SOLUTION NOTES

Advanced Systems Topics 2003 Paper 9 Question 5 (TLH)

This question relates to the four lectures given by Tim Harris on Scalable Synchronization.

(*a*) A simple spin-lock can be implemented by a loop that repeatedly performs a `CAS` operation on a shared memory location that indicates the status of the lock. For instance if `p` is the address of this location, `false` indicates that the lock is available and `true` indicates that it is held:

```
void initialize()
{
  *p = false;
}

void lock()
{
  while (CAS (p, false, true) == true)
  {
    /* Nothing */
  }
}

void unlock()
{
  *p = false;
}
```

(*b*) Attempting a `CAS` operation on a location will not allow it to be held in *shared* mode in common caching systems, so if several processors are spinning then the cache line will ping-pong between them. The increased memory traffic may degrade the performance of other tasks on the system and may increase the latency in transfering the lock from one processor to another. [A better initial alternative would be a “test-and-cas” lock in which threads spin performing reads – this gives better practical performance but still a stampede of `CAS` attempts in a large system.]

(*c*) *Sections (c) and (d) are following through the development of an MCS queue-based spin-lock which was covered in detail in the lectures.*

```
QNode pushTail (QNode q)
{
  QNode predecessor = *l;
  q.next = NULL;

  while (CAS (l, predecessor, q) != predecessor)
  {
    predecessor = *l;
  }

  predecessor.next = q;

  return predecessor;
}

QNode popHead (QNode q)
{
  QNode successor = q.next;

  if (successor == null)
  {
    if (CAS (l, q, null) == q)
    {
      return null;
    }

    while (successor == null)
    {
      successor = q.next;
    }
  }

  return successor;
}
```

(*d*) A queue based lock allows each thread to spin on a separate memory location which can remain cached locally (and ideally this spinning can be done using ordinary read operations rather than read-modify-write).

```
void initialize()
{
  *l = null;
}

void lock(QNode q)
{
  QNode predecessor;
  q.value = false;
  predecessor = pushTail (q);
  if (predecessor != null)
  {
    while (q.value == false)
    {
      /* Nothing */
    }
  }
}

void unlock(QNode q)
{
  QNode successor;
  successor = popHead (q);
  if (successor != null)
  {
    successor.value = true;
  }
}
```