The previous handout demonstrated the use of mutexes and condition variables. This handout demonstrates the use of monitors (which combine mutexes and condition variables).

1. The bounded buffer as a monitor

```cpp
class MyBuffer {
public:
    MyBuffer();
    ~MyBuffer();
    void Enqueue(Item);
    Item Dequeue();
private:
    int count;
    int in;
    int out;
    Item buffer[BUFFER_SIZE];
    Mutex* mutex;
    Cond* nonempty;
    Cond* nonfull;
};
```

```cpp
void MyBuffer::MyBuffer()
{
    in = out = count = 0;
    mutex = new Mutex;
    nonempty = new Cond;
    nonfull = new Cond;
}
```

```cpp
void MyBuffer::Enqueue(Item item)
{
    mutex.acquire();
    while (count == BUFFER_SIZE)
        cond_wait(&nonfull, &mutex);
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
    cond_signal(&nonempty, &mutex);
    mutex.release();
}
```

```cpp
Item MyBuffer::Dequeue()
{
    mutex.acquire();
    while (count == 0)
        cond_wait(&nonempty, &mutex);
    Item ret = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;
    cond_signal(&nonfull, &mutex);
    mutex.release();
    return ret;
}
```

2. Key point: *Threads* (the producer and consumer) are separate from *shared object* (MyBuffer). The synchronization happens in the shared object.
2. 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.

```c
// 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;
```

```c
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?
```

3. Shared locks

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

```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:

   ```c
   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;
   }

   /* pseudocode */
   int cmpxchg(addr, comprand, value) {
      if *addr == comprand {
         *addr = value;
      }
      return *addr
   }

   /* bare-bones version of acquire */
   void acquire(Spinlock *lock) {
      pushcli();    /* what does this do? */
      while (cmpxchg(&lock->locked, 0, TID) != TID) // yield
          yield();
      popcli();    /* what does this do? */
      }
   }

   void release(Spinlock *lock) {
      cmpxchg(&lock->locked, TID, 0);
      popcli();    /* what does this do? */
   }

   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.

2. Correct spinlock implementation

   Relies on atomic hardware instruction. For example, on the x86-64,
   doing
   
   "lock cmpxchg addr, $1, %rax"

   does the following:
   
   (i) freeze all CPUs’ memory activity for address addr
   (ii) if *addr = 1 then *addr = %rax
   (iii) return *addr

   /* bare-bones version of acquire */
   void acquire(Spinlock *lock) {
      pushcli();    /* what does this do? */
      while (cmpxchg(&lock->locked, 0, TID) != TID)
          // yield
      }
      popcli();    /* what does this do? */
   }

   the above is called a *spinlock* because acquire() spins. The
   bare-bones version is called a "test-and-set (TAS) spinlock"; the
   other is called a "test-and-test-and-set spinlock".

   The spinlock above is great for some things, not so great for
   others. The main problem is that it *busy waits*; it spins,
   chewing up CPU cycles. Sometimes this is what we want (e.g., if
   the cost of going to sleep is greater than the cost of spinning
   for a few cycles waiting for another thread or process to
   relinquish the spinlock). But sometimes this is not at all what we
   want (e.g., if the lock would be held for a while; in those
   cases, the CPU waiting for the lock would waste cycles spinning
   instead of running some other thread or process).

   NOTE: the spinlocks presented here can introduce performance issues
   when there is a lot of contention. (This happens even if the
   programmer is using spinlocks correctly.) The performance issues
   result from cross-talk among CPUs (which undermines caching and
   generates traffic on the memory bus). If we have time later, we will
   study a remediation of this issue (search the Web for "MCS locks").

   ANOTHER NOTE: In everyday application-level programming, spinlocks
   will not be something you use (use mutexes instead). But you should
   know what these are for technical literacy, and to see where the
   mutual exclusion is truly enforced on modern hardware.
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.

```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();
    }
}

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);
}