Lock

by Frederick L. Hiltz, Stage Manager

M programmers know that the LOCK command grants exclusive ownership of its argument—if your lock succeeds, then no one else's will. We usually employ LOCK to prevent two programs from simultaneously updating the same global variable, but it has more general utility to synchronize concurrent processes.

Many database management systems impose locking for every access to every record. M programmers, however, can lock only when it matters; a key to the superior performance of a well-crafted M database. To achieve that performance, we must consider the consequences of LOCK in real applications.

Scaling up

It is easy to design an application thinking "when in doubt, lock." The program runs well during testing, but in production its performance is terrible. Why? Thousands of M programs run in today's large distributed systems, and hundreds of them may refer to the database at the same time. Unnecessary locking makes them wait in line for access, so concurrency suffers. In addition, the network messages that coordinate the locks cause delays.

Copies of one global variable may reside in many caches throughout the network, and programs may change any of the copies. Which change gets written to the disk? The problem is similar to the case of several programs on one computer changing a global variable, solved by locking. LOCK creates network traffic increasing with the size of the global being locked (remember that the scope of lock is not only its argument but also all ancestors and descendants of its argument):

· Messages to all computers cause them to invalidate the caches that contain their copies of the global data.

· The next access to any of those copies causes a fresh copy to be transferred from the one process that now owns the lock.

This "cache flush" does not prevent other programs from changing the variable that is locked—successful locking still requires the cooperative use of LOCK by all programs. The cache flush does assure that the value written to disk is the one set by the program that owns the lock, rather than values left in caches before the lock was granted. Beaman and Althouse give more detail in their description of Distributed Cache Protocol [1]. Note that the M standard requires only exclusive access to the LOCK argument; it is silent on the cache behavior of the same-named global variable. We shall see how to take advantage of the distinction.

The watchword, then, is to lock as little as possible as seldom as possible:

· Consider a LOCK when writing but not when reading, permitting one writer but many readers of a datum. The value read by a single global reference may not be the latest, but it will be consistent, which is satisfactory for many applications. Consistency across several global references, however, requires a LOCK when reading, or preferably transaction processing.

· Lock the lowest possible level of the variable to maximize concurrency. For example, lock the record of one part in an inventory, not the entire inventory global.

· When locking to synchronize tasks that do not need mutual exclusion on global data, reduce network traffic by making the LOCK argument different from any global name.

· The consequence of not locking may be acceptable. When counting the uses of a program, who cares if two tasks increment the counter at the same time, thereby missing one in a million counts?

· Ask your MDC representative to act on an "atomic increment 77" proposal, which allows a more efficient implementation of the common "lock-increment-unlock" sequence. MIIS programs have used it successfully for years.

· Substitute transaction processing [2] for the homemade equivalent built with LOCK. Performed by the language processor instead of the application, TP is easier to write and understand, potentially faster, and definitely more reliable. It also avoids the deadly embrace.

The Deadly Embrace

Before 1990 the LOCK command always released all locks held by the process before requesting exclusive ownership of the locks in its argument. Such behavior discouraged modular programming—a subroutine could not lock what it needed without releasing all locks held by its caller. A conscious decision by the language designers traded this disadvantage for guaranteed freedom from the deadly embrace.

The increasing importance of modular programming led to the addition in 1990 of the incremental LOCK, by which a subroutine can lock and unlock its resources without affecting its caller's locks. The programmer must then manage the situation shown in Figure 1, where two concurrent tasks need to lock two or more resources. My program and yours both succeed with their first locks, then wait forever on their second locks.

```
MINE              YOURS

LOCK +^A          LOCK+^B
  ...               ...

LOCK +^B          LOCK +^A

  ...               ...

LOCK -^A,-^B      LOCK -^A,-^B
```

Fig. 1 Deadly Embrace

The general problem of deadlock avoidance and resolution is difficult and not well-solved by practical operating systems. Several methods, however, are available for application programs:

· Be greedy. Lock everything you might need at the beginning of your program and hold it until done. This technique yields poor concurrency.

· Lock items in an agreed order, for example alphabetically, which may not be practical in a complex program with many possible orderings for locks.

· Use timed locks and abort your operation if they fail, or unlock the locks you hold and repeat the operation—homemade transaction processing. This is often a good approach for routines with well-defined isolated operations, like filers.

. Use real transaction processing.

· Cop out. "It won't happen in a thousand years, and if it does, the operators will kill the job and restart it." Would your manager approve? Would your conscience? *M*

References

1 Peter D. Beaman and John J. Althouse, "An Efficient MUMPS Distributed Database Using a High Level LAN Interface," MUG Quarterly, 19:3, (1989).

2 Roger Partridge and Joel Segel, "Transaction Processing in MUMPS," MUG Quarterly, 21:5 19-25, (1991).

Frederick L. Hiltz, Ph.D., develops medical information system software at Brigham and Women's Hospital, Boston, Massachusetts. fhiltz@bics.bwh.harvard.edu

---