

Sep 24, 25 0:07

spinlock-mutex.txt

Page 1/3

```

1 Implementation of spinlocks and mutexes
2
3 1. Here is a BROKEN spinlock implementation:
4
5     struct Spinlock {
6         int 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

```

Sep 24, 25 0:07

spinlock-mutex.txt

Page 2/3

```

26
27 2. Correct spinlock implementation
28
29     Relies on atomic hardware instruction. For example, on the x86-64,
30     doing
31         "xchg addr, %rax"
32     does the following:
33
34         (i) freeze all CPUs' memory activity for address addr
35         (ii) temp <-- *addr
36         (iii) *addr <-- %rax
37         (iv) %rax <-- temp
38         (v) un-freeze memory activity
39
40     /* pseudocode */
41     int xchg_val(addr, value) {
42         %rax = value;
43         xchg (*addr), %rax
44     }
45
46     /* bare-bones version of acquire */
47     void acquire (Spinlock *lock) {
48         pushcli(); /* what does this do? */
49         while (1) {
50             if (xchg_val(&lock->locked, 1) == 0)
51                 break;
52         }
53     }
54
55     void release(Spinlock *lock){
56         xchg_val(&lock->locked, 0);
57         popcli(); /* what does this do? */
58     }
59
60
61     /* optimization in acquire; call xchg_val() less frequently */
62     void acquire(Spinlock* lock) {
63         pushcli();
64         while (xchg_val(&lock->locked, 1) == 1) {
65             while (lock->locked) ;
66         }
67     }
68
69     The above is called a *spinlock* because acquire() spins. The
70     bare-bones version is called a "test-and-set (TAS) spinlock"; the
71     other is called a "test-and-test-and-set spinlock".
72
73     The spinlock above is great for some things, not so great for
74     others. The main problem is that it *busy waits*: it spins,
75     chewing up CPU cycles. Sometimes this is what we want (e.g., if
76     the cost of going to sleep is greater than the cost of spinning
77     for a few cycles waiting for another thread or process to
78     relinquish the spinlock). But sometimes this is not at all what we
79     want (e.g., if the lock would be held for a while: in those
80     cases, the CPU waiting for the lock would waste cycles spinning
81     instead of running some other thread or process).
82
83     NOTE: the spinlocks presented here can introduce performance issues
84     when there is a lot of contention. (This happens even if the
85     programmer is using spinlocks correctly.) The performance issues
86     result from cross-talk among CPUs (which undermines caching and
87     generates traffic on the memory bus). To learn about a remediation of this
88     issue, search the Web for "MCS locks".
89     https://www.cs.rochester.edu/u/scott/papers/1991\_TOCS\_synch.pdf
90
91     ANOTHER NOTE: In everyday application-level programming, spinlocks
92     will not be something you use (use mutexes instead). But you should
93     know what these are for technical literacy, and to see where the
94     mutual exclusion is truly enforced on modern hardware.
95

```

Sep 24, 25 0:07

spinlock-mutex.txt

Page 3/3

96 3. Mutex implementation

97
98
99
100
101
102

The intent of a mutex is to avoid busy waiting: if the lock is not available, the locking thread is put to sleep, and tracked by a queue in the mutex. The next page has an implementation.

Sep 24, 25 0:07

fair-mutex.c

Page 1/1

```

1  #include <sys/queue.h>
2
3  typedef struct thread {
4      // ... Entries elided.
5      STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
6  } thread_t;
7
8  struct Mutex {
9      // Current owner, or 0 when mutex is not held.
10     thread_t *owner;
11
12     // List of threads waiting on mutex
13     STAILQ(thread_t) waiters;
14
15     // A lock protecting the internals of the mutex.
16     Spinlock splock; // as in item 1, above
17 };
18
19 void mutex_acquire(struct Mutex *m) {
20
21     acquire(&m->splock);
22
23     // Check if the mutex is held; if not, current thread gets mutex and returns
24     if (m->owner == 0) {
25         m->owner = id_of_this_thread;
26         release(&m->splock);
27     } else {
28         // Add thread to waiters.
29         STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
30
31         // Tell the scheduler to add current thread to the list
32         // of blocked threads. The scheduler needs to be careful
33         // when a corresponding sched_wakeup call is executed to
34         // make sure that it treats running threads correctly.
35         sched_mark_blocked(&id_of_this_thread);
36
37         // Unlock spinlock.
38         release(&m->splock);
39
40         // Stop executing until woken.
41         sched_swch();
42
43         // When we get to this line, we are guaranteed to hold the mutex. This
44         // is because we can get here only if context-switched-TO, which itself
45         // can happen only if this thread is removed from the waiting queue,
46         // marked "unblocked", and set to be the owner (in mutex_release()
47         // below). However, we might have held the mutex in lines 39-42
48         // (if we were context-switched out after the spinlock release(),
49         // followed by being run as a result of another thread's release of the
50         // mutex). But if that happens, it just means that we are
51         // context-switched out an "extra" time before proceeding.
52     }
53 }
54
55 void mutex_release(struct Mutex *m) {
56     // Acquire the spinlock in order to make changes.
57     acquire(&m->splock);
58
59     // Assert that the current thread actually owns the mutex
60     assert(m->owner == id_of_this_thread);
61
62     // Check if anyone is waiting.
63     m->owner = STAILQ_GET_HEAD(&m->waiters);
64
65     // If so, wake them up.
66     if (m->owner) {
67         sched_wakeone(&m->owner);
68         STAILQ_REMOVE_HEAD(&m->waiters, qlink);
69     }
70
71     // Release the internal spinlock
72     release(&m->splock);
73 }

```

Sep 24, 25 0:07

deadlock.txt

Page 1/3

```

1  Deadlock examples
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

```

Sep 24, 25 0:07

deadlock.txt

Page 2/3

```

23  2. More subtle deadlock example
24
25      Let M be a monitor (shared object with methods protected by mutex)
26      Let N be another monitor
27
28      class M {
29          private:
30              Mutex mutex_m;
31
32              // instance of monitor N
33              N another_monitor;
34
35              // Assumption: no other objects in the system hold a pointer
36              // to our "another_monitor"
37
38          public:
39              M();
40              ~M();
41              void methodA();
42              void methodB();
43      };
44
45      class N {
46          private:
47              Mutex mutex_n;
48              Cond cond_n;
49              int navailable;
50
51          public:
52              N();
53              ~N();
54              void* alloc(int nwanted);
55              void free(void*);
56      }
57
58      int
59      N::alloc(int nwanted) {
60          acquire(&mutex_n);
61          while (navailable < nwanted) {
62              wait(&cond_n, &mutex_n);
63          }
64
65          // peel off the memory
66
67          navailable -= nwanted;
68          release(&mutex_n);
69      }
70
71      void
72      N::free(void* returning_mem) {
73          acquire(&mutex_n);
74
75          // put the memory back
76
77          navailable += returning_mem;
78          broadcast(&cond_n, &mutex_n);
79
80          release(&mutex_n);
81      }
82
83
84

```

Sep 24, 25 0:07

deadlock.txt

Page 3/3

```
85 void
86 M::methodA() {
87
88     acquire(&mutex_m);
89
90     void* new_mem = another_monitor.alloc(int nbytes);
91
92     // do a bunch of stuff using this nice
93     // chunk of memory n allocated for us
94
95     release(&mutex_m);
96 }
97
98 void
99 M::methodB() {
100
101     acquire(&mutex_m);
102
103     // do a bunch of stuff
104
105     another_monitor.free(some_pointer);
106
107     release(&mutex_m);
108 }
109
110 QUESTION: What's the problem?
```