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;
}  

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

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();
}

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;
}

int main(int, char**)
{
  MyBuffer buf;
  int dummy;
  tid1 = thread_create(producer, &buf);
  tid2 = thread_create(consumer, &buf);
  // never reach this point
  thread_join(tid1);
  thread_join(tid2);
  return -1;
}

void producer(void* buf)
{
  MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
  for (;;) {
    // next line produces an item and puts it in nextProduced */
    Item nextProduced = means_of_production();
    sharedbuf->Enqueue(nextProduced);
  }
}

void consumer(void* buf)
{
  MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
  for (;;) {
    Item nextConsumed = sharedbuf->Dequeue();
    /* next line abstractly consumes the item */
    consume_item(nextConsumed);
  }
}

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.

```plaintext
// 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
    } else {
        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 (AR == 0 && WW > 0) { // if no other readers still
            signal(&okToWrite, &mutex);  // active, wake up writer
        } else {
            release(&mutex);
        }
    }
    AW++;
    while ((AW + AR) > 0) { // check if safe to write.
        if (AR == 0 && WW > 0) { // if no other readers still
            signal(&okToWrite, &mutex);  // active, wake up writer
        } else if (WW > 0) { // give priority to writers
            broadcast(&okToRead, &mutex);
        } else if (WR > 0) { // give priority to readers
            broadcast(&okToWrite, &mutex);
        } else {
            release(&mutex);
        }
    }
}

DATABASE::doneWrite() {
    acquire(&mutex);
    AW--; // signal(&okToWrite, &mutex);  // give priority to writers
}
```

3. Shared locks

```plaintext
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 item 2., above? (Though note that the starvation properties would be different.)