# Locking

# Contents

- **Wrap up** (Cherry)
- **Problems**
  - Memory Allocator (moderate - Sunghyeok)
  - Buffer Cache (hard - Jina)

# Wrap up

1. Why Locks?
2. Spinlocks
3. Locking Strategy
4. Deadlocks and Lock Ordering
5. Re-entrant Locks
6. Interrupts and Memory Ordering
7. Sleeplocks

# (1/7) Why Locks ? : Races and Invariants

- A race condition happens when two CPUs access shared memory and at least one writes

```
1     struct element {
2       int data;
3       struct element *next;
4     };
5
6     struct element *list = 0;
7
8     void
9     push(int data)
10    {
11      struct element *l;
12
13      l = malloc(sizeof *l);
14      l->data = data;
15      l->next = list;
16      list = l;
```

- **Problem**

: list=l at the same time without lock -> cause overwrite

- **Solution (to preserve invariants)**

: acquire(&lock) ~ release(&lock) called critical section

: allow only one CPU access at this period

**But,**

it serialize the calls that can cause performance degradation (i.e. lock contention)

=> sol) maintain separate free list per CPU and allow stealing if needed

# (1/7) Why Locks ? : Races and Invariants

- A race condition happens when two CPUs access shared memory and at least one writes

```
6   struct element *list = 0;
7   struct lock listlock;
8
9   void
10  push(int data)
11  {
12    struct element *l;
13    l = malloc(sizeof *l);
14    l->data = data;
15
16    acquire(&listlock);
17    l->next = list;
18    list = l;
19    release(&listlock);
20  }
```

- **Problem**

: list=l at the same time without lock -> cause overwrite

- **Solution (to preserve invariants)**

: acquire(&lock) ~ release(&lock) called critical section

: allow only one CPU access at this period

**But,**

it serialize the calls that can cause performance degradation (i.e. lock contention)

=> sol) maintain separate free list per CPU and allow stealing if needed

# (2/7) Spinlocks : Implementation and Semantics

- Spinlocks rely on atomic test-and-set to ensure mutual exclusion
  - **Locked** : whether be used (0 : empty, 1 : owned)
  - **Cpu** : for debugging, current cpu id

```
void acquire(struct spinlock *lk) {
  while(1) {
    if (lk->locked == 0) {
      lk->locked = 1;
      break;
    }
  }
}
```

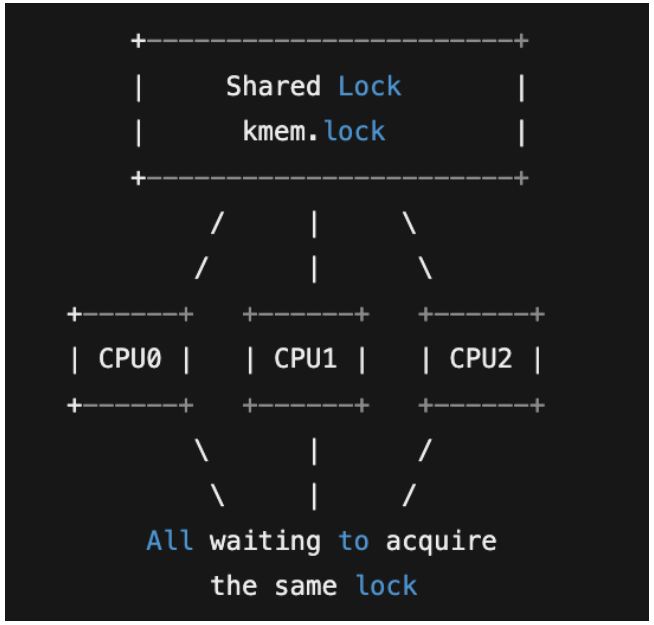
- **Problem**
: Multiple CPUs may see the lock as free at the same time and both attempt to acquire it

