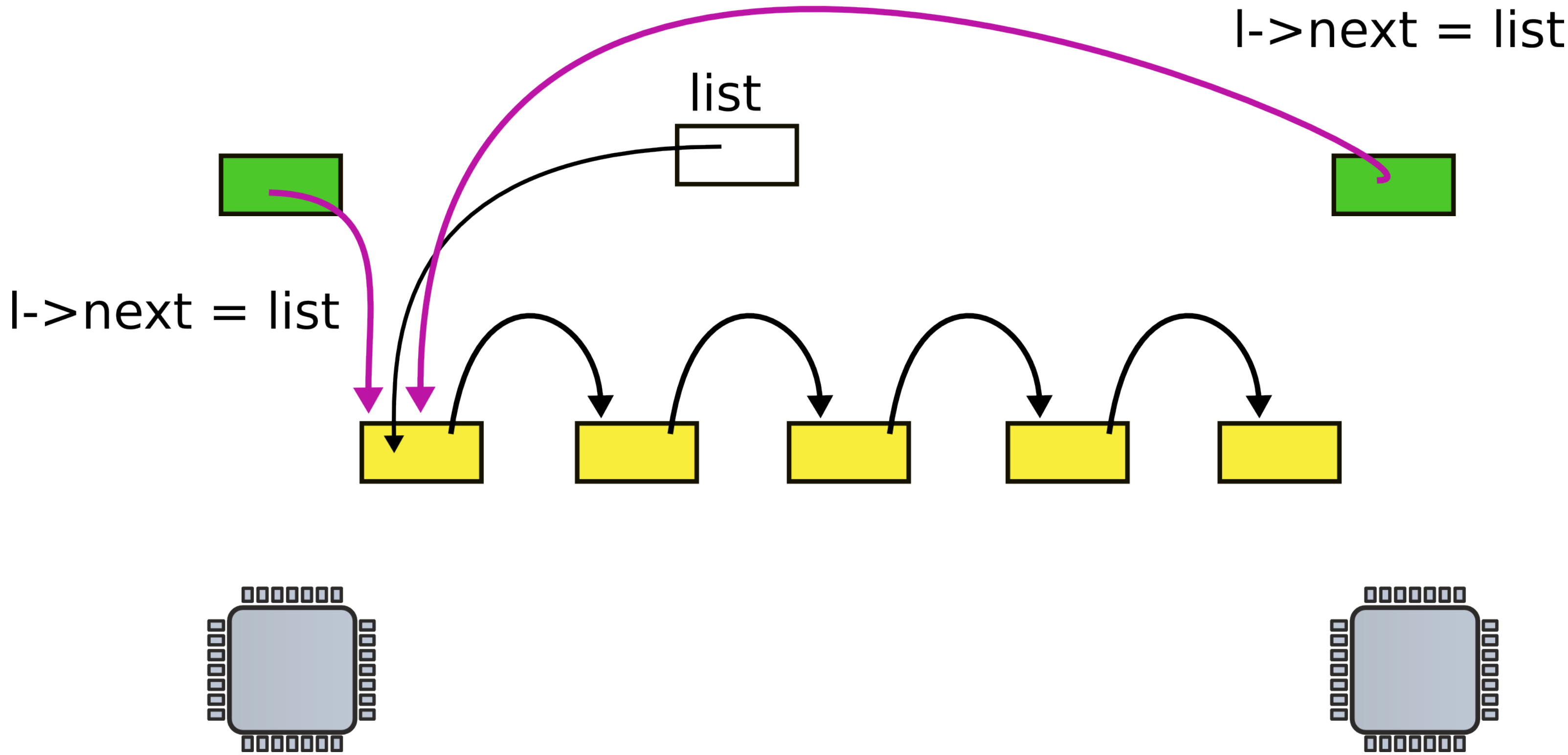
CPU1 allocates new request



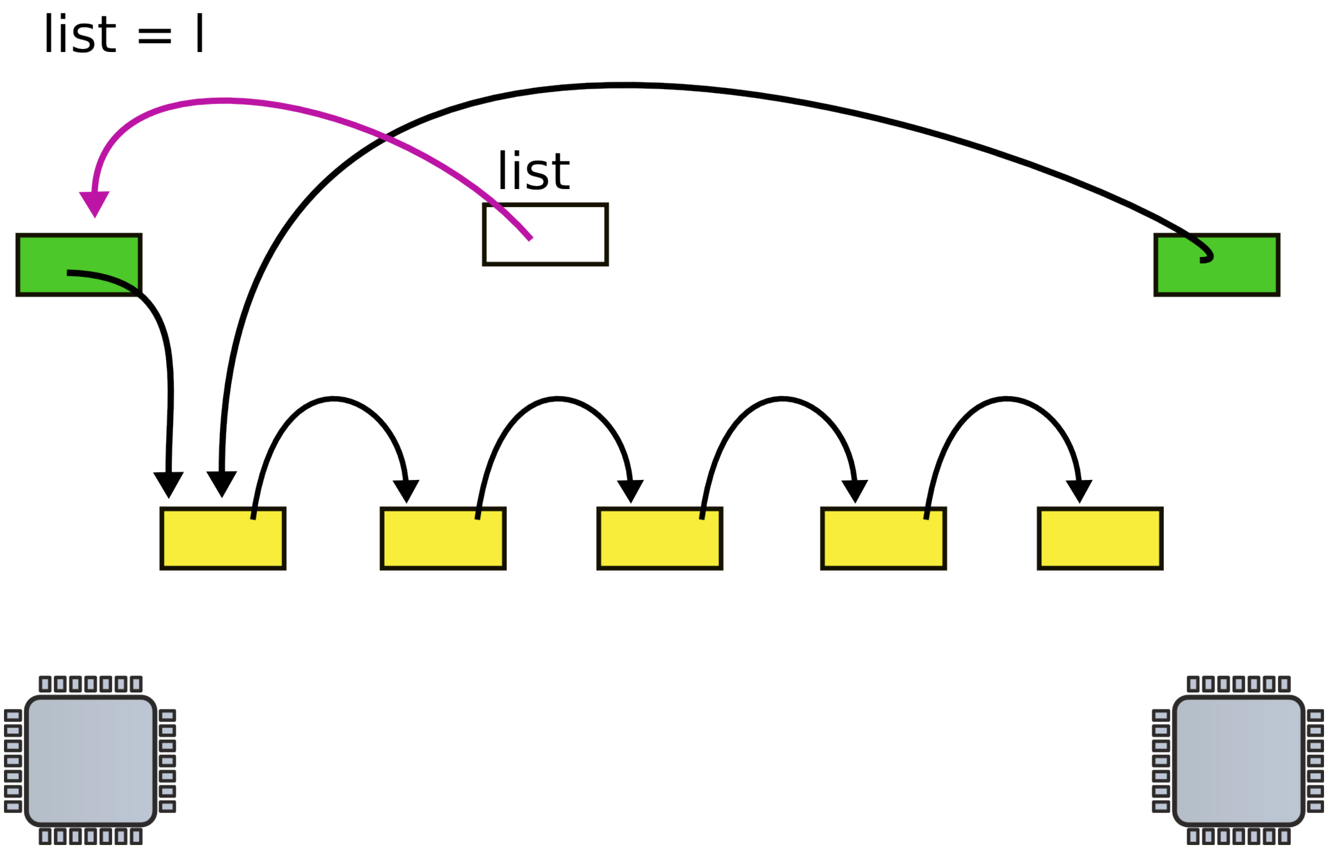
CPU2 allocates new request



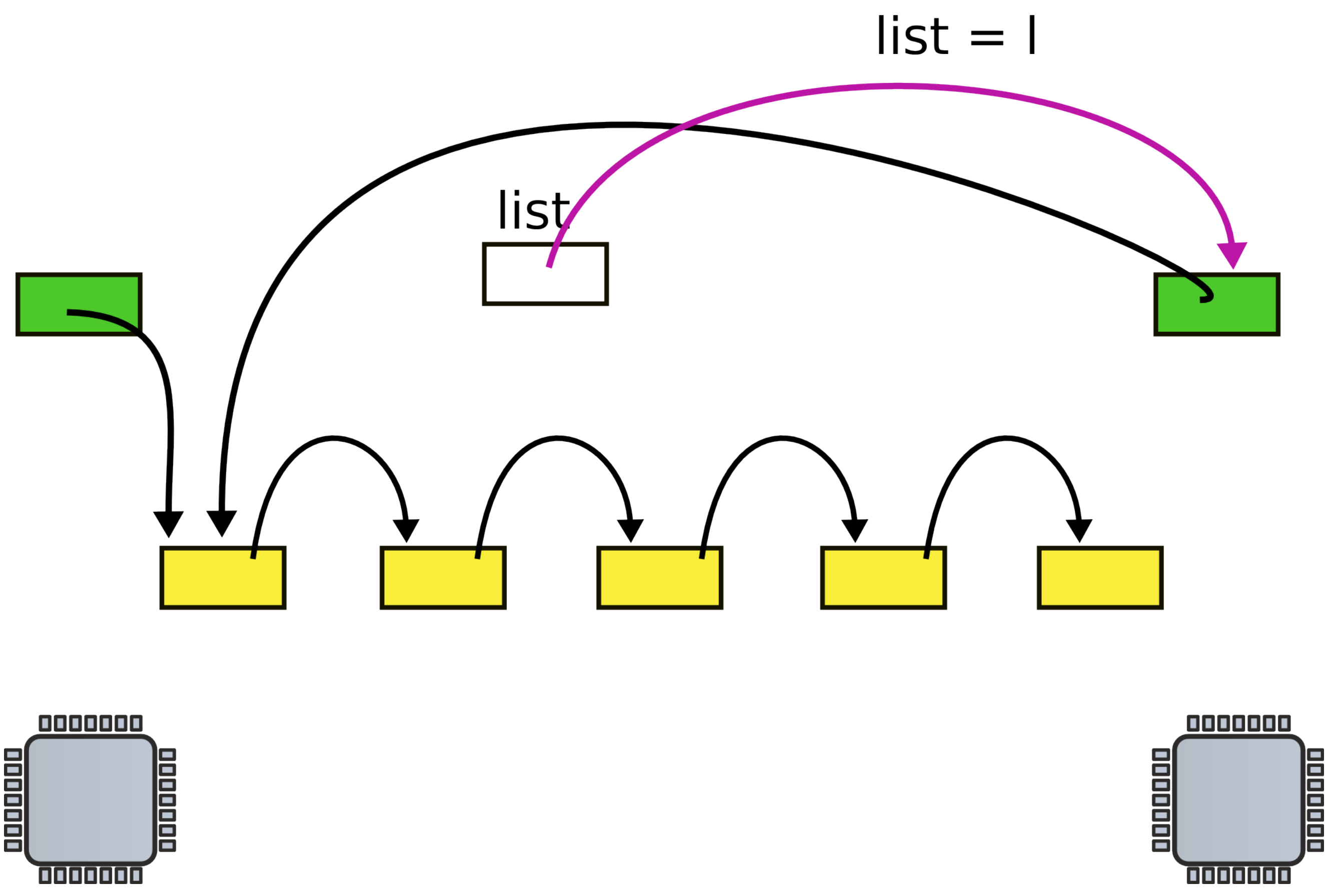
CPUs 1 and 2 update next pointer



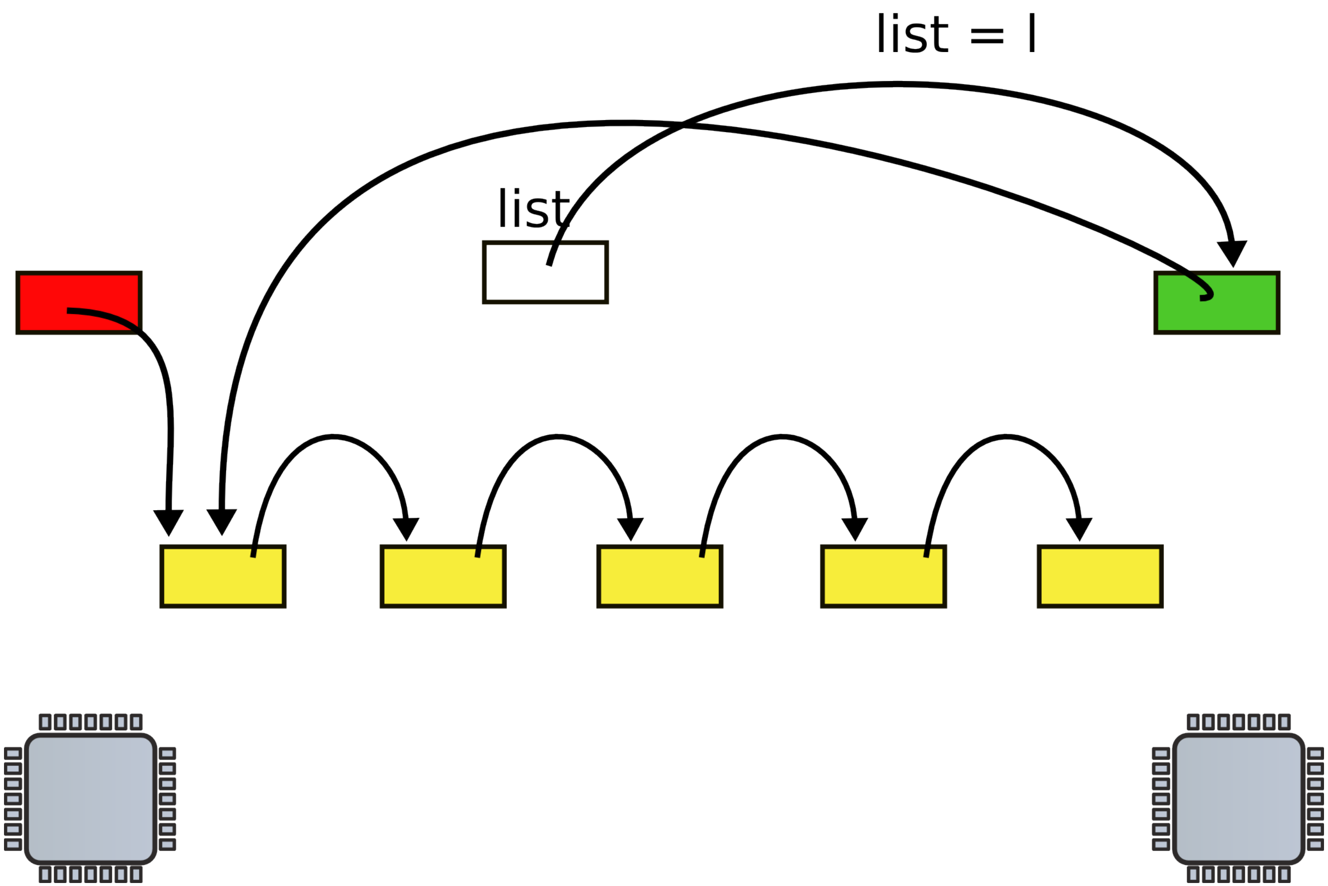
CPU1 updates head pointer



CPU2 updates head pointer



State after the race



Mutual exclusion

- Only one CPU can update list at a time

List implementation with locks

```c
struct list {
  int data;
  struct list *next;
};
