```
CS 202, Spring 2024
Handout 6 (Class 7)

1. This monitor is a model of a database with multiple readers and
writers. The high-level goal here is (a) to give a writer exclusive
access (a single active writer means there should be no other writers
and no readers) while (b) allowing multiple readers. Like the previous
example, this one is expressed in pseudocode.

    // assume that these variables are initialized in a constructor
    state variables:
        AR = 0;  // # active readers
        AW = 0;  // # active writers
        WR = 0;  // # waiting readers
        WW = 0;  // # waiting writers

        Condition okToRead = NIL;
        Condition okToWrite = NIL;
        Mutex mutex = FREE;

    Database::read() {
        startRead();  // first, check self into the system
        Access Data
        doneRead();   // check self out of system
    }

    Database::startRead() {
        acquire(&mutex);
        while((AW + WW) > 0){
            WR++;
            wait(&okToRead, &mutex);
            WR--;
        }
        AR++;
        release(&mutex);
    }

   Database::doneRead() {
        acquire(&mutex);
        AR--;
        if (AR == 0 && WW > 0) { // if no other readers still
          signal(&okToWrite, &mutex);   // active, wake up writer
        }
        release(&mutex);
    }

    Database::write(){  // symmetrical
        startWrite();  // check in
        Access Data
        doneWrite();  // check out
    }

    Database::startWrite() {
        acquire(&mutex);
        while ((AW + AR) > 0) { // check if safe to write.
                                // if any readers or writers, wait
            WW++;
            wait(&okToWrite, &mutex);
            WW--;
        }
        AW++;
        release(&mutex);
    }

    Database::doneWrite() {
        acquire(&mutex);
        AW--;
        if (WW > 0) {
            signal(&okToWrite, &mutex); // give priority to writers
        } else if (WR > 0) {
            broadcast(&okToRead, &mutex);
        }
        release(&mutex);
```



```
    }

    NOTE: what is the starvation problem here?

```

```
2. Shared locks

    struct sharedlock {
      int i;
      Mutex mutex;
      Cond c;
    };

    void AcquireExclusive (sharedlock *sl) {
      acquire(&sl->mutex);
      while (sl->i) {
        wait (&sl->c, &sl->mutex);
      }
      sl->i = -1;
      release(&sl->mutex);
    }

    void AcquireShared (sharedlock *sl) {
      acquire(&sl->mutex);
      while (sl->i < 0) {
        wait (&sl->c, &sl->mutex);
      }
      sl->i++;
      release(&sl->mutex);
    }

    void ReleaseShared (sharedlock *sl) {
      acquire(&sl->mutex);
      if (!--sl->i)
        signal (&sl->c, &sl->mutex);
      release(&sl->mutex);
    }

    void ReleaseExclusive (sharedlock *sl) {
      acquire(&sl->mutex);
      sl->i = 0;
      broadcast (&sl->c, &sl->mutex);
      release(&sl->mutex);
    }

    QUESTIONS:
    A. There is a starvation problem here. What is it? (Readers can keep
       writers out if there is a steady stream of readers.)
    B. How could you use these shared locks to write a cleaner version
       of the code in the prior item? (Though note that the starvation
       properties would be different.)
```



```
Implementation of spinlocks and mutexes

1. Here is a BROKEN spinlock implementation:

        struct Spinlock {
          int locked;
        }

        void acquire(Spinlock *lock) {
          while (1) {
            if (lock->locked == 0) { // A
              lock->locked = 1;      // B
              break;
            }
          }
        }

        void release (Spinlock *lock) {
          lock->locked = 0;
        }

        What's the problem? Two acquire()s on the same lock on different
        CPUs might both execute line A, and then both execute B. Then
        both will think they have acquired the lock. Both will proceed.
        That doesn't provide mutual exclusion.