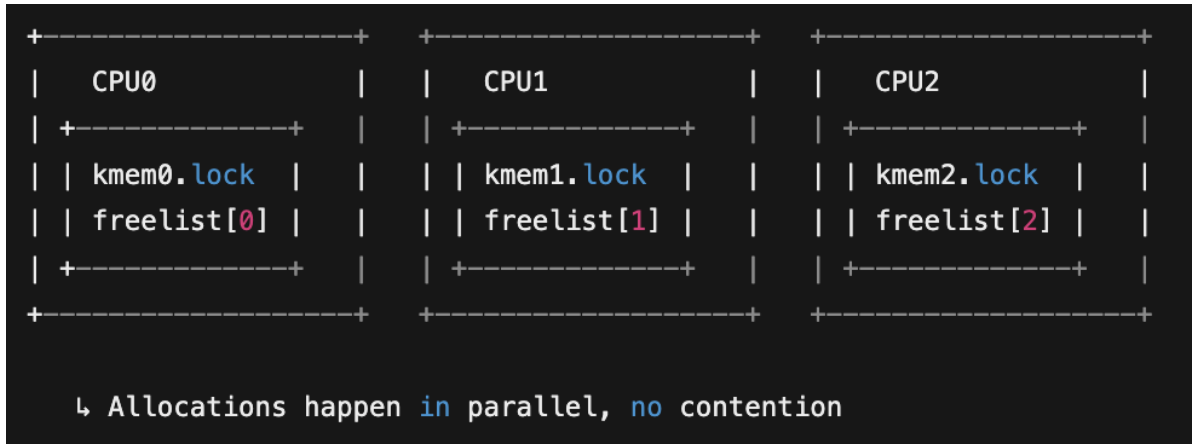
- **Solution (amoswap)**
: Initialize lk->locked and register value at the same time

```
while (__sync_lock_test_and_set(&lk->locked, 1) != 0)
  ; // spin until the lock is acquired
```

# (3/7) Locking Strategy : Granularity and Efficiency

- Coarse-grained locks are simpler to implement, but reduce CPU parallelism



# (4/7) Deadlocks and Lock Ordering

- **Deadlock** : Several threads locked at the same time, and waiting for each other forever
- Locks must be acquired in a globally consistent order

  - Example : **when creating file**
    1. Directory inode lock
    2. New file inode lock
    3. Disk block buffer lock
    4. Disk driver lock
    5. Calling process' p-> lock

1. Thread T1: `acquire(lock A)` → `acquire(lock B)`
2. Thread T2: `acquire(lock B)` → `acquire(lock A)`

# (5/7) Re-entrant Locks : Tempting but Risky

- **Re-entrant locks** seem convenient but break **atomicity of critical section**
  - **Recursive lock** : allow the same thread to acquire it multiple times without blocking
  - Xv6 does not support re-entrant locks ( rather cause panic )

