MCS Locks

(A high performance, cache-aware implementation of general R/W spinlocks. This code is copied from the texgt (Figure 6.5, page 275). It doesn't worry about disabling interrupts.)

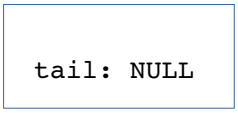
```
class MCSLock {
      private:
            Queue *tail = NULL;
}
MCSLock::release() {
      if ( CAS(&tail, myTCB, NULL)) {
            // no one is waiting, and we just freed the lock
      } else {
            // hand lock to next waiting thread
            while (myTCB->next == NULL ) ;
            myTCB->next->needToWait = FALSE;
}
MCSLock::acquire() {
      Queue *oldTail = tail;
      myTCB->next = NULL;
      while ( !CAS(&tail, oldTail, &myTCB) ) {
            oldTail = tail;
      }

      // if oldTail == NULL, we've acquire the lock
      // otherwise, wait for it

      if ( oldTail != NULL ) {
            myTCB->needToWait = TRUE;
            memory_barrier();
            oldTail->next = myTCB;
            while ( myTCB->needToWait ) ;
      }

}
```

A free MCSLock looks like this:



MCSLock Usage Example

Draw the state of the lock after the following operations have taken place:

Thread	**Action**
A	acquire()
B	acquire()
C	acquire()
A	release()
B D	release() acquire()

(For the last step, the threads alternate execution and complete a single executable source line before the other thread executes one of its lines.)

- Which line(s) of code set the lock's state back to the initial, free state shown on the previous page?
- Suppose a thread holding the lock crashes. What happens to waiting threads?

RCU (Read-Copy-Update) Locks

(General R/W spinlocks designed for very low overhead read locking, at the cost of latency for write operations. Relies on special assistance from the scheduler. Is applicable only in situations in which write operations can be performed atomically by a single memory write. Adapted from Figure 6.12 (page 281) of the text.)

```
class RCULock {
  private:
      Spinlock globalSpin;
      long globalCounter = 0;
      static long quiescentCount[NPROCESSORS];
      Spinlock writerSpin;
}
void RCULock::ReadLock() { disableInterrupts(); }
void RCULock::ReadUnlock() { enableInterrupts(); }
void RCULock::writeLock() { writerSpin.acquire(); }
void RCULock::writeUnlock() { writerSpin.release(); }
// called (only when holding write lock) to perform an update
void RCULock::publish(void **pp1, void *p2) {
  memory_barrier();
  *pp1 = p2;
  memory_barrier();
}
// wait until you're sure there are no readers who started before you wrote
void RCULock::synchronize() {
      int p,c;
      globalSpin.acquire();  c = ++globalCounter;  globalSpin.release();
      for ( p=0; p<NPROCESSORS; p++ )
            while ( quiescentCount[p] < c ) sleep(10);
      }

}
// called regularly by kernel scheduler
void RCULock::QuiescentState() {
      memory_barrier();
      for (int p=0; p<NPROCESSORS; p++)
            quiescentCount[p] = globalCounter;
      memory_barrier();
}
```

RCU Lock Exercise

Sketch the implementation of a LIFO queue with operations put(), get(), and find().

What happens if globalCounter wraps (back to 0)?

Treiber's non-blocking stack

(Adapted from Appendix D of Non-blocking Synchronization and System Design.)

```
Structure pointer_t { ptr: pointer to node, count: unsigned int}

Push(*stack, *entry) {
  pointer_t  old_top;
  do {
      old_top = stack->top;
      entry->next.ptr = old_top.ptr;
  } while ( !CAS(&(stack->top), old_top, <entry, old_top.count>) );
}


entry *Pop(*stack) {
  pointer_t old_top;
  entry *top;
  do {
      old_top = stack->top;
      top = old_top.ptr;
      if ( top == NULL ) return FALSE;
  } while ( !CAS(&(stack->top), old_top, <top->next.ptr, old_top.count+1>) );
  return top;
}
```

- *Why do we CAS a pointer and counter, rather than just a pointer?*
- *What happens if a Push() and a Pop() are executed concurrently by two threads?*