Synchronization

Part 1: Synchronization - Locks

Dekker's Algorithm and the Bakery Algorithm provide software-only synchronization. Thanks to advancements in hardware, synchronization approaches have been simplified and have significantly improvements in performance. One basis for synchronization is a simple atomic primitive, TestAndSet.

`TestAndSet(&x)` = Set a 1 in location of x, return old value

You may assume sequential consistency throughout these exercises.

1) If a location is initialized to 0, what will the first invocation of TestAndSet on that location return?

2) If 8 threads all call TestAndSet on the same location, how many different values will be seen by the threads?

3) Would 1 of the 8 threads receive a unique value, i.e. one that no other thread received?

4) Given the following code for ReleaseLock(Lock* L), provide an implementation for AcquireLock(Lock* L) using TestAndSet:

```
ReleaseLock(Lock* L):              AcquireLock(Lock* L):
      *L = 0                             While (________________):
                                               // SPIN
```

5) Executing TestAndSet requires the M coherence state. If 8 threads all call AcquireLock on the same lock and there are no calls to ReleaseLock, how many GetRdX and GetRd requests have been made after every thread has executed the first line in your AcquireLock code once?

6) Implement AcquireLock, so that (after the first attempt) it only tries to modify the lock if it is available.

```
AcquireLock(Lock* L):
    While (_________):
        While (_____________):
            // SPIN
```

7) How long a thread should wait before trying to acquire the lock varies depends. Waiting too long results in the wasted cycles, while not long enough results in extra coherence requests. What is the shortest time a thread should wait?

8) Operating systems can context switch threads when one thread is blocked. Is a thread waiting on a lock, blocked? If a context switch costs X cycles, how long should a thread wait for a lock before it yields to the operating system?

9) If we have a range of values (Q7 and Q8), what is an algorithm that will find the correct waiting time with minimal coherence requests, and no more than half of the cycles wasted?

10) Copy down what the two features are known as:

Q6 - ______________________________

Q9 - ______________________________

Part 2 – Synchronization – Queued Spinlocks

Queued locks provide better guarantees, under certain scenarios than the wait algorithm from Q9. Queued locks rely on a different synchronization primitive, compare-and-swap / compare-exchange (`CMPXCHG`):

```
int CMPXCHG (int* loc, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *loc;
  if (old_reg_val == oldval)
     *loc = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

11) Describe when compare and swap updates the provided location.

12) Provide an implementation of TestAndSet using `CMPXCHG`.

```
TestAndSetWithXchg(int* loc):

      return CMPXCHG(loc, ____, _____);
```

13) Sketch the execution of two threads calling `TestAndSetWithXchg()` on the same location.

Given the Acquire and Release routines for a queued lock:

```
AcquireQLock(*glock, *mlock)
{
  mlock->next = NULL;
  mlock->state = UNLOCKED;
  ATOMIC();
  prev = glock
  *glock = mlock
  END_ATOMIC();
  if (prev == NULL) return;
  mlock->state = LOCKED;
  prev->next = mlock;
  while (mlock->state == LOCKED)
    ; // SPIN
}
```

```
ReleaseQLock(*glock, *mlock)
{
  do {
    if (mlock->next == NULL) {
      x = CMPXCHG(glock, mlock, NULL);
      if (x == mlock) return;
    }
    else
    {
      mlock->next->state = UNLOCKED;
      return;
    }
  } while (1);
}
```

14) Knowing that glock is the global lock (with initial value of NULL) and `mlock` is space allocated by a thread; after one thread calls acquire, what is the state of glock?

15) Given Q14, briefly describe how the call to release will, in fact, release the lock.

16) Work through the acquire calls for two threads, after the two calls have been made, what is the "stable" state of the machine? That is, what values do glock, and the two `mlocks` contain?

17) After the two threads have called Acquire, one has the lock and the other waits. When the holder of the lock calls release queue lock, briefly describe how the other thread knows the lock is available.

18) If a thread is in the middle of the queue, how many different cache lines will it touch between acquiring and releasing the lock? (Assume glock and each mlock is a different cache line).

19) What happens if a thread in the middle of the queue is context switched out? That is, if it is signaled before it is context switched back in?

Part 3 – Synchronization Barriers

Barriers are when the program wants to synchronize a collection of threads together, often allowing the threads to advance to a new "time" or common unit of work.

```
counter = release = 0; //long ago, far away…
BarrierWait():
    if (AtomicIncrement(counter) == N) {// N threads total
        counter = 0;                    // Reset counter
        release = 1;                    // Release threads
    }else {
        while (release != 1)
            ;     // Spin on release to be 1
    }
```

20) Given the simple implementation above for a barrier, discuss how this would perform versus the implementation in Q20.

21) In order to reset the barrier, we need to set release back to 0. Conceptually, when should this be done?

Given that it is difficult to achieve Q21, barriers can instead be written using a toggle. We replace “1” with a toggle variable to signal thread release.

```
counter = release = 0; //long ago, far away…
toggle = !release;
BarrierWait():
    if (AtomicIncrement(counter) == N) {// N threads total
        counter = 0;                // Reset counter
        release = !release;         // Release threads
        toggle = !toggle;           // Reset signal
    }else {
        while (release != toggle)
            ;     // Spin on release
    }
```

22) This implementation has a bug in it. Identify why this barrier may not work consistently and what its intended operation is.

23) Due to various circumstances, some of the threads may not be running when the barrier is released. In such a scenario, one thread may reach the barrier for time N+1 while other threads are still sleeping on barrier for time N. Does the toggle implementation prevent this thread from executing the barrier for time N+1 early? Why or why not?

24) In the above algorithm, on what is there contention?

25) Conceptually, how might the barrier be modified to reduce contention?