Queuing Mutex Components:

- **mutex.state** - keeps track of the mutex's state: (mutex.state values: 0 ≡ locked, 1 ≡ unlocked)
- **mutext.waiters** - keeps the list of waiting threads
- **mutext.spinlock** - cache-conscious spinlock used to protect queuing mutex state from race conditions

1) Try to complete the "**Code to Unlock mutex**" below:

Code to Lock mutex:	**Code to Unlock mutex:**

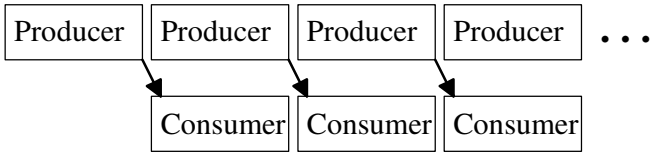
lock mutex.spinlock
if mutex.state == 1 then
  mutex.state = 0
  unlock mutex.spinlock
else
  add current thread to mutex.waiters
  remove current thread from runnable threads
  unlock mutex.spinlock
  yield to a runnable thread	

2) A common synchronization pattern is *bounded buffer* where two threads are linked by a processing pipeline where:

- a producer thread produces some data, and
- a consumer thread consumes the data

Without concurrency they could alternate:

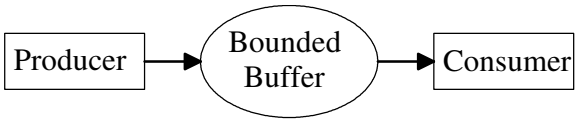
With concurrency they could pipeline:



a) What is the advantage of the pipelined version?

b) In the pipelined version, how does the rate the producer produces data compare to the rate the consumer consumes it?

3) A bounded buffer can be used to store some data between the producer and consumer threads:



a) What is the advantage of the bounded buffer approach over pipelining the producer data directly to the consumer thread?

b) In the bounded buffer approach when would the producer thread need to block?

c) In the bounded buffer approach when would the consumer thread need to block?

4) Another common synchronization pattern is *readers/writers locks* which are similar to mutexs, except threads specify whether they are locking for writing (updating) or only reading the shared data structure.

a) Since writer threads must update the shared data structure mutually exclusively, what concurrency would readers/writers locks provide?

b) Consider the scenario where one (or more) reader thread(s) hold the lock and there are waiting writer thread(s) when a new reader thread "arrives." Two possibilities are:

- Allow the new reader to immediately start reading
- Make the new reader wait until after the waiting writers

What are the advantages and disadvantages of each approach?

c) Many systems including POSIX provide an implementation of readers/writers locks (`pthread_rwlock_init`, `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`). However, the choice of prioritizing a new reader vs. waiting writer(s) is open to system implementation. What are the advantages and disadvantages of not specifying this choice?

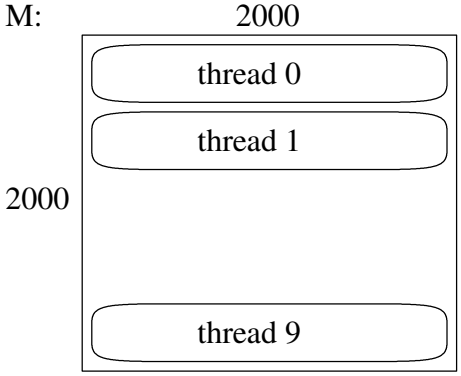
d) POSIX has specialized forms of readers/writers locks for files: `fcntl` procedure (complex) and simpler `flock` interface. Why are files likely choice for applying readers/writers locks toward?

5) Another common synchronization pattern are *synchronization barriers* where we want all threads to finish some phase of a task before any can continue. Often used in scientific calculuations on large matrices (2D arrays). Barrier are initialized with the number of threads to wait for. As threads arrive at the barrier and issue a "wait," all threads are blocked until the last thread performs a wait.

Sequential code:

```
for time = 0 to 1000 do
    for row = 0 to 1999 do
        for column = 0 to 1999 do
            M[row, column] = ...
```

Might assign 10 threads each 200 rows to calculate, but the all need to complete a time interval before any thread can move on to the next time interval.



Outline the multi-thread code focusing on where the barrier initialization and wait would be performed.

To implement these synchronization patterns we need mechanisms (tools) to allow threads to wait until circumstances are appropriate for it to proceed. Two common mechanisms are:

- *condition variables* used with monitors (or mutexes used in a monitor style) have operations
  - wait - executed by thread finding circumstances not to its liking, so it sleeps until another thread does a notify operation
  - notify (or signal) - moves a waiting thread to the run queue
- *semaphores* - more specialized (and older) tool which is basically an unsigned integer with operations restricted to:
  - initialized when it is created to programmer's specified nonnegative integer value
  - release (or V) operation - increment the semaphore by 1
  - acquire (or P) operation - try to decrement the semaphore by 1, but waits if the semaphore value is 0 since it cannot go negative. Once another thread has performed a release operation to increment the semaphore the a waiting thread can continue its release operation

Java - every object in Java has an associated mutex and condition variable with methods:

- wait - waits on the object's condition variable
- notifyAll - wakes up all threads waiting
- notify - wakes up only one thread

All must be called by a thread holding the object's mutex

Notes:

- A thread holding the mutex and issuing a wait appears to deadlock since it would prevent another threads from acquiring the mutex to perform a notify or notifyAll. However, the wait operation releases the mutex before putting the thread into a wait state
- When a waiting thread is awoken, it reacquires the mutex before the wait operation returns. (Java has recursive mutexes which have a counter for the number of times a thread has locked the mutex while holding the lock. When a thread invokes the unlock, the counter is decremented, and only when it reaches 0 is the mutex really unlocked.)

```
public class BoundedBuffer {
  private Object[] buffer = new Object[20]; // arbitrary size
  private int numOccupied = 0;
  private int firstOccupied = 0;

  /* invariant: 0 <= numOccupied <= buffer.length
     0 <= firstOccupied < buffer.length
     buffer[(firstOccupied + i) % buffer.length]
     contains the (i+1)th oldest entry,
     for all i such that 0 <= i < numOccupied  */

  public synchronized void insert(Object o)
    throws InterruptedException
  {
    while(numOccupied == buffer.length)
      // wait for space
      wait();
    buffer[(firstOccupied + numOccupied) % buffer.length] = o;
    numOccupied++;
    // in case any retrieves are waiting for data, wake them
    notifyAll();
  }

  public synchronized Object retrieve()
    throws InterruptedException
  {
    while(numOccupied == 0)
      // wait for data
      wait();
    Object retrieved = buffer[firstOccupied];
    buffer[firstOccupied] = null; // may help garbage collector
    firstOccupied = (firstOccupied + 1) % buffer.length;
    numOccupied--;
    // in case any inserts are waiting for space, wake them
    notifyAll();
    return retrieved;
  }
}
```

Figure 4.17: BoundedBuffer class using monitors and condition variables