```

```
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). If we have time later, we will
88      study a remediation of this issue (search the Web for "MCS locks").
89
90      ANOTHER NOTE: In everyday application-level programming, spinlocks
91      will not be something you use (use mutexes instead). But you should
92      know what these are for technical literacy, and to see where the
93      mutual exclusion is truly enforced on modern hardware.
94
```



```
95  3. Mutex implementation
96
97      The intent of a mutex is to avoid busy waiting: if the lock is not
98      available, the locking thread is put to sleep, and tracked by a
99      queue in the mutex. The next page has an implementation.
100
101
```

```c
#include <sys/queue.h>

typedef struct thread {
    // ... Entries elided.
    STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
} thread_t;

struct Mutex {
    // Current owner, or 0 when mutex is not held.
    thread_t *owner;

    // List of threads waiting on mutex
    STAILQ(thread_t) waiters;

    // A lock protecting the internals of the mutex.
    Spinlock splock;   // as in item 1, above
};

void mutex_acquire(struct Mutex *m) {

    acquire(&m->splock);

    // Check if the mutex is held; if not, current thread gets mutex and returns
    if (m->owner == 0) {
        m->owner = id_of_this_thread;
        release(&m->splock);
    } else {
        // Add thread to waiters.
        STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);

        // Tell the scheduler to add current thread to the list
        // of blocked threads. The scheduler needs to be careful
        // when a corresponding sched_wakeup call is executed to
        // make sure that it treats running threads correctly.
        sched_mark_blocked(&id_of_this_thread);

        // Unlock spinlock.
        release(&m->splock);

        // Stop executing until woken.
        sched_swtch();

        // When we get to this line, we are guaranteed to hold the mutex. This
        // is because we can get here only if context-switched-TO, which itself
        // can happen only if this thread is removed from the waiting queue,
        // marked "unblocked", and set to be the owner (in mutex_release()
        // below). However, we might have held the mutex in lines 39-42
        // (if we were context-switched out after the spinlock release(),
        // followed by being run as a result of another thread's release of the
        // mutex). But if that happens, it just means that we are
        // context-switched out an "extra" time before proceeding.
    }
}

void mutex_release(struct Mutex *m) {
    // Acquire the spinlock in order to make changes.
    acquire(&m->splock);

    // Assert that the current thread actually owns the mutex
    assert(m->owner == id_of_this_thread);

    // Check if anyone is waiting.
    m->owner = STAILQ_GET_HEAD(&m->waiters);

    // If so, wake them up.
    if (m->owner) {
        sched_wakeone(&m->owner);
        STAILQ_REMOVE_HEAD(&m->waiters, qlink);
    }

    // Release the internal spinlock
    release(&m->splock);
}
```



```
Deadlock examples

1. Simple deadlock example

    T1:
        acquire(mutexA);
        acquire(mutexB);

        // do some stuff

        release(mutexB);
        release(mutexA);

    T2:
        acquire(mutexB);
        acquire(mutexA);

        // do some stuff

        release(mutexA);
        release(mutexB);

```

```
2. More subtle deadlock example

    Let M be a monitor (shared object with methods protected by mutex)
    Let N be another monitor

    class M {
        private:
            Mutex mutex_m;

            // instance of monitor N
            N another_monitor;

            // Assumption: no other objects in the system hold a pointer
            // to our "another_monitor"

        public:
            M();
            ~M();
            void methodA();
            void methodB();
    };

    class N {
        private:
            Mutex mutex_n;
            Cond cond_n;
            int navailable;

        public:
            N();
            ~N();
            void* alloc(int nwanted);
            void  free(void*);
    }

    int
    N::alloc(int nwanted) {
        acquire(&mutex_n);
        while (navailable < nwanted) {
            wait(&cond_n, &mutex_n);
        }

        // peel off the memory

        navailable -= nwanted;
        release(&mutex_n);
    }

    void
    N::free(void* returning_mem) {

        acquire(&mutex_n);

        // put the memory back

        navailable += returning_mem;

        broadcast(&cond_n, &mutex_n);

        release(&mutex_n);
    }

```



```
    void
    M::methodA() {

        acquire(&mutex_m);

        void* new_mem = another_monitor.alloc(int nbytes);

        // do a bunch of stuff using this nice
        // chunk of memory n allocated for us

        release(&mutex_m);
    }

    void
    M::methodB() {

        acquire(&mutex_m);

        // do a bunch of stuff

        another_monitor.free(some_pointer);

        release(&mutex_m);
    }

    QUESTION: What's the problem?
```