Andrew Onozuka
CSE 120 | A01
Prof. Geoffrey Voelker
Due: Saturday October 28 at 11:59pm

1. XCHG can be used instead of test and set to implement the acquire() and release() functions of the spinlock data structure. It may look something like this:

```
struct lock {
   bool flag; // A boolean flag indicating whether the lock is acquired
}

void acquire(struct lock *lock) {
   bool acquired;

   // Try to acquire the lock using XCHG
   do {
      acquired = true; // Initially assume we acquired the lock
      XCHG(&acquired, &lock->flag); // Atomically swap the values

      // If acquired is now false, another thread holds the lock,
      // so we need to spin and wait.
      while (!acquired) {
         // Busy-wait or yield CPU time to avoid spinning too much
      }
   } while (!acquired);
}

void release(struct lock *lock) {
   // Simply set the lock's flag to false to release the lock
   lock->flag = false;
}
```

The acquire function uses the XCHG instruction to atomically swap a flag to indicate the lock is acquired; if the flag was already set, the current thread enters a spin loop until the lock is released. The release function simply sets the lock's flag to indicate it's released, enabling another thread to acquire the lock. This use of XCHG ensures that both the acquire and release operations are atomic, providing mutual exclusion for critical sections in a spinlock, effectively replacing the traditional test-and-set operation.

2. a. Context switches: main → thread A → thread B → thread A

   b. fee
      foe

foo
far
fie
fum
fun

c. currentThread: main thread
readyQueue: empty
join wait queue: empty

The main thread is the current thread when ‘selfTest’ returns. The ready queue and join wait queue are both empty as there are no more threads left to execute or join at this point.

3. a.

```
monitor Barrier {
   int count = 0; // Number of threads that have called Done

   condition waitCondition; // Condition variable for waiting threads

   void Done(int n) {
      count++;

      if (count < n) {
         wait(waitCondition); // Wait until all threads have called Done
      } else {
         count = 0; // Reset the count for the next phase
         notifyAll(waitCondition); // Signal all waiting threads to proceed
      }
   }
}
```

b.

```
class Barrier {
   int count = 0; // Number of threads that have called Done
   int n; // Total number of threads

   Lock lock; // Lock for mutual exclusion
   Condition condition; // Condition variable for waiting threads

   Barrier(int n) {
      this.n = n;
      this.count = 0;
      this.lock = new Lock();
      this.condition = new Condition();
   }
```

```
    void Done() {
        lock.acquire(); // Acquire the lock to ensure mutual exclusion

        count++;

        if (count < n) {
            condition.Wait(lock); // Wait until all threads have called Done
        } else {
            count = 0; // Reset the count for the next phase
            condition.Broadcast(lock); // Signal all waiting threads to proceed
        }

        lock.release(); // Release the lock
    }
}
```

4.

```
class Surfing {
    enum State { calm, breaking; }
    enum Direction { LEFT, RIGHT, BOTH; }

    State oceanState = calm;
    Direction waveDirection = BOTH;

    Lock lock;
    Condition surfersWaiting;
    Condition oceanWaiting;

    Surfing () {
        lock = new Lock();
        surfersWaiting = new Condition(lock);
        oceanWaiting = new Condition(lock);
    }

    void paddle(Direction dir) {
        lock.Acquire();

        if (oceanState == calm || dir == waveDirection || waveDirection == BOTH) {
            // Surf the wave or wait for the next one
            surfersWaiting.Wait();
        }

        lock.Release();
```

```
    }

    void wave(Direction dir) {
        lock.Acquire();

        oceanState = breaking;
        waveDirection = dir;

        // Wake up all surfers if the wave is breaking in their direction or for BOTH
        surfersWaiting.Broadcast();

        lock.Release();
    }

    void done() {
        lock.Acquire();

        oceanState = calm;
        waveDirection = BOTH;

        // Notify the ocean thread that the wave is done
        oceanWaiting.Signal();

        lock.Release();
    }
}
```

5.

Eleanor has a thesaurus and needs the dictionary.
Eleanor -> Thesaurus
Eleanor <- Dictionary

Chidi has a thesaurus and a coffee cup.
Chidi -> Thesaurus
Chidi -> Coffee Cup 1

Tahani has the dictionary and needs a thesaurus.
Tahani -> Dictionary
Tahani <- Thesaurus

Jason has a coffee cup and needs another coffee cup.
Jason -> Coffee Cup 1
Jason <- Coffee Cup 2
The system is not deadlocked because there are no cycles in the current state.