# (5/7) Re-entrant Locks : Tempting but Risky

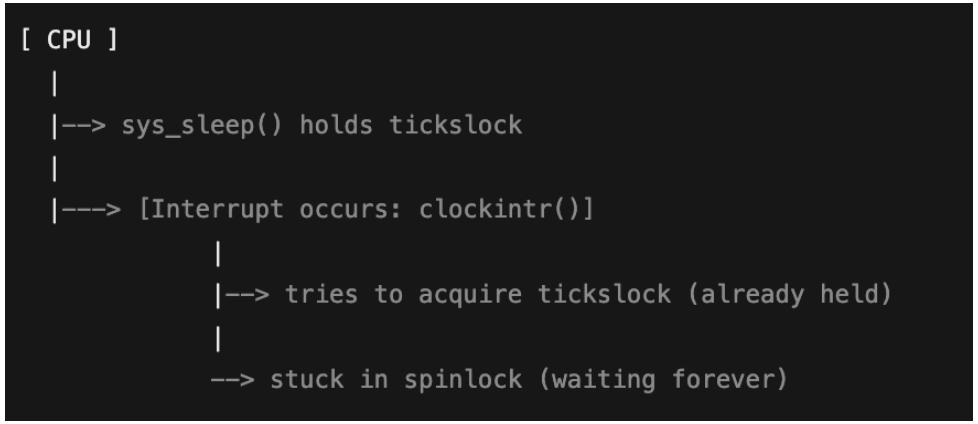
- Risky Scenario

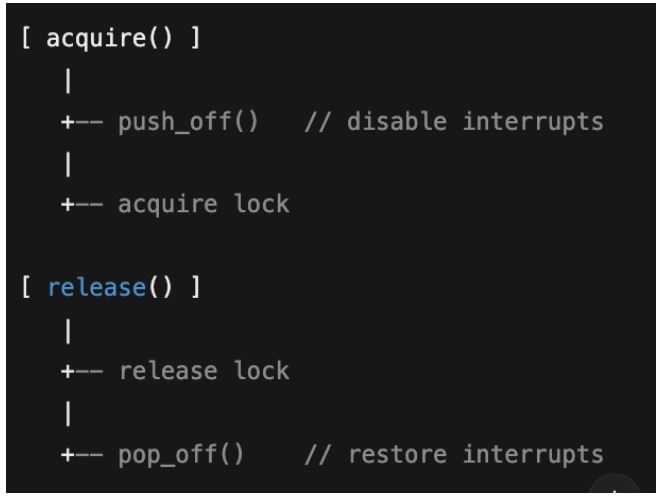
```
struct spinlock lock;
int data = 0; // protected by lock

f() {
  acquire(&lock);
  if(data == 0){
    call_once();
    h();
    data = 1;
  }
  release(&lock);
}

g() {
  aquire(&lock);
  if(data == 0){
    call_once();
    data = 1;
  }
  release(&lock);
}

h() {
    ...
}
```

- F() acquires the lock
- H() is called, which internally calls g()
- G() acquires the same lock again
- Then, call_once() may be called twice > logic error

# (6/7) Interrupts and Memory Ordering

- In xv6, push_off() is always called before acquiring any spinlock, enforcing this rule system-wide
- **Problem (Interrupts + locks = hidden deadlock)**

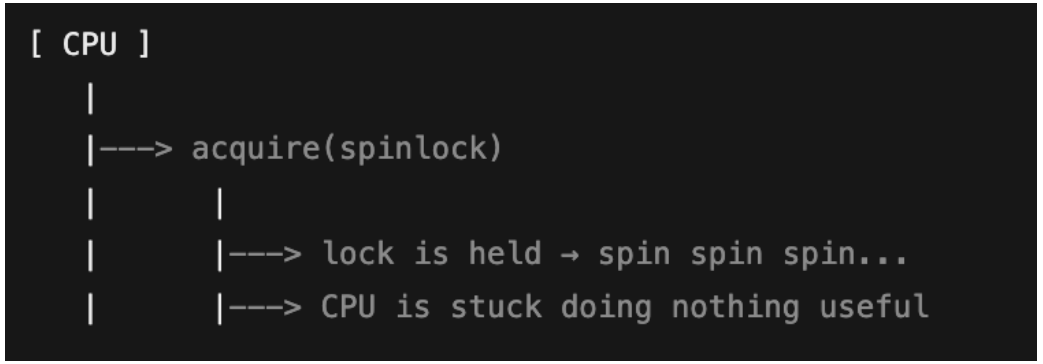


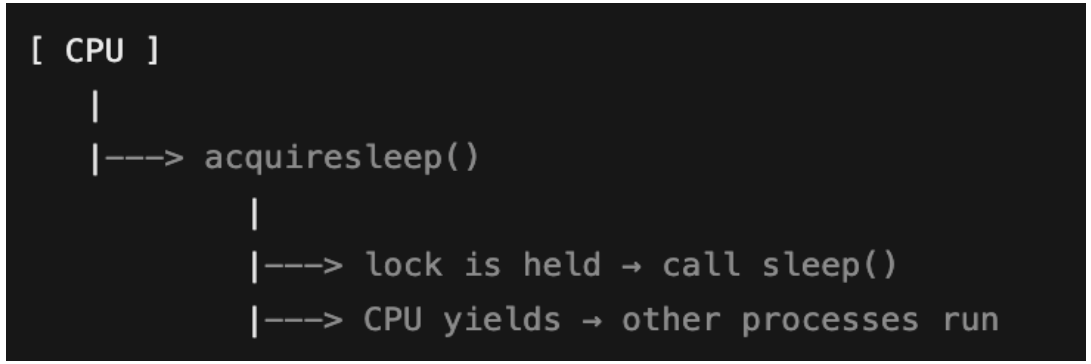




- **Solution (xv6)**
  - Always disable interrupts before acquiring a spinlock

# (7/7) Sleeplocks : For Long Critical Sections

- Use **sleeplocks** for long waits – they don't waste CPU
- Limitation of **spinlock -** they can't yield the cpu
- Limitation of **sleeplock**
  1. **Cannot** be used in interrupt handler
  2. **Cannot** use inside spinlock critical sections
     - It's generally safe to use spinlocks inside sleeplocks (but must avoid circular dependencies)



