Handwritten title: **MemAllocater**

```cpp
1   // from the header file
2   class Singleton {
3   public:
4   static Singleton* instance();
5     ...
6   private:
7     static Singleton* pInstance;
8   };
9
10  // from the implementation file
11  Singleton* Singleton::pInstance = 0;
12
13  Singleton* Singleton::instance() {
14    if (pInstance == 0) {
15      pInstance = new Singleton;
16    }
17    return pInstance;
18  }
```
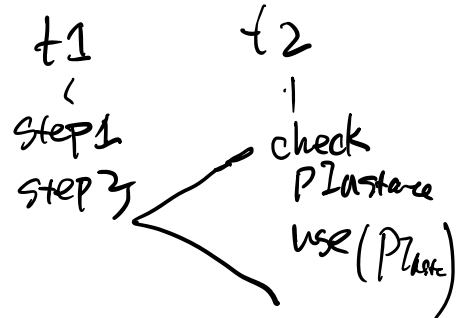
Handwritten annotations:

new MemAlloc X
← MemAlloc.instance()
obj

acquire mutex → (line 14)
release → (line 16/17)

first time (near line 13/14)

```cpp
Singleton* Singleton::instance() {
  if (pInstance == 0) {           // 1st test
    Lock lock;
    if (pInstance == 0) {         // 2nd test
      pInstance = new Singleton;
    }
  }
  return pInstance;
}
```

C++

t1   t2
step1   check pInstance
step3   use(PInst)

```cpp
→ pInstance =                                // Step 3
    operator new(sizeof(Singleton)); // Step 1
  new (pInstance) Singleton;        // Step 2
```

← assign (Step 3)
← alloc mem (Step 1)
← init (Step 2)

Example borrowed from "C++ and the Perils of Double-Checked Locking"
https://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

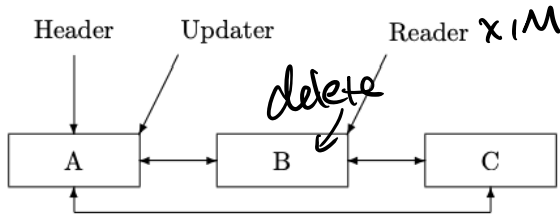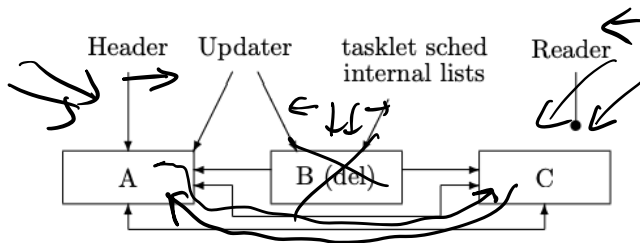Header   Updater   Reader x1M

delete

A ↔ B ↔ C

Figure 11: List Initial State

Header   Updater   Reader

A ↔ B (del) ↔ C

Figure 12: Element B Unlinked From List

Header   Updater   tasklet sched   Reader
                    internal lists

A ↔ B (del) ↔ C

Figure 13: List After Grace Period

Header   Updater   Reader

A ↔ C
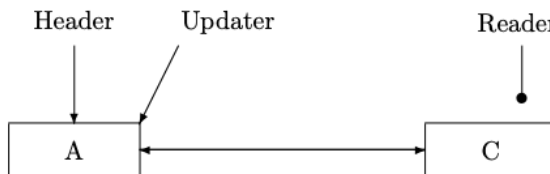
Figure 14: List After Element B Returned to Freel-ist

Borrowed from "Read-Copy Update"
https://cs.nyu.edu/~mwalfish/classes/14fa/ref/rcu2.pdf

1. Concurrency, continued
2. Deadlock ←
3. Other concurrency issues ↙
4. Other synchronization mechanisms
----

midterm.   review Session
      52   30 min + 1hr Q&A

(echo abc ; echo def) | wc -l
                              ↘ 2   dup2 (10, 1)
              ↳ abc
                def
                                    1   10

- Condition Variable   ⊠ ↘ Screen kernel
                                          ↘ file .txt
      cond_wait (cond, mutex)
        ① release , ② wait

      C   ③ acquire , ④ resume
   signal
   /broadcast ⟶ while (! cond) {   t1 t2
                    ↓ cond-wait ↙↙ t3
                 }   ↓↓

