Extended Model Answer

Advanced Systems Topics 2005 – Paper 8 Question 5 (KAF)

Simple multi-reader spin-lock

2 marks for each of the four operations; 2 marks for the data layout. A very simple spin-lock is all that is required, but more complex solutions are acceptable.

The lock's status is stored in a single word of memory. The least-significant bit is used to indicate an active writer (W). The remaining bits store the number of active readers (R). The lock protocol must ensure that $W \neq 0 \Rightarrow R = 0$ and $R \neq 0 \Rightarrow W = 0$.

```
WriteLock(lock):
 do { val = *lock; }
 while ( (val != 0) ||
         (CAS(lock, 0, 1) != 0) );

WriteUnlock(lock):
 *lock = 0;

ReadLock(lock):
 do { val = *lock; }
 while ( ((val & 1) != 0) ||
         (CAS(lock, val, val+2) != val) );

ReadUnlock(lock):
 do { val = *lock; }
 while ( CAS(lock, val, val-2) != val );
```

Simple spin-lock scalability

This answer is more detailed than required in the exam.

When the lock becomes available (because all readers or the single writer have exited their critical regions) every thread that is waiting to acquire the lock will 'stampede' to acquire the lock's cache line for exclusive access. This increases the average time to acquire or release the lock, and reduces lock scalability.

Two possible fixes are:

(*a*) Add exponential backoff to the testing loop in the lock operations. This will 'spread out' the stampede, but of course will increase latency for threads that just miss observing that the lock is available during their penultimate testing round.

(*b*) Implement a list-based spin-lock with local spinning. An sequence of adjacent readers will wake each other up in turn. Subsequent readers that attend the lock can also proceed directly into their critical sections. When the final reader exits its critical section it checks if any writer is waiting and unblocks it if so.

Improving scalability

2 marks each for any three sensible suggestions.

Some possible suggestions:

(*a*) An improved lock implementation improves scalability (for example, list-based lock rather than a single-word lock). This is most beneficial when the critical region protected by the lock is not very expensive.

(*b*) Use a more permissive type of lock (for example, multi-reader instead of simple mutual-exclusion). This is most beneficial when a significant proportion of threads are not updating shared state.

(*c*) Use a larger number of locks, each of which protects a smaller section of the shared data structure. This is most useful when threads are attempting to access disjoint sections of the shared data.

(*d*) Send work requests to a smaller number of worker threads via efficient work queues. This is useful when there are real data conflicts between work requests that cannot be resolved by any of the previous suggestions. In such cases it is often best to simply serialise work requests by queuing them for a single worker thread. This may also benefit cache locality.