# Memory Allocator

moderate

# Problem Summary

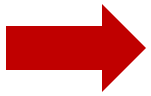
- Currently :
  - one shared free list in the kernel causing lock contention between `kalloc()` and `kfree()`
- Goal :
  - decrease lock contention by dividing the free list (shared resource) into separate, unshared free list

# Solution

- each CPU has its own free list
- each free list is protected by a spinlock
- “steal” from other CPU’s free list when there are no free memory page in its own free list to allocate

# data structure

```
rtm, 19 years ago | 1 author (rtm)
struct run {
  struct run *next;
};

rsc, 16 years ago | 1 author (rsc)
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;

void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange(pa_start: end, pa_end: (void*)PHYSTOP);
}
```



```
rtm, 19 years ago | 1 author (rtm)
struct run {
  struct run *next;
};

You, 3 days ago | 2 authors (rsc and one other)
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU];

char* kmemlock_names[NCPU] = { "kmem0", "kmem1", "kmem2", "kmem3",
  "kmem4", "kmem5", "kmem6", "kmem7" };

void
kinit()
{
  for (int i = 0; i < NCPU; ++i) {
    initlock(&kmem[i].lock, kmemlock_names[i]);
  }
  freerange(pa_start: end, pa_end: (void*)PHYSTOP);
}
```

# kfree()

```
// Free the page of physical memory pointed at by pa,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  push_off();
  int _cpuid = cpuid();
  pop_off();

  acquire(&kmem[_cpuid].lock);
  r->next = kmem[_cpuid].freelist;
  kmem[_cpuid].freelist = r;
  release(&kmem[_cpuid].lock);
}
```

# kalloc()

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
  struct run *r;

  push_off();
  int _cpuid = cpuid();
  pop_off();

  acquire(&kmem[_cpuid].lock);
  r = kmem[_cpuid].freelist;
  if (r) {
    kmem[_cpuid].freelist = r->next;
  }
  release(&kmem[_cpuid].lock);
```

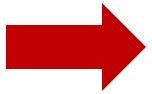
1. see if there is a free page in current CPU's free list

2. If no free page in current CPU's free list, steal from other CPU's free list

```
  if (!r) {
    for (int i = 0; i < NCPU; ++i) {
      if (_cpuid == i) {
        continue;
      }
      acquire(&kmem[i].lock);
      r = kmem[i].freelist;
      if (r) {
        kmem[i].freelist = r->next;
      }
      release(&kmem[i].lock);
      if (r) {
        break;
      }
    }
  }

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

# test result

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 137032 #acquire() 433068
lock: bcache: #test-and-set 0 #acquire() 1272
--- top 5 contended locks:
lock: kmem: #test-and-set 137032 #acquire() 433068
lock: virtio_disk: #test-and-set 57406 #acquire() 144
lock: pr: #test-and-set 39354 #acquire() 5
lock: uart: #test-and-set 8655 #acquire() 69
lock: proc: #test-and-set 8346 #acquire() 807
tot= 137032
test1 FAIL
start test2
total free number of pages: 32465 (out of 32768)
.....
test2 OK
start test3
..........child done 100000
--- lock kmem/bcache stats
lock: kmem: #test-and-set 339131 #acquire() 4233123
lock: bcache: #test-and-set 0 #acquire() 1372
--- top 5 contended locks:
lock: kmem: #test-and-set 339131 #acquire() 4233123
lock: uart: #test-and-set 229094 #acquire() 650
lock: virtio_disk: #test-and-set 57406 #acquire() 144
lock: pr: #test-and-set 39354 #acquire() 5
lock: proc: #test-and-set 38583 #acquire() 3317485
tot= 339131
test3 FAIL m 137032 n 339131
```