   (signal)
      /broadcast ?   IF for us,
                             YES.   MemAllocator
                                              0
   (broadcast)   - alloc
       /signal ?   Not always.   - free
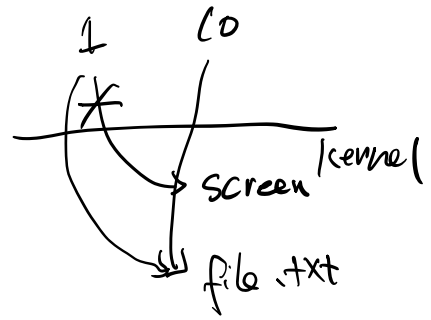   ↗ t1 . alloc (100)
                              signal   t2   alloc (50)
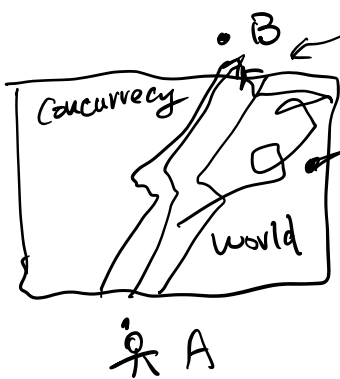   ↳ t3   free (75)

Monitor (= C.V. + mutex)

↑ 6 rules
↑ 4-step design

Safe way

Concurrency
world
• B
A

→ double-checked
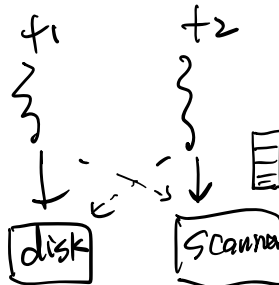locking Pattern

lock/C.V.
threads.

state ← monitor

Deadlock.

resource
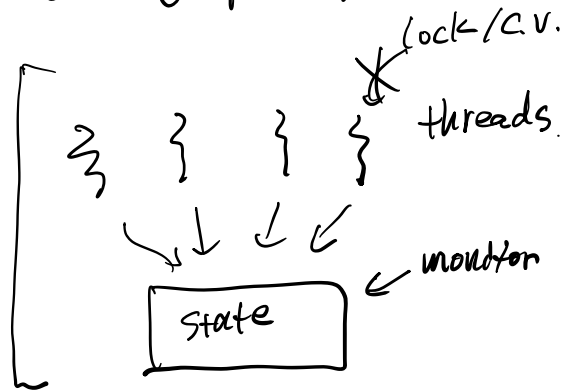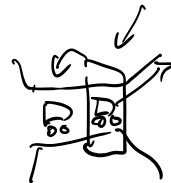
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait.

t1   t2

disk   Scanner

Solutions:

① ignore it

② detect & recover

③ avoid-dealock algo.

banker's algo $\{$ ① max res
② if <u>safe state</u>, then Okay
↳ exist schedule
to fishish

bank: 10
A: 9
B: 4
C: 7

	has	max
A	3+1	9
B	2	4
C	2	7

bank: 3÷1=2

Q: Safe state?   Yes.   (bank: 5)   (bank=7)
$B^{(+2)} \to C^{(+5)} \to A$

Q: A asks for 1 coin? NO.

④ negate one of 4 conditions
negate 4
lock A ⟶ lock B
+1
+2
+3.
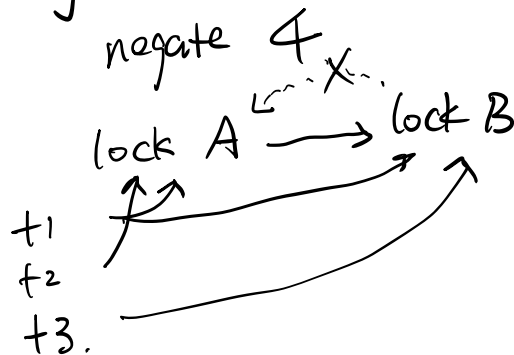
① C.V.
② what if
sb/some thread
fail follow order
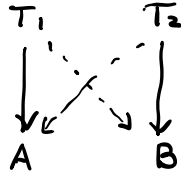
⑤ static & dynamic detector

```
1    CS5600, Handout week 7.a
2
3    ① Simple deadlock example
4
5      T1:
6        acquire(mutexA);
7        acquire(mutexB);
8
9        // do some stuff
10
11       release(mutexB);
12       release(mutexA);
13
14     T2:
15       acquire(mutexB);
16       acquire(mutexA);
17
18       // do some stuff
19
20       release(mutexA);
21       release(mutexB);
22
23
24
25   2. More subtle deadlock example
26
27     Let M be a monitor (shared object with methods protected by mutex)
28     Let N be another monitor
29
30     class M {
31       private:
32         Mutex mutex_m;
33
34         // instance of monitor N
35         N another_monitor;
36
37         // Assumption: no other objects in the system hold a pointer
38         // to our "another_monitor"
39
40       public:
41         M();
42         ~M();
43         void methodA();
44         void methodB();
45     };
```
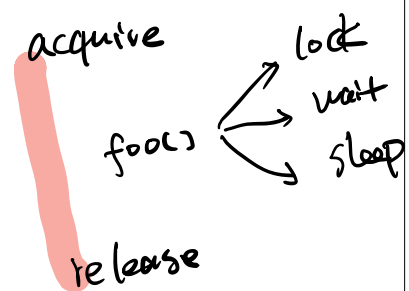
(handwritten) T1   T2 ... A   B