struct list *list = 0;
struct lock listlock;
insert(int data)
{
   struct list *l;
   l = malloc(sizeof *l);
   acquire(&listlock);
   l->data = data;
   l->next = list;
   list = l;
   release(&listlock);
}
```

- Critical section

- How can we implement acquire()?

Spinlock

```
void
acquire(struct spinlock *lk)
{
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}
```

- 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

```
void
acquire(struct spinlock *lk)
{
...
  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;
...
}
```

xchgl instruction

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

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

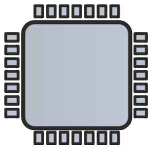
Deadlocks

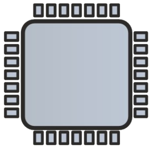
acquire(A)

acquire(B)

```
acquire(B) {
    while(xchg(&B−>locked, 1) != 0)
}
```

```
acquire(A) {
    while(xchg(&A−>locked, 1) != 0)
}
```





Lock ordering

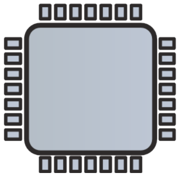
- Locks need to be acquired in the same order

Locks and interrupts

Network interrupt

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

}
```

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

}
```



Locks and interrupts

- Never hold a lock with interrupts enabled

```c
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();
...
}
```

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()

```
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();
}
```

Thank you!