```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem0: #test-and-set 0 #acquire() 176509
lock: kmem1: #test-and-set 0 #acquire() 137134
lock: kmem2: #test-and-set 0 #acquire() 119468
lock: bcache: #test-and-set 0 #acquire() 1272
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 88282 #acquire() 144
lock: uart: #test-and-set 14327 #acquire() 67
lock: pr: #test-and-set 6009 #acquire() 5
lock: proc: #test-and-set 4340 #acquire() 1518
lock: proc: #test-and-set 3669 #acquire() 401559
tot= 0
test1 OK
start test2
total free number of pages: 32465 (out of 32768)
.....
test2 OK
start test3
..........child done 100000
--- lock kmem/bcache stats
lock: kmem0: #test-and-set 6627 #acquire() 2436766
lock: kmem1: #test-and-set 3321 #acquire() 1518320
lock: kmem2: #test-and-set 11604 #acquire() 1260738
lock: kmem3: #test-and-set 0 #acquire() 55
lock: kmem4: #test-and-set 0 #acquire() 55
lock: kmem5: #test-and-set 0 #acquire() 55
lock: kmem6: #test-and-set 0 #acquire() 55
lock: kmem7: #test-and-set 0 #acquire() 55
lock: bcache: #test-and-set 0 #acquire() 1372
--- top 5 contended locks:
lock: proc: #test-and-set 408118 #acquire() 3317569
lock: uart: #test-and-set 161536 #acquire() 780
lock: virtio_disk: #test-and-set 88282 #acquire() 144
lock: proc: #test-and-set 11620 #acquire() 716223
lock: kmem2: #test-and-set 11604 #acquire() 1260738
tot= 21552
test3 FAIL m 0 n 21552
```

# Buffer Cache

hard

# Problem Summary

- Bcachetest creates multiple processes that repeatedly read different files to generate contention on bcache.lock

- Currently :
  bcache.lock protects the list of cached block buffers, reference counts in each block buffer, and IDs of cached blocks

- Goal : modify the block cache so that the number of test-and-set iterations for all locks is close to zero

# Solution

- Use hash table instead of a single list for all blocks
- Each bucket has its own lock (concurrent access to different blocks)
- Remove LRU (use refcnt)
- Steal unused buffers from other buckets

# Hash Table Bucket

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // Sorted by how recently the buffer was used.
  // head.next is most recent, head.prev is least.
  struct buf head;
} bcache;
```

Prime number to reduce hash collision

```
#define NBUCKETS 13

struct {
  struct buf buf[NBUF];  // The buffer array

  // Hash table buckets
  struct {
    struct spinlock lock;
    struct buf head;
  } buckets[NBUCKETS];

  struct spinlock eviction_lock; // Lock for buffer eviction
} bcache;
```

```
static uint
hash(uint dev, uint blockno)
{
  return (dev + blockno) % NBUCKETS;
}
```

Hash function maps block numbers to specific buckets

# Locks for each bucket

```
void
binit(void)
{
  struct buf *b;

  initlock(&bcache.lock, "bcache");

  // Create linked list of buffers
  bcache.head.prev = &bcache.head;
  bcache.head.next = &bcache.head;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    initsleeplock(&b->lock, "buffer");
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}
```

```
void
binit(void)
{
  initlock(&bcache.eviction_lock, "bcache");

  for(int i = 0; i < NBUCKETS; i++){
    initlock(&bcache.buckets[i].lock, "bcache.bucket");

    bcache.buckets[i].head.prev = &bcache.buckets[i].head;
    bcache.buckets[i].head.next = &bcache.buckets[i].head;
  }

  // Initialize all buffers
  for(struct buf *b = bcache.buf; b < bcache.buf + NBUF; b++){
    b->next = b->prev = b;  // Not linked to any list yet
    initsleeplock(&b->lock, "buffer");
    b->refcnt = 0;
    b->valid = 0;

    b->next = bcache.buckets[0].head.next;
    b->prev = &bcache.buckets[0].head;
    bcache.buckets[0].head.next->prev = b;
    bcache.buckets[0].head.next = b;
  }
}
```