(handwritten) M, N : monitors

(handwritten) acquire ... foo() → lock, wait, sleep ... release

(handwritten) MemAllocator
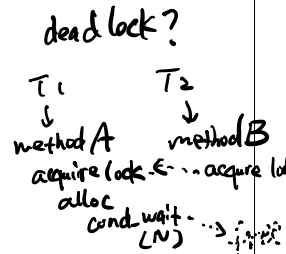
```
46     class N {
47       private:
48         Mutex mutex_n;
49         Cond cond_n;
50         int navailable;
51
52       public:
53         N();
54         ~N();
55         void* alloc(int nwanted);
56         void free(void*);
57     }
58
59     int
60     N::alloc(int nwanted) {
61       acquire(&mutex_n);
62       while (navailable < nwanted) {
63         wait(&cond_n, &mutex_n);
64       }
65
66       // peel off the memory
67
68       navailable -= nwanted;
69       release(&mutex_n);
70     }
71
72     void
73     N::free(void* returning_mem) {
74
75       acquire(&mutex_n);
76
77       // put the memory back
78
79       navailable += returning_mem;
80
81       broadcast(&cond_n, &mutex_n);
82
83       release(&mutex_n);
84     }
85
86     void
87     M::methodA() {
88
89       acquire(&mutex_m);
90
91       void* new_mem = another_monitor.alloc(int nbytes);
92
93       // do a bunch of stuff using this nice
94       // chunk of memory n allocated for us
95
96       release(&mutex_m);
97     }
98
99     void
100    M::methodB() {
101
102      acquire(&mutex_m);
103
104      // do a bunch of stuff
105
106      another_monitor.free(some_pointer);
107
108      release(&mutex_m);
109    }
110
111  QUESTION: What's the problem?
```

(handwritten) signal X

(handwritten) deadlock? T1 → methodA   T2 → methodB   acquire lock ← ... → acquire lock, alloc, cond_wait (N)

