Using Lock-Free Objects in Hard Real-Time Applications[*]

James H. Anderson and Srikanth Ramamurthy
Department of Computer Science, The University of North Carolina at Chapel Hill

Lock-based approaches to object sharing are the accepted means of interprocess communication in real-time systems. The main problem that arises under such approaches is that of priority inversion, i.e., the situation in which a given task[1] waits on another task of lower priority to exit a critical section. Mechanisms such as the *priority ceiling protocol* (PCP) [3] are used to solve this problem. The PCP requires the operating system to identify those tasks that may lock a semaphore, which results in additional complexity in operating system services. This information is used to ensure that the priority of a task holding a semaphore is at least that of the highest-priority task that locks that semaphore.

In [1], we propose using lock-free objects as an alternative for object sharing in real-time systems. Lock-free objects are usually implemented using "retry loops". For example, in the universal, lock-free implementation presented in [2], a task performs an operation by repeating the following steps: first, a shared object pointer is *load-linked* and a local copy of the object is made; then, the desired operation is performed on the local copy; finally, a *store-conditional* is performed to attempt to "swing" the shared object pointer to point to the local copy. These steps are repeated until the last step succeeds.

The main contribution of [1] is to derive scheduling conditions for periodic tasks that share lock-free objects on a uniprocessor. This work pertains to hard real-time systems,[2] and encompasses both static and dynamic priority schemes. The scheduling conditions we derive show that for hard real-time applications on a uniprocessor, lock-free objects often incur less overhead than either wait-free objects or lock-based objects implemented using the PCP.

From a real-time perspective, lock-free objects are of interest because they do not give rise to priority inversions, and can be implemented with minimal operating system support. Despite these advantages, it may seem that unbounded retry loops render such objects useless in hard real-time systems. Nonetheless, we show that if tasks on a uniprocessor are scheduled appropriately, then such loops are indeed bounded. We now explain intuitively why such bounds exist. For the sake of explanation, let us call an iteration of a retry loop a *successful update* if it successfully completes, and a *failed update* otherwise. Thus, a single invocation of a lock-free operation consists of any number of failed updates followed by a successful one.

Consider two tasks T_i and T_j that access a common lock-free object B. Suppose that T_i causes T_j to experience a failed update of B. On an uniprocessor, this can only happen if T_i preempts the access of T_j and then updates B successfully. However, T_i preempts T_j only if T_i has higher priority than T_j. Thus, there is a correlation between failed updates at one priority level and successful updates at higher levels. The maximum number of successful updates within a time interval can be determined from the timing requirements and code of each task. Using this information, it is possible to determine a bound on the number of failed updates in that interval. Intuitively, a set of tasks that share lock-free objects is schedulable if there is enough free processor time to accommodate the failed updates that can occur over any interval. This insight is the basis of the scheduling conditions we derive.

References

[1] J. Anderson, S. Ramamurthy, and K. Jeffay, "Real-Time Computing with Lock-Free Shared Objects", Technical Report, Department of Computer Science, The University of North Carolina, 1995.

[2] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 5, 1993, pp. 745-770.

[3] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time System Synchronization", *IEEE Transactions on Computers*, Vol. 39, No. 9, 1990, pp. 1175-1185.

[*]Work supported by NSF Contract CCR 9216421.

[1]A *task* is a sequential program that is invoked in response to an external stimulus or timer, and that must complete execution by a specified *deadline*. Tasks are usually prioritized and multiprogrammed on a single processor.

[2]Such systems must guarantee that no deadline is ever missed.