- Separate eviction lock for buffer allocation operations
- Each bucket gets its own lock

# Remove LRU – bget function

Find and return a buffer for a specific block

- If requested block is cached and found, increment refcnt and return buffer
- If not found, allocate an unused buffer

```
// Not cached.
// Recycle the least recently used (LRU) unused buffer.
for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
  if(b->refcnt == 0) {
    b->dev = dev;
    b->blockno = blockno;
    b->valid = 0;
    b->refcnt = 1;
    release(&bcache.lock);
    acquiresleep(&b->lock);
    return b;
  }
}
```

```
for(b = bucket_head->next; b != bucket_head; b = b->next){
  if(b->refcnt == 0) {
    b->dev = dev;
    b->blockno = blockno;
    b->valid = 0;
    b->refcnt = 1;
    release(&bcache.buckets[bucket_id].lock);
    acquiresleep(&b->lock);
    return b;
  }
}
```

- Look for unused buffer in this bucket
- Instead of searching through a global LRU list, find any buffer with refcnt == 0

# Remove LRU – brelse function

```
// Release a locked buffer.
// Move to the head of the most-recently-used list.
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  acquire(&bcache.lock);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }

  release(&bcache.lock);
}
```

```
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  uint bucket_id = hash(b->dev, b->blockno);

  acquire(&bcache.buckets[bucket_id].lock);
  b->refcnt--;
  release(&bcache.buckets[bucket_id].lock);
}
```

- Just decrement the ref count
- Only requires the lock for this specific bucket containing the buffer

```
panic("bget: no buffers");
```

# Buffer Stealing – bget function

```
acquire(&bcache.eviction_lock);
```

```
struct buf *victim = 0;

for(int i = 0; i < NBUCKETS; i++) {
  if(i == bucket_id) continue; // Skip current bucket

  struct buf *other_head = &bcache.buckets[i].head;
  acquire(&bcache.buckets[i].lock);

  for(b = other_head->next; b != other_head; b = b->next) {
    if(b->refcnt == 0) { // Found not-in-use buffer to steal
      victim = b;

      // Remove from current bucket's list
      b->next->prev = b->prev;
      b->prev->next = b->next;

      release(&bcache.buckets[i].lock);
      goto found_victim;
    }
  }

  release(&bcache.buckets[i].lock);
}
```

- Find an unused buffer in other buckets
  if no buffer is available in this bucket
- Acquires eviction_lock

- When victim is found,
  it is removed from its bucket and added to the target bucket

```
found_victim:
  // Use the victim buffer
  acquire(&bcache.buckets[bucket_id].lock);
  victim->dev = dev;
  victim->blockno = blockno;
  victim->valid = 0;
  victim->refcnt = 1;

  // Add to our bucket's list
  victim->next = bucket_head->next;
  victim->prev = bucket_head;
  bucket_head->next->prev = victim;
  bucket_head->next = victim;

  release(&bcache.buckets[bucket_id].lock);
  release(&bcache.eviction_lock);

  acquiresleep(&victim->lock);
  return victim;
```

# Buffer Stealing

```
// need to steal a buffer from another bucket
acquire(&bcache.eviction_lock);

acquire(&bcache.buckets[bucket_id].lock);
// check our bucket for empty buffer
release(&bcache.buckets[bucket_id].lock);

// find empty buffer from another bucket

acquire(&bcache.buckets[bucket_id].lock);
// add victim to our bucket
release(&bcache.buckets[bucket_id].lock);

release(&bcache.eviction_lock);
```