(handwritten) Mem: 0

```
112
113    3. Locking brings a performance vs. complexity trade-off
114
115    /*
116     * linux/mm/filemap.c
117     *
118     * Copyright (C) 1994-1999 Linus Torvalds
119     */
120
121    /*
122     * This file handles the generic file mmap semantics used by
123     * most "normal" filesystems (but you don't /have/ to use this:
124     * the NFS filesystem used to do this differently, for example)
125     */
126    #include <linux/export.h>
127    #include <linux/compiler.h>
128    #include <linux/dax.h>
129    #include <linux/fs.h>
130    #include <linux/sched/signal.h>
131    #include <linux/uaccess.h>
132    #include <linux/capability.h>
133    #include <linux/kernel_stat.h>
134    #include <linux/gfp.h>
135    #include <linux/mm.h>
136    #include <linux/swap.h>
137    #include <linux/mman.h>
138    #include <linux/pagemap.h>
139    #include <linux/file.h>
140    #include <linux/uio.h>
141    #include <linux/hash.h>
142    #include <linux/writeback.h>
143    #include <linux/backing-dev.h>
144    #include <linux/pagevec.h>
145    #include <linux/blkdev.h>
146    #include <linux/security.h>
147    #include <linux/cpuset.h>
148    #include <linux/hugetlb.h>
149    #include <linux/memcontrol.h>
150    #include <linux/cleancache.h>
151    #include <linux/shmem_fs.h>
152    #include <linux/rmap.h>
153    #include "internal.h"
154
155    #define CREATE_TRACE_POINTS
156    #include <trace/events/filemap.h>
157
158    /*
159     * FIXME: remove all knowledge of the buffer layer from the core VM
160     */
161    #include <linux/buffer_head.h> /* for try_to_free_buffers */
162
163    #include <asm/mman.h>
164
165    /*
166     * Shared mappings implemented 30.11.1994. It's not fully working yet,
167     * though.
168     *
169     * Shared mappings now work. 15.8.1995 Bruno.
170     *
171     * finished 'unifying' the page and buffer cache and SMP-threaded the
172     * page-cache, 21.05.1999, Ingo Molnar <mingo@redhat.com>
173     *
174     * SMP-threaded pagemap-LRU 1999, Andrea Arcangeli <andrea@suse.de>
175     */
176
177    /*
178     * Lock ordering:
179     *
180     *  ->i_mmap_rwsem            (truncate_pagecache)
181     *    ->private_lock          (__free_pte->__set_page_dirty_buffers)
182     *      ->swap_lock           (exclusive_swap_page, others)
183     *        ->i_pages lock
184     *
185     *  ->i_mutex
```
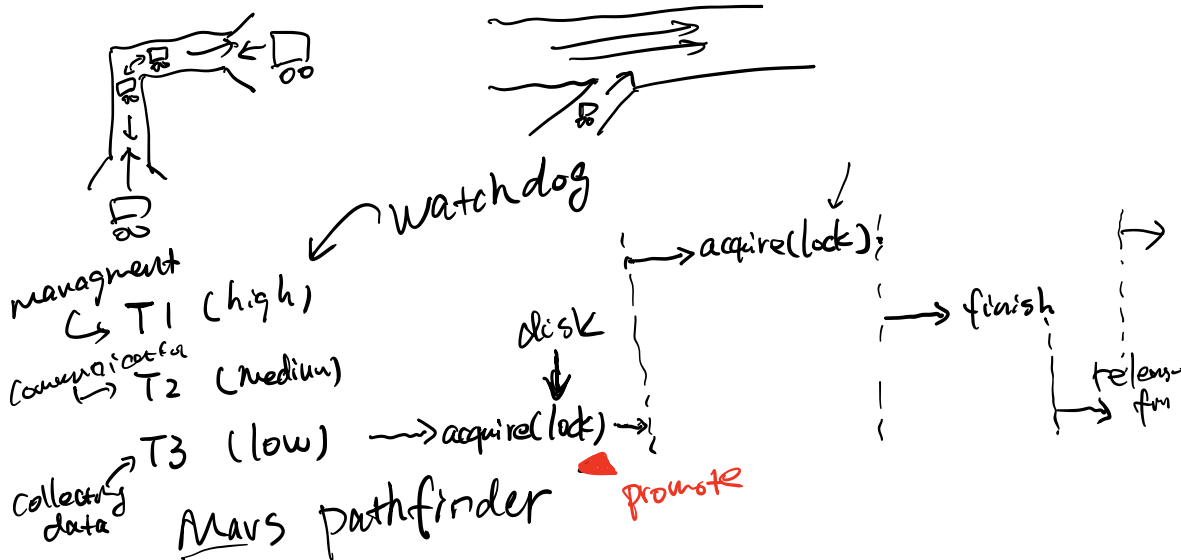
```
186    *    ->i_mmap_rwsem          (truncate->unmap_mapping_range)
187    *
188    *  ->mmap_sem
189    *    ->i_mmap_rwsem
190    *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
191    *        ->i_pages lock      (arch-dependent flush_dcache_mmap_lock)
192    *
193    *  ->mmap_sem
194    *    ->lock_page             (access_process_vm)
195    *
196    *  ->i_mutex                 (generic_perform_write)
197    *    ->mmap_sem              (fault_in_pages_readable->do_page_fault)
198    *
199    * bdi->wb.list_lock
200    *    sb_lock                 (fs/fs-writeback.c)
201    *    ->i_pages lock          (__sync_single_inode)
202    *
203    *  ->i_mmap_rwsem
204    *    ->anon_vma.lock         (vma_adjust)
205    *
206    *  ->anon_vma.lock
207    *    ->page_table_lock or pte_lock   (anon_vma_prepare and various)
208    *
209    *  ->page_table_lock or pte_lock
210    *    ->swap_lock             (try_to_unmap_one)
211    *    ->private_lock          (try_to_unmap_one)
212    *    ->i_pages lock          (try_to_unmap_one)
213    *    ->zone_lru_lock(zone)   (follow_page->mark_page_accessed)
214    *    ->zone_lru_lock(zone)   (check_pte_range->isolate_lru_page)
215    *    ->private_lock          (page_remove_rmap->set_page_dirty)
216    *    ->i_pages lock          (page_remove_rmap->set_page_dirty)
217    *    bdi.wb->list_lock       (page_remove_rmap->set_page_dirty)
218    *    ->inode->i_lock         (page_remove_rmap->set_page_dirty)
219    *    ->memcg->move_lock      (page_remove_rmap->lock_page_memcg)
220    *    bdi.wb->list_lock       (zap_pte_range->set_page_dirty)
221    *    ->inode->i_lock         (zap_pte_range->set_page_dirty)
222    *    ->private_lock          (zap_pte_range->__set_page_dirty_buffers)
223    *
224    *  ->i_mmap_rwsem
225    *    ->tasklist_lock         (memory_failure, collect_procs_ao)
226    */
227
228    static int page_cache_tree_insert(struct address_space *mapping,
229    struct page *page, void **shadowp)
230    {
231         struct radix_tree_node *node;
232         .....
233
234    [the point is: fine-grained locking leads to complexity.]
235
```
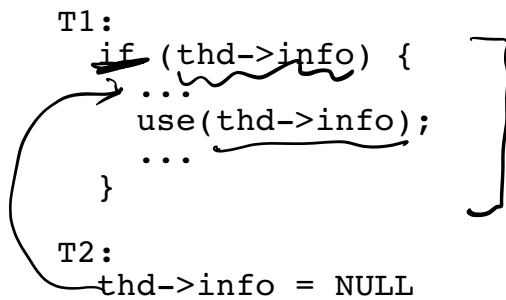
Other concurrency issues

(a) progress issues (deadlock)

- Livelock

- Starvation

- Priority inversion

watchdog

management
↳ T1 (high)

communication
↳ T2 (medium)

disk

T3 (low) ──→ acquire(lock) ──→

collecting
data

Mars pathfinder   promote

acquire(lock);

finish

release
fm

(b) other concurrency bugs

  - atomicity-violation bugs

```
    T1:
      if (thd->info) {
        ...
        use(thd->info);
        ...
      }

    T2:
      thd->info = NULL
```

  - order-violation bugs

```
    T1:
      void init() {
        thd = createThread(...)
      }

    T2:
      void main() {
        state = thd->info
      }
```

$T1 \rightarrow T2$

$T2 \rightarrow T1 \quad X$
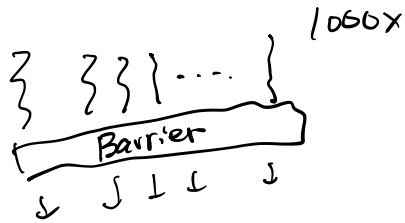
(c) performance vs. complexity

Focus on Correctness

Other synchronization mechanisms
- semaphores ←
- Barrier ←
- spinlocks ←
- reader-writer locks ←
- Read-copy-update (RCU) ←
- deterministic multithreading ←

{ mutex
{ c.v.

reader × 1M.