# The hardest part

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 33030
lock: kmem: #test-and-set 0 #acquire() 28
lock: kmem: #test-and-set 0 #acquire() 73
lock: bcache: #test-and-set 0 #acquire() 96
lock: bcache.bucket: #test-and-set 0 #acquire() 6229
lock: bcache.bucket: #test-and-set 0 #acquire() 6204
lock: bcache.bucket: #test-and-set 0 #acquire() 4298
lock: bcache.bucket: #test-and-set 0 #acquire() 4286
lock: bcache.bucket: #test-and-set 0 #acquire() 2302
lock: bcache.bucket: #test-and-set 0 #acquire() 4272
lock: bcache.bucket: #test-and-set 0 #acquire() 2695
lock: bcache.bucket: #test-and-set 0 #acquire() 4709
lock: bcache.bucket: #test-and-set 0 #acquire() 6512
lock: bcache.bucket: #test-and-set 0 #acquire() 6197
lock: bcache.bucket: #test-and-set 0 #acquire() 6196
lock: bcache.bucket: #test-and-set 0 #acquire() 6201
lock: bcache.bucket: #test-and-set 0 #acquire() 6201
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 1483888 #acquire() 1221
lock: proc: #test-and-set 38718 #acquire() 76050
lock: proc: #test-and-set 34460 #acquire() 76039
lock: proc: #test-and-set 31663 #acquire() 75963
lock: wait_lock: #test-and-set 11794 #acquire() 16
tot= 0
test0: OK
```

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: bcache: #test-and-set 0 #acquire() 20
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 1862755 #acquire() 1284
lock: proc: #test-and-set 97977 #acquire() 67086
lock: proc: #test-and-set 38479 #acquire() 67089
lock: proc: #test-and-set 27676 #acquire() 67076
lock: proc: #test-and-set 24742 #acquire() 67089
tot= 0
test0: OK
```

# The hardest part

```
void
kinit()
{
  char lockname[16];
  for(int i = 0; i < NCPU; i++) { // per-CPU locks
    snprintf(lockname, sizeof(lockname), "kmem%d", i);
    initlock(&kmem[i].lock, lockname);
  }
  freerange(end, (void*)PHYSTOP);
}
```

```
void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->cpu = 0;
#ifdef LAB_LOCK
  lk->nts = 0;
  lk->n = 0;
  findslot(lk);
#endif
}
```

# The hardest part

```
void
kinit()
{
  // char lockname[16];
  for(int i = 0; i < NCPU; i++) { // per-CPU locks
    // snprintf(lockname, sizeof(lockname), "kmem%d", i);
    initlock(&kmem[i].lock, "kmem");
  }
  freerange(end, (void*)PHYSTOP);
}
```

```
void
binit(void)
{
  // char lockname[16];

  initlock(&bcache.eviction_lock, "bcache");

  for(int i = 0; i < NBUCKETS; i++){
    // snprintf(lockname, sizeof(lockname), "bcache.bucket%d", i);
    initlock(&bcache.buckets[i].lock, "bcache.bucket");

    bcache.buckets[i].head.prev = &bcache.buckets[i].head;
    bcache.buckets[i].head.next = &bcache.buckets[i].head;
  }
```

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 32930
lock: kmem: #test-and-set 0 #acquire() 85
lock: kmem: #test-and-set 0 #acquire() 86
lock: bcache: #test-and-set 0 #acquire() 20
lock: bcache.bucket: #test-and-set 0 #acquire() 2130
lock: bcache.bucket: #test-and-set 0 #acquire() 4120
lock: bcache.bucket: #test-and-set 0 #acquire() 2122
lock: bcache.bucket: #test-and-set 0 #acquire() 4278
lock: bcache.bucket: #test-and-set 0 #acquire() 4324
lock: bcache.bucket: #test-and-set 0 #acquire() 6334
lock: bcache.bucket: #test-and-set 0 #acquire() 6334
lock: bcache.bucket: #test-and-set 0 #acquire() 6752
lock: bcache.bucket: #test-and-set 0 #acquire() 9014
lock: bcache.bucket: #test-and-set 0 #acquire() 6176
lock: bcache.bucket: #test-and-set 0 #acquire() 6180
lock: bcache.bucket: #test-and-set 0 #acquire() 4120
lock: bcache.bucket: #test-and-set 0 #acquire() 4130
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 1994932 #acquire() 1284
lock: proc: #test-and-set 104663 #acquire() 66985
lock: proc: #test-and-set 24448 #acquire() 66994
lock: pr: #test-and-set 9690 #acquire() 5
lock: proc: #test-and-set 5249 #acquire() 66463
tot= 0
test0: OK
```