1000×

{ { { ... {
Barrier
↓ ↓ ↓ ↓   ↓

Only run

"correct" interleaving

```
 1    Implementation of spinlocks and mutexes
 2
 3    1. Here is a BROKEN spinlock implementation:
 4
 5        struct Spinlock {
 6            int locked;        ←  0: not locker , 1: locked
 7        }
 8
 9        void acquire(Spinlock *lock) {
10            while (1) {
11                if (lock->locked == 0) { // A
12                    lock->locked = 1; // B
13                    break;
14                }
15            }
16        }
17
18        void release (Spinlock *lock) {
19            lock->locked = 0;
20        }
21
22    What's the problem? Two acquire()s on the same lock on different
23    CPUs might both execute line A, and then both execute B. Then
24    both will think they have acquired the lock. Both will proceed.
25    That doesn't provide mutual exclusion.
26
27
```

```
28
29
30    2. Correct spinlock implementation
31
32    Relies on atomic hardware instruction. For example, on the x86-64,
33    doing
34        "xchg addr, %rax" ←
35    does the following:
36
37        (i) freeze all CPUs' memory activity for address addr
38        (ii) temp <-- *addr
39        (iii) *addr <-- %rax
40        (iv) %rax <-- temp
41        (v) un-freeze memory activity
42
43        /* pseudocode */
44        int xchg_val(addr, value) {
45            %rax = value;
46            xchg (*addr), %rax
47        }
48
49        /* bare-bones version of acquire */
50        void acquire (Spinlock *lock) {
51            pushcli(); /* what does this do? */      → disable interrupt
52            while (1) {
53                if (xchg_val(&lock->locked, 1) == 0)
54                    break;
55            }
56        }
57
58        void release(Spinlock *lock){
59            xchg_val(&lock->locked, 0);
60            popcli(); /* what does this do? */      → enable interrupt
61        }
62
63        /* optimization in acquire; call xchg_val() less frequently */
64        void acquire(Spinlock* lock) {
65            pushcli();
66            while (xchg_val(&lock->locked, 1) == 1) {
67                while (lock->locked) ;
68            }
69        }
70
71    The above is called a *spinlock* because acquire() spins. The
72    bare-bones version is called a "test-and-set (TAS) spinlock"; the
73    other is called a "test-and-test-and-set spinlock".
74
75    The spinlock above is great for some things, not so great for
76    others. The main problem is that it *busy waits*: it spins,
77    chewing up CPU cycles. Sometimes this is what we want (e.g., if
78    the cost of going to sleep is greater than the cost of spinning
79    for a few cycles waiting for another thread or process to
80    relinquish the spinlock). But sometimes this is not at all what we
81    want (e.g., if the lock would be held for a while: in those
82    cases, the CPU waiting for the lock would waste cycles spinning
83    instead of running some other thread or process).
84
85    NOTE: the spinlocks presented here can introduce performance issues
86    when there is a lot of contention. (This happens even if the
87    programmer is using spinlocks correctly.) The performance issues
88    result from cross-talk among CPUs (which undermines caching and
89    generates traffic on the memory bus). If we have time later, we will
90    study a remediation of this issue (search the Web for "MCS locks").
91
92    ANOTHER NOTE: In everyday application-level programming, spinlocks
93    will not be something you use (use mutexes instead). But you should
94    know what these are for technical literacy, and to see where the
95    mutual exclusion is truly enforced on modern hardware.
96
```

```
 97   3. Mutex implementation
 98
 99      The intent of a mutex is to avoid busy waiting: if the lock is not
100      available, the locking thread is put to sleep, and tracked by a
101      queue in the mutex. The next page has an implementation.
102
103      #include <sys/queue.h>
104
105      typedef struct thread {
106          // ... Entries elided.
107          STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
108      } thread_t;
109
110      struct Mutex {
111          // Current owner, or 0 when mutex is not held.
112          thread_t *owner;
113
114          // List of threads waiting on mutex
115          STAILQ(thread_t) waiters;
116
117          // A lock protecting the internals of the mutex.
118          Spinlock splock; // as in item 1, above
119      };
120
121      void mutex_acquire(struct Mutex *m) {
122
123          acquire(&m->splock);
124
125          // Check if the mutex is held; if not, current thread gets mutex and returns
126          if (m->owner == 0) {
127              m->owner = id_of_this_thread;
128              release(&m->splock);
129          } else {
130              // Add thread to waiters.
131              STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
132
133              // Tell the scheduler to add current thread to the list
134              // of blocked threads. The scheduler needs to be careful
135              // when a corresponding sched_wakeup call is executed to
136              // make sure that it treats running threads correctly.
137              sched_mark_blocked(&id_of_this_thread);
138
139              // Unlock spinlock.
140              release(&m->splock);
141
142              // Stop executing until woken.
143              sched_swtch();
144
145              // When we get to this line, we are guaranteed to hold the mutex. This
146              // is because we can get here only if context-switched-TO, which itself
147              // can happen only if this thread is removed from the waiting queue,
148              // marked "unblocked",/ and set to be the owner (in mutex_release()
149              // below). However, we might actually have held the mutex in lines 141-144
150              // (if we were context-switched out after the spinlock release(),
151              // followed by being run as a result of another thread's release of the
152              // mutex). But if that happens, it just means that we are
153              // context-switched out an "extra" time before proceeding.
154          }
155      }
156
```

```
157      void mutex_release(struct Mutex *m) {
158          // Acquire the spinlock in order to make changes.
159          acquire(&m->splock);
160
161          // Assert that the current thread actually owns the mutex
162          assert(m->owner == id_of_this_thread);
163
164          // Check if anyone is waiting.
165          m->owner = STAILQ_GET_HEAD(&m->waiters);
166
167          // If so, wake them up.
168          if (m->owner) {
169              sched_wakeone(&m->owner);
170              STAILQ_REMOVE_HEAD(&m->waiters, qlink);
171          }
172
173          // Release the internal spinlock
174          release(&m->splock);
175      }
```